Extending the Object Teams Programming Model into Distributed Environments

vorgelegt von
Diplom-Informatiker
Abdullah Odeh Al-Zaghameem
aus Tafila, Jordanien

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr. Ing.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Ben Juurlink
Berichter: Prof. Dr. Ing. Stefan Jähnichen
Berichter: Prof. Dr. Samir Abou El-Seoud

Tag der wissenschaftlichen Aussprache: 19. September 2012

Berlin 2012

D 83

To the Soul of My Mother

To my kindhearted Father

To my wonderful wife … Rana

To my kids: Ahmad, Omar, and Noor

I dedicate this work.

Acknowledgments

I would like to express my sincere thanks to everyone helping me to accomplish this dissertation.

Above all, I would like to thank my wonderful wife Rana for her sacrifice and great patience all the time.

I am so thankful to my supervisor, professor Stefan Jähnichen, for giving me this precious opportunity in SWT-group to perform my research, and for his guidance and valuable advice without which this work won't be done.

I am most grateful to professor Samir Abou El-Seoud for his kindness and very valuable notes. I highly appreciate his acceptance for being a second-reviewer on my dissertation.

I would like to give my special thanks to Stephan Herrmann, Marco Mosconi, and Andreas Mertgen; they lit a candle for me at my beginning and continue providing me with their valuable advice and support to the end. Thanks go to Steffen Helke and Roger Burmeister for their friendship, support and kindness. I am indebted to you my dear colleagues.

I will not forget to thank in particular Mark-Oliver Reiser for his unrivaled assistance, support and advice. Thank you Oliver for making time in your busy schedule and listen to me. Whenever I was in a real need for someone to hear me, I found you.

Dr. Haytham Al-Sarrayrih, Akram Al-Kouz, Sadeq Abdelall, Mustafa Al-Bado, many thanks to you my dear friends.

Best thank to SWT-group; my big family in Germany.

Abdullah O. Al-Zaghameem
Berlin
September, 2012

Abstract

The growing complexity of distributed applications, as well as changes in their execution environments, demands for applications that are more adaptable and easier to compose and evolve. Often such distributed applications are required to adapt during run-time, which calls for *dynamically adaptable distributed applications*. The dynamic adaptation of distributed application objects is the process of enabling these objects to change their behaviors dynamically at runtime as a response to changes in their execution environment. Lately, the Aspect-Oriented Programming (AOP) technique has been employed in distributed programming due to its prosperity to improve applications' modularity. It allows separating those crosscutting concerns that are tangled and/or scattered in application code, and capturing them in *aspect* modules. For this purpose, several distributed AOP approaches like AWED, JAC, etc. have been developed. These approaches enable application objects to adapt through replacing their *methods* with new code segments called *advices* at specific points designated by *remote pointcuts*.

From an AOP perspective, the dynamic adaptation of distributed applications requires weaving aspects dynamically at runtime. The current distributed AOP models lack for supporting consistent *dynamic aspects weaving*. Besides, in these approaches there is no explicit representation of the context in which aspects are applied, which reduces their expressivity to specify in a clear and understandable manner how distributed applications can adapt and when? In addition, application developers do not have the proper mechanisms to control the effects of aspects on application base objects dynamically; because most of these approaches do not allow aspect instances to be accessed explicitly.

Object Teams is a programming model that implements the collaboration-based (role-based) design for the object-oriented languages. It employs the AOP concepts to separate the crosscutting collaborations within application objects: it captures collaborations in modules called "teams", and the participation of application objects inside these teams within "role" modules. In this research, we map the fundamentals of the Object Teams model to distributed environments to improve the modularity of adaptable distributed applications. Through employing the expressive "playedBy" relationship in distributed applications, we enhance the modularity of distributed collaborative applications, and support distributed aspects at runtime.

The dissertation introduces a new programming concept called the Remote Role Playing (RRP), which aims at enabling the objects of distributed applications to *play* different roles dynamically at runtime and remotely from any application node. This will provide a modular and expressive technique for realizing the dynamic adaptation of distributed applications. The Distributed Object Teams for Java (DOT/J) framework is introduced as an implementation of the RRP by extending the OT/J (the language that implements the OT model for Java language). DOT/J augments the OT/J infrastructure with the *DOT/J Transformation Library (DTL)*, and establishes the *DOT/J Runtime System (DRS)* to facilitate the RRP between distributed objects and roles. The dissertation demonstrated the capability of RRP to improve dynamic adaptation, and realized other sub-goals like improving distributed aspects' modularity and expressiveness. The approach has been evaluated in a set of case studies, and compared to counterpart distributed-AOP approaches. Finally, the dissertation discussed some future works and further DOT/J enhancements.

Abstract

Die zunehmende Komplexität verteilter Anwendungen sowie veränderte Ausführungsumgebungen erfordern Anwendungen, die leichter adaptiert, kombiniert und weiterentwickelt werden können. Häufig müssen derartige verteilte Anwendungen auch zur Laufzeit adaptiert werden, es werden somit *dynamisch adaptierbare, verteilte Anwendungen* benötigt. Die dynamische Adaptierbarkeit der Objekte einer verteilten Anwendung bedeutet, diese Objekte in die Lage zu versetzen, ihr Verhalten in Abhängigkeit von Änderungen ihrer Ausführungsumgebung zur Laufzeit dynamisch zu verändern. Als ein vielversprechendes Mittel zur Verbesserung der Modularität von Anwendungen wurde in letzter Zeit das Konzept der Aspektorientierten Programmierung (AOP) auch im Bereich der Programmierung verteilter Anwendungen eingesetzt. Es erlaubt querschnittliche Funktionalitäten (sog. crosscutting concerns), die an einer Vielzahl von Stellen im Programmcode auftreten, herauszulösen und zusammenhängend in separaten Aspektmodulen zu definieren. Hierzu wurden eine Reihe von verteilten AOP-Ansätzen entwickelt, wie zum Beispiel AWED und JAC. Diese Ansätze erlauben die Adaptierung von Objekten der Anwendung durch Austausch ihrer Methoden mit neuen Code-Segmenten, den sog. *advices*, an bestimmten, durch sog. *remote pointcuts* definierten Stellen.

Aus Sicht der AOP erfordert die dynamische Adaptierung verteilter Anwendungen ein dynamisches Einweben von Aspekten zur Laufzeit. Die derzeitigen Ansätze zur verteilen AOP bieten jedoch keine Unterstützung für ein konsistentes dynamisches Weben von Aspekten. Außerdem gibt es in diesen Ansätzen keine explizite Repräsentation des Kontextes, in dem Aspekte angewendet werden. Dies erschwert eine klare und verständliche Beschreibung, wie und wann eine verteilte Anwendungen adaptiert wird. Darüber hinaus geben sie den Entwicklern keine geeigneten Mechanismen an die Hand, um die Auswirkungen von Aspekten auf die Basisobjekte der Anwendung dynamisch zu steuern, da die meisten dieser Ansätze keinen expliziten Zugang zu Aspektinstanzen erlauben.

Object Teams ist ein Programmiermodell, welches den kollaborationsbasierten (rollenbasierten) Entwurf in objektorientierten Sprachen bereitstellt. Es verwendet die AOP-Konzepte um querschnittliche Kollaborationen in Anwendungsobjekten zu separieren: es faßt Kollaborationen in Modulen zusammen, die als „Teams" bezeichnet werden, und beschreibt die Teilnahme von Anwendungsobjekten innerhalb dieser Teams in „Rollenmodulen". In dieser Dissertation werden die grundlegenden Konzepte von Object Teams auf verteilte Anwendungen übertragen um die Modularität adaptierbarer verteilter Anwendungen zu verbessern. Durch Verwendung der ausdrucksmächtigen „playedBy"-Beziehung in verteilten Anwendungen wird die Modularität verteilter, kollaborativer Anwendungen verbessert und verteilte Aspekte zur Laufzeit unterstützt.

Die vorliegende Forschungsarbeit führt ein neues Konzept mit dem Namen *Remote Role Playing* (RRP) ein. Dies erlaubt es den Objekten einer verteilten Anwendung zur Laufzeit dynamisch verschiedene Rollen innerhalb der Kollaborationen von Object Teams

einzunehmen, und dies von jedem beliebigen – auch von entfernten – Ausführungsknoten aus. Hiermit steht eine modulare und ausdrucksmächtige Technik zur dynamischen Anpassung verteilter Anwendungen zur Verfügung. Es wird das *Distributed Object Teams for Java* (DOT/J) Framework vorgestellt, das die bestehende Sprache OT/J (die Implementierung des Object Teams-Programmiermodells in Java) um RRP erweitert: hierzu wurde die Infrastruktur von OT/J um die *DOT/J Transformation Library* (DTL) ergänzt und das *DOT/J Runtime System* (DRS) bereitgestellt, um das RRP zwischen verteilten Objekten und Rollen zu vereinfachen. In der Arbeit wird die Effektivität des RRP zur Verbesserung einer dynamischen Adaptierbarkeit verteilter Anwendungen demonstriert und es werden eine Reihe weiterer Vorteile aufgezeigt, wie die Verbesserung der Modularität und Ausdrucksmächtigkeit verteilter Aspekte. Der Ansatz wurde im Rahmen mehrerer Fallstudien evaluiert und mit anderen Ansätzen zum verteilten AOP verglichen. Abschließend werden Möglichkeiten für zukünftige Arbeiten und Erweiterungen von DOT/J aufgezeigt.

Contents

Chapter 1

Introduction

Software Engineering has been rapidly growing over the past two decades due to major developments in hardware (like the high speed of processors, large Internet bandwidths, etc) and programming techniques. This development leads to new user and application requirements like mobility, applications adaptation [65], context-awareness [13 and 49], etc.

The term "*dynamic adaptation*" has been introduced by Valetto *et. al.* [30] to refer to modifications in behavior and structure (which could be made at the granularity of application objects, a set of objects, or the overall system) such as adding, removing or substituting components at runtime without "bringing the application down." A review of the state of the art on dynamic adaptation has been established in [39] and re-published in [40]. Dynamic adaption has gained its importance due to the real need for supporting runtime software composition and seamless systems upgrading [30]. In addition, dynamic adaptation is a prerequisite to realize *ubiquitous computing* [60 and 45], in which computer software should adapt to the surrounding physical environment, and to the virtual environment of computing and communication networks [80]. However, the achievement of dynamic adaptation is considered a major challenge in mobile and distributed systems [41].

Currently, the demand for developing *dynamically adaptable distributed applications* and ubiquitous systems is highly increased [45]; because the environments, on which these applications execute, become more mobile and changeable. Reasons to changes in execution environments include network bandwidth variation, changes in the underlying protocols, and the need to apply new algorithms dynamically at the application or platform levels in order to improve application performance. In the Object-oriented Model (OOM), dynamic adaptation of distributed application objects is the capability of these objects to adapt to changes in their executing environment by changing their *behaviors* dynamically at runtime without interrupting application execution. Relevantly, the behavior of an object is implemented by the *methods* declared in its descriptor module (e.g. in its class type).

Aspect-Oriented Programming (AOP) [28], on the other hand, is a programming technique that aims at improving the quality of software by decreasing the level of code scattering and tangling [82]. It proposes a modular solution for the problem of separation of crosscutting concerns by encapsulating these concerns in new modules called *aspects*. An aspect module comprises declarative expressions called *pointcuts*, which specify where in program core-code the separated functionalities (called *advices*) should be *injected* using a mechanism called *Aspects Weaving*. The employment of new functionalities or replacing specific existing ones at the granularity of objects methods using AOP technique is considered a modular mechanism to enable these objects to adapt and change their behaviors.

The prosperity of AOP technique to improve applications' modularity inspires employing its concepts in distributed programming. Therefore, several distributed AOP approaches like AWED [47], DyMAC [11], JAC [70] and DJcutter [55] have been developed. From a dynamic adaptation perspective, these approaches enable application developers to declare new functionalities (i.e. advices) inside aspects, and provide a mechanism to *deploy* and *weave* these aspects in application base classes. To support the dynamic adaptation of distributed application objects at runtime, aspects need to be woven dynamically without stopping application execution; whereas not all these approaches support the dynamic weaving of aspects. The distributed AOP languages that support dynamic aspect weaving, on the other hand, do not provide language constructs that support consistent aspect weaving [51]. Another obvious, and yet common, shortcoming of these approaches is their lack for a mechanism that enables application developers to control aspects' application at runtime like stopping and resuming aspect advising modularly [2].

Furthermore, they lack a clear and expressive relationship between aspects and application base classes, which is highly desired to simplify developing dynamic adaptable distributed applications. This shortcoming is resulted because of that these approaches do not modularize the contexts of aspects employment. That is, application developers declare aspects without specifying in what context these aspects are employed. Besides, some problems emerge from the fact that aspects in distributed AOP models are too powerful and unrestricted [36], which can lead to errors when semantics of the original applications is modified unexpectedly.

In this dissertation, besides discussing the aforementioned shortcomings, a study is made for the Object Teams [73] programming model, which aims at implementing the collaboration-based (role-based) design [75, 44, 59, 32, 58 and 52] for the Object-Orientation Model (OOM). It employs the concepts of AOP to separate collaborations that crosscut several application objects, and captures them in new modules called "*teams.*" Also, it captures the participation of application objects in collaborations within new modules called "*roles*." The features of this model will be explored, and a discussion will be made on how it can facilitate adapting the behaviors of application objects in an expressive and modular fashion. Then, the dissertation explains the Object Teams (OT) model's lack for supporting the dynamic adaptation of behaviors of distributed application objects, and investigates the barriers which prevent employing fundamental features like "role-playing" in distributed environments.

This dissertation proposes the Distributed Object Teams/Java (DOT/J); a new framework for mapping the features of the OT model to Java-based distributed environments [1]. The mapping process aims at enabling Java-based distributed application objects to *adapt* by changing their behaviors *dynamically*. For this purpose, the dissertation introduces the *Remote Role-playing (RRP)* relationship, which aims at connecting the separated application objects (called *bases*) and the roles of teams preserving the semantics of the local base-role relationship (called *playedBy*) in OT model. In this way, application objects can adapt by the employment of new teams by playing their roles dynamically at runtime and remotely from anywhere. The carrying out of this mapping simplifies developing dynamically adaptable distributed applications, and reduces the complexity of their composition by using *remote* team and role modules, besides the expressive RRP relationship, which glues base objects and role instances. The dissertation emphasizes the advantages of this approach over counterpart distributed-AOP approaches [47, 55, and 51] through a set of case studies.

1.1 Background

This section paves the road of discussion by presenting briefly the basis of the OT programming model. Then, it gives a quick overview of Aspects with Explicit Distribution (AWED) [47] as an example of distributed AOP approaches. The section explains its programming model, and discusses how it can support the behavior adaptation of distributed applications. AWED had been cited because it combines AOP and distribution concepts like remote pointcuts and remote advice execution, in addition to its support for dynamic aspects weaving at runtime. The section introduces OT/J [75] (the Java-based implementation of OT model), and emphasizes its AOP features. Finally, it introduces the concepts of distributed applications and how to program them using the Java-RMI [83] middleware; a simple and powerful object-oriented middleware for developing Java-based distributed applications.

1.1.1 The Object Teams Model – An Overview

The object-oriented Collaboration-Based Design (CBD) describes a methodology for decomposing object-oriented applications into a set of classes and a set of collaborations [52]. In a CBD of object-oriented application, an object of specific class can participate in several collaborations, where a single collaboration may spin several objects from different classes as shown in Figure 1.1. Each object can *play* a particular *role* within each of the collaborations. In this regard, role is considered the fundamental concept for building collaborative applications [31]. For example, in Figure 1.1, objects of **ClassA** class play the role **RoleA1** in the collaboration **Collaboration1**. In the programming world, the OOM encapsulates the description of objects within modules (e.g. *classes*), but lacks a similar module concepts that can capture collaborations and roles.

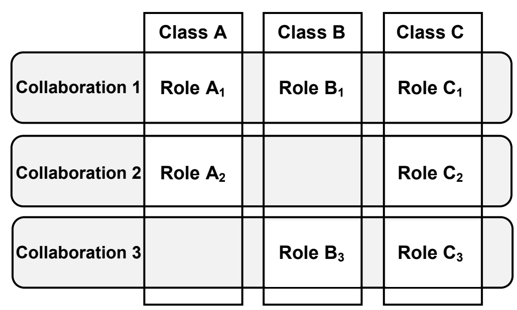

Figure 1.1. Dimensions of application classes and collaborations crosscutting (Source [73]).

According to Figure 1.1, collaborations are *crosscutting* several classes. With this observation, the OT model benefits from the modules composing technique proposed in the context of AOP to separate these crosscutting collaborations within modules called *teams*. The intersection between collaborations (or teams) and system classes (called *base* classes) could be defined as the *behavior* of base objects (which are generated from base classes) inside a specific team. The OT model captures and encapsulates the behavior of each base class in each team within a *role* module. By this conception, the term “team” could be redefined as a module that encloses roles.

For modeling purposes, the OT model uses special modeling diagrams which extend the UML diagrams, called *UML for Aspects* (UFA) [74], and enable developers to design OT-based applications diagrammatically. For example, **Collaboration1** shown in Figure 1.1 could be represented using UFA as shown in Figure 1.2. The figure illustrates **Collaboration1** as a team package that encloses three roles: $RoleA_1$, $RoleB_1$, and $RoleC_1$.

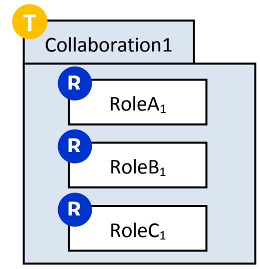

Figure 1.2. Collaboration1 as a team diagram.

In the OT model, application base classes and their roles are bound within teams through a relationship called *playedBy*. This relationship involves a communication channel between base objects and the corresponding role instances at runtime. In this channel, control flow is mutually exchanged through two types of *bindings*, namely: the Call-In Method Binding (CIMB) and the Call-Out Method Binding (COMB). These bindings are the only link between *base* and *role* objects.

Now, it is possible to construct a complete design for **Collaboration1** using UFA emphasizing the *playedBy* relationships as shown in Figure 1.3 (b), in which each base class is bound to a specific role. Note from part (a) of the figure that "team" packages are combined to application *core-code* through "adapt" relationships. This application of teams to a core application module has the effect of adapting that application [74].

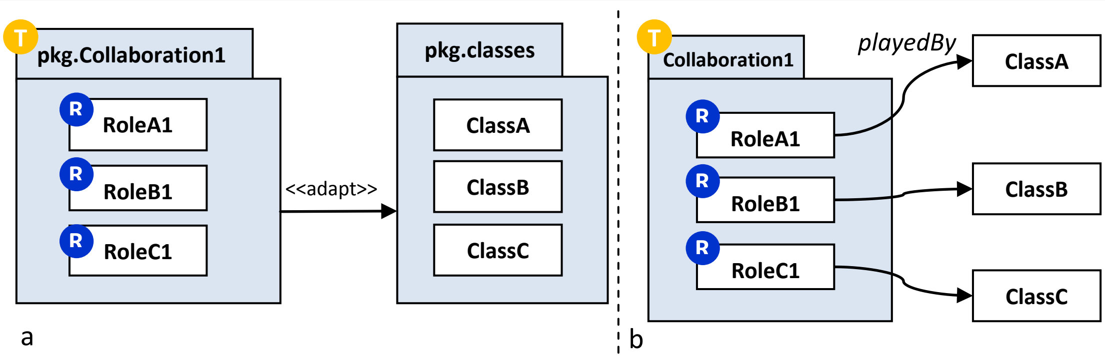

Figure 1.3. A UFA diagram to Figure 1.1 in (a) and *playedBy* relationships are shown in (b).

To illustrate the *playedBy* relationship and its underlying CIMB and COMB channels, let us take a real example. Figure 1.4 shows a *playedBy* relationship between the **Person** class and the role **Employee** in the **Company** team. This relationship could be simply read as follows: "*in a company, persons are employees, or a person can play the role of employee in company collaboration*." Note that the **Company** team can have its own methods and attributes. Likewise, the role **Employee** can declare its own methods and attributes, which emphasizes its individuality.

The figure shows a *callin* link between **Person** and **Employee**; which depicts the CIMB channel. The expression [`getEmpInfo` ← **replace** `getInfo`] is called a CIMB declaration. It will intercept the calls of method `getInfo()` in **Person** objects, and *replaces* them with calls to the role's method `getEmpInfo()`, which is equivalent to *aspect's advice* in the AOP. This is to say, persons who are employed in the company should *adapt* their behavior; specifically this method, because they acquire new information like employee number and salary. The name "callin" depicts the way the control flow is dispatched from a *base* objects (in this example the objects of **Person** class) *into* their roles.

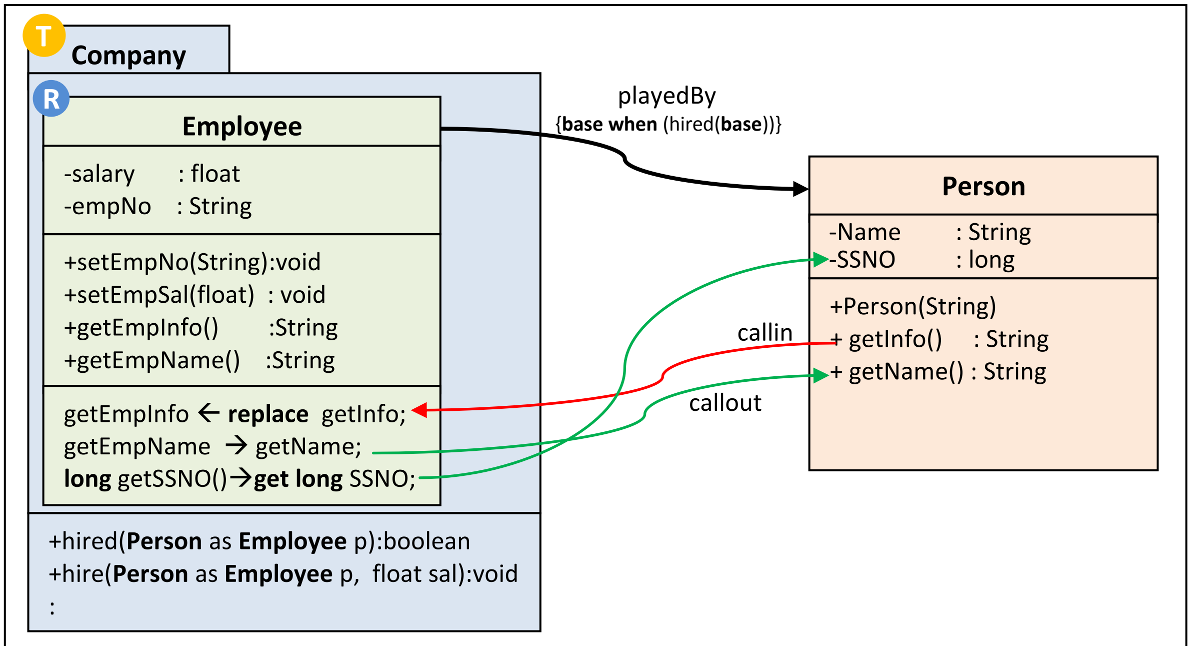

Figure 1.4. A simple OT/J application: **Person** class plays the **Employee** role within the **Company** team.

The figure illustrates a *callout* link that connects the method getEmpName() of **Employee** role with the method getName() of **Person** class. The expression [getEmpName → getName] is a COMB declaration. The name "*callout*" depicts the mechanism for dispatching the control flow from inside the role outward into its base class. In this example, whenever the method getEmpName() is invoked in a role instance of **Employee**, then the method getName() of the *bound* base object will be invoked instead.

The OT model presents three *callin binding modifiers*; *after, before,* and *replace* which designate, in base objects, the execution points at which the CIMBs are matched. Practically, the CIMB channel and its modifiers are considered as a key concept for enabling base objects of an application to *adapt* through replacing *their original functionalities* with *roles' callin methods*. A generalization of this idea could be done as follows: "*base objects considered adaptable if they can play different roles within different teams*." This means that, for example, a **Person** object can acquire different behavior when it plays the **Parent** role in a **Family** team, or the **Student** role in a **Course** team, etc.

Regarding the COMBs, the OT model presents two types of COMB. The first is called *method-COMB* which binds a role's *expected* method with a method (declared or inherited) of a base class, like the one shown in Figure 1.4. The second type is called *field-COMB* which enables a role instance to access the base object's fields through *getting* their values or *setting* new values. For example, the **Employee** role declares the field-getter-COMB [**long** getSSNO()→ **get long** SSNO] which reads the current value of the field SSNO of the base object when the method getSSNO() is invoked in a role instance. Actually, the COMB is a pioneer concept described within the OT model [73], and has no counterpart in other AOP models.

The *playedBy* relationship is constrained by *Guard Predicates (GPs)* [75, 79], which are conditional expressions that enable the refinement of binding between base objects and role instances at runtime. For example, Figure 1.4 shows the *Base Guard Predicate (BGP)* [**base when** (hired(**base**))] which refines the *playedBy* relationship by stipulating "only persons who have been hired by the company are considered employees."

The method hired() of class **Company** team has the special declaration [**Person** as **Employee** p], which is called *Explicit Lifting Parameter* declaration [75, 79]. It stipulates that: "whenever an argument of type **Person** base class is delivered to this method, then it should be *lifted* into a corresponding role instance". Basically, teams need to lift base objects into role instances because they cannot access (precisely, prohibited to access) these objects directly. A discussion of base lifting and other advanced issues will be made in more detail in Chapter 2.

1.1.2 Distributed Aspects with AWED

Aspects with Explicit Distribution (AWED) [47] is an AOP language with explicit support for distribution. The current implementation of AWED is a prototype based on DJAsCo [87]; in which many language-specific features are still unimplemented. DJAsCo augments JAsCo (Java Aspects for Components) [17] with a set of distributed features (discussed in the following paragraphs). JAsCo is a dynamic AOP language tailored for the component-based models. It extends the Java language with *Aspect bean* and *Connector* constructs. An aspect bean is a module describing crosscutting concerns (i.e. advices) independent of application base objects' types. One or more advices could be declared in a *hook* construct. A hook construct specifies in its constructor an *abstract* pointcut, and then provides the actual definition of that pointcut. A connector, on the other hand, deploys one or more aspect beans within application components, and allows specifying the combination of their aspectual behaviors by using combination strategies.

AWED extends JAsCo with: (1) Remote pointcuts that can match joinpoints on remote hosts including remote calls, (2) Distributed advices that are executed on remote hosts, and (3) Distributed aspects support which enables aspects to be configured using different deployment and instantiation options.

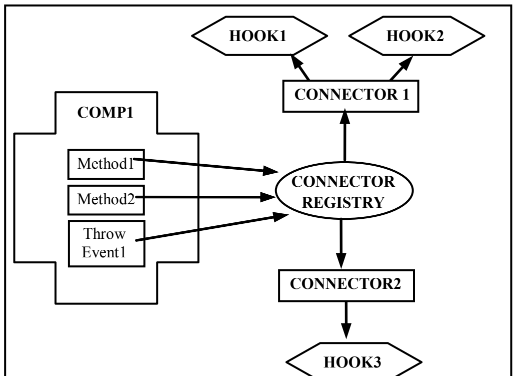

Figure 1.5. JAsCo architecture [17].

The architecture of JAsCo is shown in Figure 1.5. The central connector registry acts as the main addressing point for all JAsCo entities. It is notified when a new connector has been loaded, or when a *trap* has been reached. In JAsCo (hence AWED), all methods of a target application component (e.g. COMP1 shown in Figure 1.5) are equipped with traps to allow their execution deferred to the connector registry. The connector registry executes those matched pointcuts that are declared in the deployed aspects. DJAsCo extends this architecture by allowing several connector-registries to communicate with each other to preserve synchronization in case of remote joinpoints matching and remote advices execution.

Listing 1.1 demonstrates a simple AWED example. In part (a), a simple JAsCo aspect-bean named **MessengerAsp** is defined, which declares the hook **MessengerHook**. The hook's pointcut matches calls of a base object method which is not yet bounded. In addition, the pointcut matches calls on hosts other than the local host; as specified at line 5.

In part (b), the connector **MessengerCon** is defined to bind a **MessengerHook** instance with the method **processMessage** of base class **Messenger** (not shown in the example)[1] at line 2. In line 3, the connector specifies when to trap **processMessage** method calls by that hook.

Listing 1.1: A simple AWED example: (a) an aspect-bean, and (b) a connector implementation.

a

```
1. package all.pkg;
2. class MessengerAsp {
3.   hook MessengerHook {
4.     MessengerHook(method(String s))
5.     { execution(method) && !joinpointhost(localhost); }
6.   before() {//...;}
7.   }
8. }
```

b

```
1.static connector MessengerCon
2.{ all.pkg.MessengerAsp.MessengerHook hook0 = new all.pkg.MessengerAsp.MessengerHook
            (void all.pkg.Messenger.processMessage(String));
3.  hook0.before();
4.}
```

The separation between aspects declaration and the process of binding their hooks to application components allows the aspects *reusability* in AWED; for example, developers can bind several instances from the **MessengerHook** hook with different base methods in different base classes. Though, this reduces aspects' expressivity and clarity. In addition, the architecture of DJAsCo enables new aspects to be woven dynamically at runtime. That is, the application developer can declare a new aspect with the required hooks (i.e. advices), and binds the desired base methods with aspect's hooks via connectors. The connector registry in DJAsCo is implemented with the capability to automatically deploy new attached aspects on remote hosts at runtime. This capability is considered a key concept to realize the dynamic adaptation of distributed applications in that a new hook's advice can replace a specific base method.

1.1.3 Object Teams/Java – the AOP dimension

Object Teams/Java (OT/J) is the programming language that implements the OT model in Java programming language [75]. In OT/J, teams and roles are first-class modules that could be defined at the source code level. A *role*[2] in OT/J is the counterpart of aspect-bean of AWED, and *aspect* class of AspectJ [27] (the leader of AOP languages), but has several advantages. First, a role module captures the behavior a base class has in a specific team; thus, it offers much more modular mechanism for separating these behaviors that crosscut application classes. Second, a role class presents a clear and understandable advising process by the means of role's *callin methods* (equivalent to a hook's advices in AWED and aspect's advices of AspectJ) and the CIMB expressions. Third, OT/J supports the dynamic applicability of roles by using teams (de)activation mechanism, which is impossible in approaches that *hide* aspect instances like JAsCo and AspectJ. Furthermore, OT/J allows constraining role-playing applicability and callin effects by using GP expressions.

Finally, in AWED (and all distributed AOP approaches in general), aspects and their advices cannot reason about the target base object at which a remote joinpoint has been

[1] This example will be revisited in Chapter 6 and discussed in detail to compare DOT/J and AWED with respect to distributed aspect programming.

[2] OT/J presents two types of roles; bounded and unbounded roles. We refer mostly in this dissertation to the bounded roles that are coupled to specific base class via the *playedBy* relationship [75, 79].

matched. OT/J, on other hand, uniquely offers the COMB as a mechanism to de-capsulate target base objects at runtime. This enables roles to the attributes and stat of their base objects at advice execution time; i.e. at role's callin methods execution.

The relationship between roles and their base classes is highly expressive in OT/J, which adequately reflects the real world features into the programming world. All these properties make applications understandable and their maintenance simple and quick. Furthermore, OT/J can easily answer questions like "*in what context aspects are woven in applications?* [2]".

1.1.4 Programming Distributed Applications using Java-RMI

In OOM, a distributed application is a set of objects that reside at different nodes (network hosts or application processes at the same host), and communicate with each other over a specific middleware in order to fulfill specific tasks. A middleware is an executive layer that facilitates data exchange (like *Messages* transmission) between the separated objects. Examples are CORBA [89], Enterprise Java Beans (EJB) [46], and Java-RMI [83]. The middleware helps programmers to design and implement distributed components appropriately according to middleware disciplines, and provides the necessary mechanisms to access them *remotely*.

Java™ Remote Method Invocation [83 and 94] (Java-RMI for short) is a distributed object-oriented model for the Java programming language. It makes distributed objects easy to implement and to use. Once a programmer implements a specific distributed object using Java-RMI, he/she can invoke that object's methods *remotely* the same way they are invoked locally.

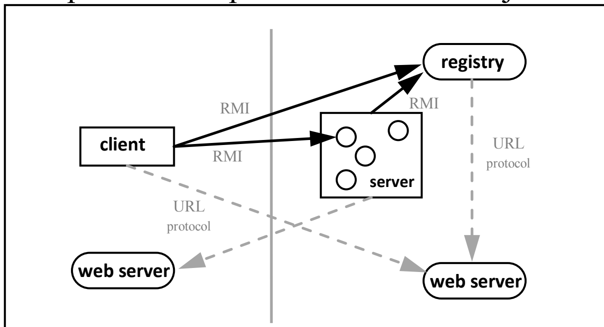

Figure 1.6. A typical RMI-application (Source [84]).

Distributed applications could be easily developed with the Java-RMI. Figure 1.6 shows a typical distributed application in Java-RMI. The remote server objects are registered in a *Registry* by associating an identification name for each one of them. A file server (*web* or *ftp-server*) might be used to *store* the *bytecode* of remote classes (called *stub* classes), which are downloaded on-demand by client programs. The ordinary scenario for accessing remote objects is to *look* them *up* at the registry using the associated identification names. Then, the RMI system downloads the *stub* class from the web-server into the client program. A new *stub object* is then created as a *proxy* for the desired remote object, and clients can start invoking its methods as usual.

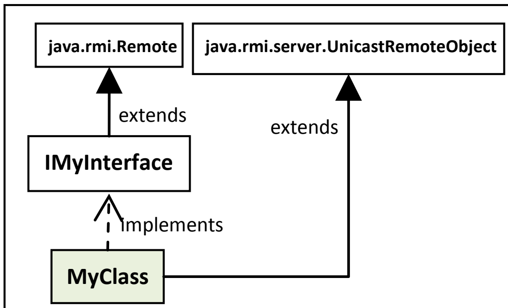

Figure 1.7. A Remote Class must implement a remote interface.

The *traditional* mechanism for implementing remote objects in Java-RMI is straightforward; the class of remote object *must* implement at least a *remote interface*, and extends the Java-RMI class `UnicastRemoteObject` [83], as shown in Figure 1.7. The class **MyClass** implements the **IMyInterface** remote interface, which is a normal Java interfaces that *extends* the Java-RMI marker interface `java.rmi.Remote`. A remote interface declares the remote methods that will be invoked remotely.

One of the most significant features of Java-RMI is *transparency*; specifically, it hides a lot of verbose low-level code for preparing the actual communication *before*, *during*, and *after* remote method invocations. In addition, it takes care of transmitting data and methods' arguments during method invocations. Besides, it garbage collects unreferenced remote stubs over the network. This is carried out without the user perception. Moreover, since Java 1.5, the Java-RMI system supports the *dynamic generation* of remote stubs at client side [88]. This enhancement eliminates the cumbersome process of generating remote stubs manually with the "*rmic*" compiler, and then uploading them at a file server. In this dissertation, Java-RMI is adopted in DOT/J due to its adequacy to fulfill the requirements of RRP especially with issues like transparency and its full compatibility with the OT/J language.

1.2 Dynamic Adaptation of Distributed Application Objects' Behaviors

In general, adaptability of object-oriented applications is the ability of their objects to modify their *behavior* and/or *structure* to adapt to changes in their execution environments. The adaptation of applications aims at enabling them to obtain a desirable level of performance (or to enhance it) [23], or to improve the objects' adequacy to execute new tasks and conform to new requirements. The dynamic adaptation of an application (or the *runtime system evolution* as referred in [69]), on the other hand, is the capability of its objects to adapt at runtime without re-engineering the classes source of these objects, or interrupting the execution of the application.

In this dissertation, the Dynamic Adaptation of Distributed Application Objects (DADAO) is defined as follows [3]:

Definition 1.1: Dynamic Adaptation of Distributed Application Objects (DADAO)

The DADAO is the capability of the distributed application object **dObj** to adapt by changing its behavior by replacing a specific functionality (i.e. a method) with a new one, satisfying the following conditions:

1. The adaptation is performed dynamically at runtime and continuously (i.e. **dObj** can adapt further more).
2. **dObj** must preserve its original behavior and can reclaim it if the cause of adaptation has no longer applied.
3. Consistency of individual objects and the entire application must be ensured after adaptation.
4. The execution of the application must not be interrupted.

In this section, a brief survey of the literature of dynamic application adaptability will be given. The section will shed a light on the different approaches that support adaptable applications, and then present our concept for realizing the DADAO using the fundamentals of the OT model.

1.2.1 Adaptation Levels

Basically, dynamic adaptability of applications could be considered at two levels [34]:

1. *Adaptability of the individual software component*.

Adaptability at this level is carried out by changing the *behavior* of application components so that they acquire new functionalities which empower them to behave differently (as a response to changes in their execution environment). One interpretation of adaptability at this level is to modify the architecture of components' modules and replace the old instances of these components with new ones like in PROSE [9]. Changing the structure of components this way, however, can cause serious deformations in application functionality and impact its integrity especially in case of intensive dynamic adaptation.

A group of non-AOP approaches like MolèNE [41], FORMAware [68 and 72], iPOJO [15] and HADAS [33] have been developed to support adaptation of distributed applications by using *adaptable-ready* components to construct an adaptable application. These components are equipped with the capability of dynamic composition. For example, MolèNE (an object-oriented middleware framework) enables developers to implement adaptive components to facilitate dynamic application adaptability. It separates the functional part of adaptive components from the reaction mechanism to changes in the execution environment. In the same scope, the authors in [23] have recommended that developers should consider, in addition to the implementation of distributed application functional behavior, some other underlying issues should be taken into account like:

- Monitoring of resource usage and application-specific interactions.
- Specifying which environment elements should be monitored.
- A mechanism to detect such environmental changes must be provided.
- Determine software adaptations to be handled and the time to apply them.

In practice, by using the AOP technique new functionalities (in the name of *aspect weaving*) could be *injected* into target application classes at specific joinpoints. Also, some AOP approaches like AspectJ and DJcutter [55] support so-called *Inter-Type declarations*, which enable application developers to perform "internal structure modifications" like injecting new *methods* and *fields* into target classes, and "compositional structure modifications" like defining new interfaces. Thus, it modifies the structure of target classes, which can cause several problems like field and methods *ambiguities*; especially when several aspects target the same base class. For example, inter-type declaring a public method in a specific target class can cause compile-time conflict if that class already implements a method with the same signature [95]. In addition, injecting inter-type declarations at the *bytecode* level leads to hard code these declarations in base and aspect classes simultaneously, which makes realizing the same technique on distributed applications (to facilitate objects adaptability) very hard and error-prone. Most importantly, hard coding aspects in application classes *prevents* accomplishing the DADAO without violating conditions 3 and 4 of Definition 1.1.

In addition, adapting application components using the current AOP approaches creates some interoperability problems in the application. These problems have been discussed in [24] and could be summarized as follows:

- *Aspects' precedence*. It was noticed that pointcuts of new aspects might be applied to the same join point that other (unknown) aspects matched. Then, the expected sequence of interception can't be guaranteed.
- *Unintended aspect effects*. Pointcuts of new aspects may match to undesired join points, which can trigger serious side effects.

- *Partial weaving*. When the code of application is modified, the aspects woven in it may not be applied to further modifications.
- *Unknown aspect assumptions*. It could happen that when pointcuts of new aspects are applied, they may not find any join points matching existing requirements.
- *Failure to preserve state invariants*. When an aspect is applied, it could break the state invariants of the system.

Other AOP approaches like PROSE [9], Lasagne [20] and DandyJ [51] have introduced a dynamic structural adaptation of objects at runtime by using *dynamic weaving* mechanism, which allows developers to *weave* and *unweave* aspects from running applications [63]. Practically, problems of consistency of aspects and the target objects can occur during aspects weaving and unweaving especially if aspects have inter-dependencies to each other. JAsCo supports dynamic aspect weaving [17] through *late binding* the target objects of application to aspect beans via connectors.

In case of distributed applications, AWED does not support completely objects behavior adaptation since it cannot hook onto *invisible* methods, i.e. only public methods could be hooked in aspect beans. Besides, aspects need to be deployed at every application node to simulate remote advice executions. Other approaches like DyMAC [11] implement new frameworks to support aspectual-components in order to employ AOP concepts in component-based systems. However, this and other approaches like JAC [70] and DJutter [55] have a disparate level in supporting adaptability since each one of them represents aspect modules and Joinpoint Models (JPMs) differently. All in all, these approaches success in encapsulating the desired new functionalities needed to achieve DA, but fail to perform the dynamic adaptation itself, besides their lack for expressivity and simple application development. For example, application developers in JAC need to implement four types of programs to perform aspects weaving; namely: the core application, the wrapper program, the weaver program, and composition plan descriptors.

Roles are Extensions of their Base Objects

In the OT/J programming language, bounded roles are "*personalities"* of application base objects inside teams. For example, an *employee* in a *company* is a special personality/characteristic of a *person*. Thus, the behavior of an employee is the behavior of person inside the company. This concept could be generalized in the programming world as follows: "*a bounded role class is a class extension to its base*". Actually, this concept is consolidated in this dissertation with the following facts of OT/J about the relationship between a base object and its role instances:

First: A role instance *can* access all of its base object's methods and fields *systematically* via the COMBs only. This capability is similar to *objects inheritance* in Java [79].

Second: Each of base object's methods could be *overridden* by a role's method via *replacement* CIMB (i.e. a callin method and CIMB with "`replace`" modifier).

Third: Bounded role instances *cannot* be created without the existence of *immutable* base objects.

Therefore, the roles' behaviors inside team instances are considered as an inherent part of the behaviors of base objects. This consideration stems from the fact that the pair (role, base) involves a *tight coupling* between their objects; hence it is considered as a single unit of modularity. To make use of these features to realize expressive and modular DADAO, the following outline could be considered as a RRP fundamental [2, 3]: "*Attaching roles dynamically at runtime (through deploying new team instances) allows application objects to acquire new behaviors. As an object is able to play new roles dynamically, it can change its behavior; thus adapt, due to the effects of this role-playing.*"

2. *Adaptability of application inter-components relationships*.

Adaptability at this level targets the contextual changes of application components like adjusting components' locations (e.g. due to migration over the network from one node to another), or change in the relationships between these components. This adaptation affects the communication environment of the interacting components and not the components themselves – unless contextual changes directly affect components' states. In this dissertation, however, the dynamic adaptation of distributed application objects at the behavioral levels will be studied.

1.2.2 Dynamic Adaptation Features

An important key feature for adapting distributed application objects is the *dynamicity*, i.e. distributed objects should be able to change their behaviors dynamically at runtime. In addition, dynamic distributed application adaptation must be continuous [20]. Continuity of adaptation imposes a mechanism that allows *inserting* new functionalities into, and *withdrawing* others from, applications during execution without interrupting their availability, i.e. without service stopping. In the AOP scope, this technique is known as the *dynamic weaving* of aspects. In case of distributed applications, applying this technique requires weaving distributed aspects *atomically*[1]. Thus, aspects must be woven "simultaneously and consistently" on each application's targeted nodes [51]. At the same time, an acceptable level of transparency and flexibility in the employment of these functionalities should be preserved, and improve the modularity of adaptable distributed applications in a reliable way. Finally, *expressivity* of dynamic adaptation [3] means the capability to modularize application objects' adaptation in an expressive and understandable fashion. Thus, composing adaptation as a concept to applications becomes clearer, which helps to maintain and evolve applications efficiently.

1.3 Motivation

The significant features of OT model and the lack of current distributed-AOP approaches for supporting the DADAO are the main motivations for this dissertation. In addition, these approaches lack a clear and expressive mechanism for composing adaptable distributed applications which could simplify their evolution and maintenance. Besides, current distributed dynamic AOP languages do not provide language constructs that support

[1] Atomicity as defined in [20] is the process of adding or removing the mutually dependent functionalities in "an all or nothing" fashion.

consistent aspect weaving [51]. Mapping the features of the OT model to distributed environments, on the other hand, will enable the *dynamic adaptation* of application objects at the behavioral level if they are *enabled to play different roles within different teams dynamically* at runtime *and remotely* from any application node. The dynamic playing of roles at runtime allows adding new functionalities, which are considered as new *adaptation strategies* [8] for distributed objects, without worrying about aspects weaving atomicity; since the remote role-playing process does not require deploying roles at application nodes. Finally, employing new teams in distributed applications at runtime is considered an *adaptation trigger* (i.e. change in application execution) that imposes application objects to adapt by obtaining new behavior through RRP.

1.4 Challenges for Remote Role-Playing in OT/J

Remote Role-Playing (RRP) is defined as follows [3]:

Definition 1.2: The Remote Role-Playing (RRP)

The RRP is the process of binding a specific application object **obj** to a specific role instance **r** that is confined within team instance **t** such that: **obj** and **t** are logically or physically separated, and base object **obj** is able to play the role **r** according to the semantics of local role-playing in the OT model.

This section discusses the shortcomings of OT/J to enable the RRP; whereas, the dynamic RRP is considered as a primary requirement for realizing the DADAO. This section is divided into two subsections: In the first section, a detailed explanation through an example will be given about the reasons after this shortcoming and, in the second subsection the problem statement will be addressed in points.

1.4.1 Understanding the Problem – A Technical Discussion

With respect to the *playedBy* relationship, roles and their players, as well as roles and their enclosing teams, are *tightly coupled* due to two major reasons: *First*, a role instance can directly access its base methods and fields at runtime through the COMBs. *Second*, role instances must be confined inside the boundaries of team instances; thus, the execution of role's callin methods is handled *transparently* via the enclosing team instance. This mix of coupling imposes that team, role, and base instances must reside in a single Java Virtual Machine (JVM), i.e. in a single application space.

Moreover, in the core of OT/J, two main activities are taking place to realize the *playedBy* relationship: compiling the source code of the OT/J application, and transforming classes' bytecode at the load-time in order to weave the declared roles into base classes.

The compiler of OT/J translates *playedBy* relationships by adding to team classes the *Translation Polymorphism* [77] code segments (which includes the code of *lifting* base objects into role instances and *lowering* roles into base objects – this will be discussed in Chapter 2). At this stage, OT/J's compiler *verifies* the names of all base classes (the class names at the right-hand-side of *playedBy* relationship) to ensure binding role instances with *type-safe* base objects at runtime. In addition, it appends a set of class- and method-level *attributes* into team and role classes (class-level attributes), as well as their methods (method-level attributes). For example, it adds the `BaseClassTags` attribute to team class to *give* it an ID in each base class bounded to one of its roles. Another example is the `CallinFlags` attribute, which is attached to every role method bounded with a specific CIMB expression.

These attributes (and others) are used mainly at the load-time during the classes' bytecode transformation in order to *resolve* CIMBs precedence, designate team implicit activation and deactivation statements, etc. The transformers of OT/J read *successively* these attributes and store them in the Callin Binding Manager (CBM); a central component that is used by OT/J transformers as a repository of information. The CBM is referenced whenever these transformers need information about specific role, base, or team classes, and the CIMB expressions declared in roles.

Role Weaving is hard coded in Team and Base Classes

To illustrate the obstacles resulting from tight-coupling roles and base objects, let us refer to the example in Figure 1.4. Listing 1.2 below shows a snippet of the actual compiled code of **Company** team class[1]. Lines 1 and 6 in Listing 1.2 show the compiled declarations of `hired` and `hire` methods, respectively. Recall that these methods declare *explicit lifting parameters* [75, 79] to the base class **Person** (see class diagram of the **Company** team in Figure 1.4). Note at lines 3 and 8 that the base instance provided to the method is first *lifted* [2] into a role instance by invoking the special-purpose method `_OT$liftTo$Employee(..)` in the team instance (This method is part of the Translation Polymorphism [77]). If the provided base object is *not* of a proper type, then a lifting *exception* will be raised.

Listing 1.2: A compiled code snippet of the **Company** team class.

```
1. public boolean hired(Person _OT$p)
2. {
3.   Employee p = _OT$liftTo$Employee(_OT$p);
4.   return employees.contains((Object)p);
5. }

6.    public boolean hire(Person _OT$p, float salary)
7.    {
8.        Employee p = _OT$liftTo$Employee(_OT$p);
9.        p.setEmpNo("E"+p.hashCode());
10.       p.setSalary(salary);
11.       return employees.add((Object)p);
12.   }

13.   protected Employee _OT$liftTo$Employee(Person base)
14.    {
15.        Employee myRole;
           :
16.        if(!_OT$cache_OT$Employee.containsKey(base))
17.        {
18.          switch(base.getde$tub$cs$abdullah$teams$Company_OT$Tag())
19.          {
               :
20.}}}
```

[1] We use (DJ Java Decompiler v.3.10.10.93 © 2000-2007 - Atanas Neshkov) to decompile class files.

[2] Lifting mechanism will be discussed in details in Chapter 2, Section 2.3.1.

Inside the lifting process, the team instance checks if the base instance under lifting is already existed in its *caches*. If that base object does not exist, then a new role instance will be created, and then bound to that base object. Before creating a new role instance, the team instance *verifies* that the base object under lifting has the same *tag* (Identifier) value that was given to the **Company** team class in the **Person** class. The team instance performs this verification by invoking the method `getde$tub$cs$abdullah$teams$Company_OT$Tag()` in the base object (see line 18). In fact, the invocation statement at line 18 is *injected* into the code of the lifting method during the team class compilation and not by the OT/J transformers. Afterward, the OT/J *Base Transformer* injects this method into the base class **Person** at the load-time to return the *previously known* team's tag value (of the **Company** team) that has been stored in the CBM.

When the bytecode of **Person** class is loaded into the JVM, the "aspects weaving" process is taking place. Specifically, the OT/J Base-transformer communicates with the CBM to determine which of base class methods have been bounded with CIMBs. Once determined, the transformer injects the suitable *traps* in the methods' code according to the modifier types of these CIMBs and the roles' precedence. In addition, it weaves a set of model-specific methods that facilitate role playing like registering the activated team instances and unregistering the deactivated team instances (will be discussed in the following paragraphs).

Listing 1.3 illustrates part of **Person** class code after transformation. Note the new method `getde$tub$cs$abdullah$teams$Company_OT$Tag()` at lines 18 - 21. Note also the *reformulation* of method `getInfo()` (lines 1-7) since it has been bound with

Listing 1.3: Part of **Person** class implementation after transformation.

```
1.  public String getInfo()
2.  {
       :// prepare the list of registered activated teams
3.     for(...){
4.            if(!_OT$activeTeams[i].isActive()){...}
5.     }
6.      return (String)_OT$getInfo$chain(...);
7.  }
8.  public Object _OT$getInfo$chain(...)
9.  {
10.     Object _OT$result = null;
11.     if(_OT$idx >= _OT$teams.length)
12.        return (Object)_OT$getInfo$orig();
       :
13.      switch(_OT$teamIDs[_OT$idx])
14.      {
15.       case 1: // '\001'
16.       _OT$result =
            (Object)((Company)_OT$team)._OT$Employee$getEmpInfo$getInfo(...);
          :
17. }
18. public short getde$tub$cs$abdullah$teams$Company_OT$Tag()
19. {
20.   return 1;
21. }
22. public static synchronized void _OT$addTeam(Team team, int teamID)
    { … }
23. public static synchronized void _OT$removeTeam(Team team)
    {…}
```

replacement CIMB (as shown in Figure 1.4). Line 4 shows that a **Person** object can *directly* access team's the method `isActive()` to check the *activation status* of team instance. Moreover, at line 6 the control flow is directed to invoke the new injected method (`Object _OT$getInfo$chain(…)`), which is the *trap-method* that will handle the *chaining* interceptions of `getInfo()`. This method will be invoked *recursively* to handle the CIMB interceptions of roles of the currently *activated* team instances according to activation order and role playing precedence. The actual interception of method `getInfo()`is carried out when the base object invokes directly the *Team-level Wrapping Method* `_OT$Employee$getEmpInfo$getInfo(…)` of team class **Company** as shown at line 16, which *wraps* the invocation of role-level method `getEmpInfo()`.

In addition, the Base Transformer of OT/J injects into the bytecode of **Person** class the *static* methods `_OT$addTeam(...)` and `_OT$removeTeam(...)` that will be invoked at runtime by team instances to register themselves in **Person** base class and unregister dynamically when they are, respectively, activated and deactivated. Listing 1.3 shows the declarations of the two methods at lines 22 and 23, respectively. The invocation statements to these methods will be injected into the bytecode of **Company** team class at the load-time by the OT/J Team Transformer.

1.4.2 Problem statement

From the previous elaborated analysis, the strong coupling between base classes and their roles (from one side), and base and team classes (from the other side) could be easily deduced, which practically *prevents the dynamic weaving of new aspects (roles) at runtime* [1, 2]. The following paragraphs will describe this tight-coupling in clear points, and illustrate how it directly impacts (prevents) mapping the features of the OT model to distributed environments.

First – The "Aspects weaving" technique used in OT/J results in hard coding roles in the bytecode of base classes. Specifically, the CIMB interception traps are woven into the bytecode of base classes (see for example Listing 1.3 lines 15 and 16); thus, transformation of base classes is *dependent on* transforming team and role classes. This includes injecting team-specific methods into the bytecode of base classes, which impose a *prior knowledge* of the accurate precedence of role playing. Therefore, to ensure an accurate role playing, base classes *must not* be transformed *before* Teams' transformer *resolves* roles' precedence, and stores attributes of teams and CIMBs expressions in the CBM.

Practically, to map (or employ) the *playedBy* relationship in distributed applications while – at the same time – preserving the above properties, team and role classes need to be compiled first, and then a copy of the team and *all* of its role classes must be deployed (manually) on every application node; in order to enable the OT/J Base transformer to transform the bytecode of base classes accurately. In this case, several identical copies of team instance *must* be created at each node to handle role playing on that node. Practically, this solution has several problems:

(1) The existence of multiple identical team instances (hereafter *sub-team instances*) imposes a mechanism that can preserve *state consistency* of sub-team instances, which is a primary requirement to unify the *activation status* of the entire team.
(2) A modification to the team class description imposes re-compiling team class and all its role classes; hence, application execution will be interrupted. Moreover, it will

lead to *Class Versioning Problem* when team instances of old team class are persisted and then reclaimed according to the new team class.

(3) The inter-relationship between role classes will be broken. For example, if two role classes played by two separated base classes establish an association relationship (as the one shown in Figure 1.8 (a) between **RoleA** and **RoleB**), then the deployment of sub-team instances as shown in part (b) will bind role instances of type **RoleA** with base objects of class type **BaseA** that have been deployed on host **H1**. Likewise, role instances of **RoleB** role will be bound to objects of **BaseB** class at **H2**. The deployed sub-team instances at hosts **H1** and **H2** cannot execute the method `comp(…)` because the set of role instances of each sub-team resides in different locations.

(4) If a team class defines several bounded roles, then deploying team instance at each of application nodes compulsory result in generating *residual* role and base classes at these nodes; since OTRE requires the existence of base classes at load-time to complete the role weaving process. For example, in Figure 1.8 (b) the role class **RoleB** and base class **BaseB** (with strikethrough names) are residual classes generated at **H1**.

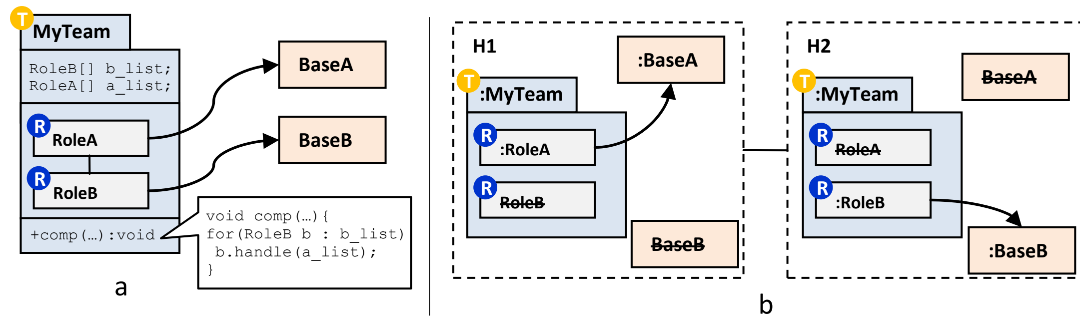

Figure 1.8. Mapping the OT/J application in (a) to the distributed environment in (b).

Second - In the current OT/J implementation, base objects of an application can play only the defined roles, i.e. they will not be able to play further new roles *dynamically* after their classes are loaded into the JVM. Therefore, employing new teams in order to append new roles requires re-compiling and then re-loading the whole application (in order to enable OT/J Base transformer to weave the new roles). Thus, in order to support the dynamic adaptation of application objects via role playing, OT/J must enable the *dynamic appending* of new teams that comprise new desired functionalities within roles' classes [3].

Third – if the *playedBy* relationship is employed in distributed applications at the abstraction level (i.e. at the source code), then the problem of *playing roles by interfaces* will be faced [79 and 1]. More precisely, Java-based distributed applications are implemented mainly by using so-called *Contract-based design* [12], which stipulates that distributed objects must implement one or more interface that facilitate their remote access. For Java-based distributed applications, CORBA IDLs [64] and Java-RMI are the most popular approaches used to develop distributed applications. Both models rely on so-called *Remote Interfaces* to represent distributed objects in the remote environment. However, OT/J so far does not *fully* support playing roles by *base interfaces* due to compiler limitations [79]. Currently, it restricts binding a role to base interface if and only if that role declares COMBs only (This issue will be discussed in more detail in Chapter 2). One more limitation is that the OT/J transformers require the bytecode of base classes in order to weave roles; whereas interfaces cannot implement methods, and a single interface could be implemented by different classes (thus, different methods' implementation), which might result in an application inconsistency.

Fourth – the mapping of *playedBy* relationship to distributed environments imposes *relaxing* the tight coupling between base classes and their roles after they have been loaded into JVM for execution [1, 2]. Without a systematic and precise substitution to the bond between their objects, many of OT/J primary features like access of base objects directly by their roles, and the dynamic registration of team instances in base classes upon activation, become inoperative. To sum up, the *playedBy* relationship will be broken if base object and its roles are distributed over several nodes.

1.5 The Dissertation Goals

In simple words, the main goal is to develop a framework to map the features of OT model to distributed environments and realize the remote playing of roles. If this goal has been accomplished, the modularity of distributed aspects will be improved, and their employment could be controlled dynamically during application execution. Therefore, a modular and dynamic behavior adaptation of distributed application objects is achieved.

This primary goal involves the following objectives:

1. Introduce the notions of *Remote team, Remote Role,* and *Remote base* [1] as distributed versions of the corresponding notions team, role, and base in OT/J [75].
2. Enable the application objects and roles bounded by *playedBy* relationship to reside on distinct nodes/processes, preserving a high level of reliability and transparency regarding the process of role-playing.
3. Enable the dynamic functional adaptation (behavior adaptability) of distributed application objects following a modular and systematic way without compromising their internal structures or interrupt application availability.
4. Enable the dynamic attaching/detaching of teams at runtime in a consistent fashion, and allow the team activation/deactivation processes dynamically without breaking the original functionality of application base objects. To allow the control over aspects' effects at runtime, guard predicates and base guard predicates should be employed, which enables developers to manage aspects dynamically like stop and resume their effects.
5. Ensure that the above concepts for dynamic RRP effectively support the collaboration-based programming technique in a distributed context.
6. Achieve a secure RRP and safe dynamic adaptability that prevent exposing application components to intrusion risks and undesirable access.

1.6 Solution Overview [1, 2 and 3]

Extending local-AOP approaches into distributed computing, in order to exploiting their capabilities, is not a new idea. Several distributed AOP models have extended local-AOP models by adding *remote pointcut designators* [55, 47] and supporting *remote advice execution* [47, 11, 70, and 55]. For example, DJcutter [55] extends AspectJ [27] by remote pointcuts and distributed inter-type declarations. AWED [47], as another example, extends

JAsCo [17] by adding explicit distributed pointcuts, and supporting the dynamic weaving of aspects.

Mapping the features of OT model to distributed environments, on the other hand, is not trivial; since it involves extending "team", "role", and the "playedBy" concepts into these environments. The fact that distributed application objects reside at different network nodes (besides the aforementioned obstacles), calls for a precise and well-designed mechanism in order to combine both features altogether. This section mainly studies the relationship *playedBy* to identify those entities that participate in the role-playing process. Afterward, it presents an approach to map these entities to a distributed environment, and introduces the adequate protocol required to facilitate the communication between them in order to accomplish the RRP.

Basically, the *playedBy* relationship spins the following entities:

1. A team class (T).
2. A base class (B).
3. A bounded role class (R).

Formally, this relationship could be expressed as follows:

$$playedBy\,(T,R,B) = T(R{:}B); \quad \text{[read as: team } \mathbf{T} \text{ binds role } \mathbf{R} \text{ to base class } \mathbf{B}\text{]} \tag{1.1}$$

In contrast, remote role-playing involves that base class, and the team class encloses its role, are separated. The base class will be referred to as $\mathbf{B_{H1}}$ to indicate that objects generated from this class resides at host H1; precisely, the objects generated from this class *live* in the JVM executing on host H1. Hereafter, $\mathbf{B_{H1}}$ will be called as *remote base class*. Likewise, the team class will be referred to as $\mathbf{T_{H2}}$ and called as *remote team class*. Thus, remote role-playing is expressed as a relationship between $\mathbf{T_{H2}}$ and $\mathbf{B_{H1}}$ as follows:

$$remote\_playedBy\,(T_{H2},R,B_{H1}) = T_{H2}\,(R{:}B_{H1}); \tag{1.2}$$

In both cases (local and remote role-playing), base objects must be bounded to role instances within team instance at runtime. The rest then is the remote communication between these entities to realize role-playing activities like intercepting base objects' functionality by CIMBs and execute the corresponding role's callin methods, evaluate the GPs, etc.

Broadly-speaking, the followings are the guidelines to achieve the DADAO using the RRP, which satisfy the conditions of Definition 1.1:

1. The *playedBy* relationship is a *replaceable* relationship [3]. That is, the code segments generated for realizing local *playedBy* relationships could be replaced by code segments of *remote playedBy* without violating the integrity of structure of base or team (including role) classes.
2. The communication between remote base and team objects cannot be established without a *middleware layer* that facilitates their allocation and accurate binding, and ensures a precise dynamic adaptation for objects' behaviors [3].

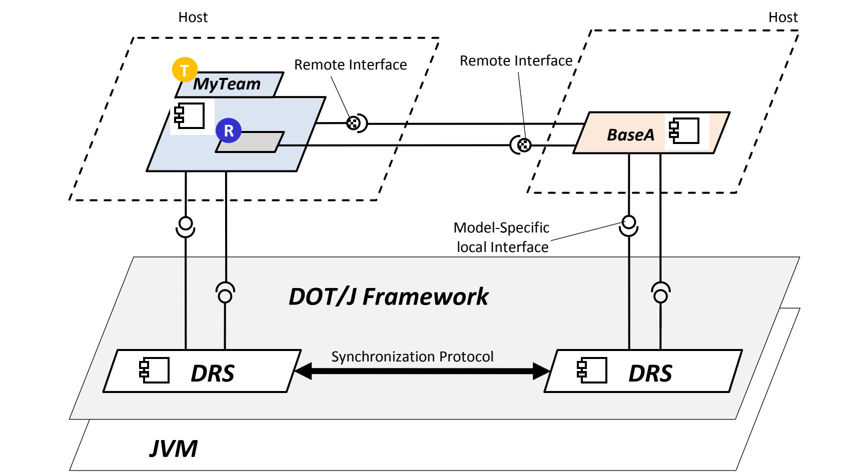

Figure 1.9. Mapping OT/J application to the DOT/J Framework.

Following these guidelines, the DOT/J framework will be introduced. The DOT/J framework consists of two primary units: the DOT/J Transformation Library (DTL) and the DOT/J Runtime System (DRS) [3]. The DTL *reformulates* remote base and remote team classes as *components* [3]. Each component provides an interface and requires another one in a relation aims at establishing the required communication between their instances at runtime. To explain and design of this relation, Figure 1.9 illustrates the base class **BaseA** (from Figure 1.8) as a component that provides a *remote interface*, which is required by the remote team **MyTeam**. Likewise, the remote team **MyTeam** provides a *remote interface*, which is required by the remote base class **BaseA**.

The provided-required interface relationship must *substitute* the local *playedBy* relationship. Thus, it must be implemented in a way remote CIMBs and remote COMBs are handled accurately at runtime; i.e. as a fulfillment of the RRP requirements. The remote interfaces had been chosen to be *as generic as* possible; in case of remote base objects, the provided interface is considered generic if it enables these objects to play any roles that might be declared dynamically at runtime. In fact, this specification is important to accomplish a consistent dynamic role weaving. For remote teams, they should provide a generic remote interface that both enables remote base objects of classes specified in *playedBy* relationships to play their roles, and enables application developers to access remote teams as distributed components. The provided-required interface design provides the necessary separation which allows putting roles for a dynamic and remote playing without further modifying remote base classes, which supports the continuous adaptation of distributed components.

The DRS, on the other hand, forms the necessary *middleware layer* on which remote base and remote team components communicate [3]. The DRS should perform two main tasks:

1. Supports and organizes the coordination between remote base objects and remote team instances *before* the actual RRP activities take place. This involves a set of subtasks like remote team instances allocation and lookup, remote base objects notification in case of new remote team instance attachments, etc.
2. Provides synchronization between the underlying components of the DOT/J framework to ensure the consistency of the RRP at every application node. In practice, executing the DRS at every node of a distributed application is needed. Thus, a reliable inter-communication channel is required between the different DRSs. This inter-

communication is very important to ensure that all remote base objects of a distributed application can play the same group of remote roles at any point of time.

The remote base and team components could be connected to the DRS via a set of model-specific interfaces as Figure 1.9 illustrates. These interfaces should provide a communication between remote team instances and the DRS in order to simplify the allocation of these team instances by the remote base objects of distributed application *at* all nodes. Likewise, remote base objects should be able to communicate with the DRS in order to receive the notifications of new remote team instances' attachment to the application *at precise time of execution*. Consequently, these base objects can play their roles accurately.

For simplicity, the design of DOT/J framework is divided into three contribution levels, namely: (1) the static level, (2) the class-loading level, and (3) the runtime level. At each level, the requirements of mapping the OT model to support remote role playing at that level will be studied. Then a discussion will be made to the various alternatives for achieving these requirements.

1.6.1 The Static Phase

The static phase points out to the source code of OT/J applications before compilation and class-loading processes are taking place. At this phase, no new keywords or compiler directives have been added to the OT/J language. Rather, DOT/J framework reuses the OT/J's compiler to compile team and role classes that are to be mapped to a distributed environment. In this way, the consistency of local *playedBy* relationships and base and role type-checking is ensured. All what is needed at this phase is binding the compiled classes which enrolled in RRPs to the DOT/J framework. This phase will be called *the Labeling phase* [1], which simplifies distinguishing application classes at load-time in order to perform the necessary *component conversion* had been mentioned earlier. In this way, the costs of developing new compiler, or extending the current OT/J compiler, are saved. At the same time, the transparency degree of RRP is increased.

1.6.2 The Load-time Level

The application base, role and team classes (which involved in a RRP) need to be made as "*RRP-ready*" components transparently. But the roles of remote teams cannot be played by the remote interfaces provided by remote base classes. Thus, these roles must be *compelled* to accept binding with the remote base interfaces *before* they are loaded into the JVMs. Thus, the bytecode of the application classes labeled as remote classes must be transformed at load-time. This process aims at implementing remote-, and other, framework-specific interfaces provided by remote base classes. These interfaces enable remote base objects to:

a. Transparently allocate the interested remote team instances, and participate in these teams by playing the enclosed roles.
b. Accurately respond to remote CIMBs that intercept any of their methods, and facilitate remote COMB requests.

Likewise, remote team classes must be transformed accordingly so that their instances facilitate a precise binding between remote base objects and role instances at runtime, and enable the dynamic remote team (de)activation processes. In addition, remote role classes need to be transformed in order to enable their instances to bind with remote base objects rather than local base objects. In fact, this transformation wanted to be carried out transparently. To achieve this, two extra bytecode transformers have been appended to the Object Teams Runtime Environment's (OTRE's) Transformers-stack: the first one transforms remote team classes, and the other will transform remote base classes. Figure 1.10 illustrates the incorporation between DOT/J's transformers and OT/J's transformation system. As the figure shows, the compiled bytecode of application classes is manipulated first by the OT/J transformers, and then DOT/J transformers perform further modifications. These modifications aim at conforming the transformed classes to the RRP disciplines.

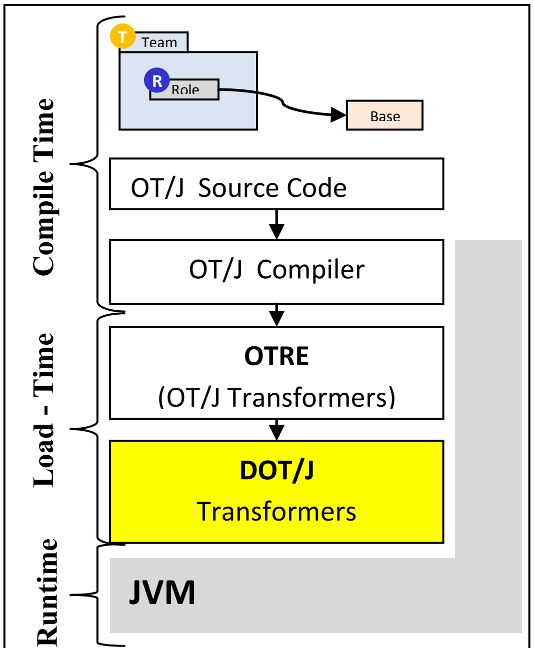

Figure 1.10. Incorporating DOT/J transformers and OT/J transformers.

Listing 1.4 illustrates a pseudo code for the algorithm used by the DOT/J Base Transformer to transform remote base classes, and in Listing 1.5 the algorithm of the Remote Team Transformer. In Chapter 3, the detailed specifications of DOT/J transformers and the main transformation operations they perform on classes' bytecode will be discussed.

1.6.3 The Runtime Level

Besides the main tasks performed by the DRS, the DOT/J framework must take care of important issues at runtime like the dynamic attachment/detachment of remote teams, the *activation order* of remote team instances, and the accurate order of CIMBs employment, etc.

By accomplishing the DOT/J framework, it will be possible to map the OT/J application shown in Figure 1.8 (a) to the distributed environment shown in part (b) of the figure using the RRP relationship; as shown in Figure 1.11. Now, a single team instance is needed; thus, the RRP activities could be controlled via activating/deactivating that team instance rather than (de)activating multiple team-copies. In addition, the inter-relationship between team's roles are preserved unbroken, and preserve the same modularity offered by the OT model.

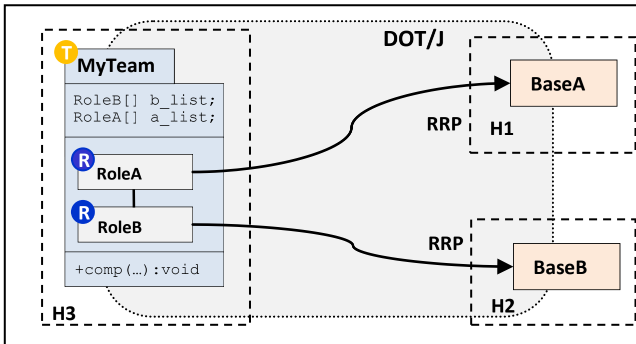

Figure 1.11. Mapping the OT/J application in Figure 1.8 to a distributed environment with the RRP relationships.

Listing 1.4: Remote-Base Transformer pseudo-code

```
If base class is labeled as remote base then
   Add the suitable remote interface
   Enable instances of this class exporting themselves
          as remote base objects
   Add the required methods that:
       Facilitate designating remote teams
       Handle remote callin method bindings precisely
       Handle field set/get operations
```

Listing 1.5: Remote-Team Transformer pseudo-code

```
If team classes is labeled as remote team then
     Add the suitable remote interface
     Enable instances of this team exporting themselves as remote team instances
     Add the required methods that:
          Support detecting the CIMBs by different remote base objects
          Facilitate role-level callin method to be remotely invoked
          Facilitate Lifting/Lowering of remote bases/roles instances
          Facilitate the access of team instance as distributed component
     For each role to be played by a remote base do
          Reformulate its and its role-implementer classes appropriately to conform to the
             new remote playedBy relationship.
```

1.6.4 How can RRP support the DADAO?

The RRP can support the DADAO as follows:

1. Roles can freely come and go without affecting the structure of remote base objects, and without stopping application execution or interrupting its services.
2. The roles of a specific remote team instance are played *inside* that team; thus, no need to weave and unweave roles (aspects) dynamically by injecting new code in base objects; which obviates the problems of *atomicity* [51, 20] and application architecture consistency. Rather, to employ roles (aspects) in applications, base objects need only to play them remotely. The playing of roles occurs simultaneously at all nodes.
3. All the conditions mentioned in Definition 1.1 are satisfied when distributed base and team classes of application are properly mapped to the DOT/J framework.

1.7 Structure of the Dissertation

This dissertation is organized as follows:
Chapter 2: *Object Teams / Java – The concepts*. This chapter will present the OT model, and discusses through examples its fundamentals and programming features, as well as the process of application development in OT/J.
Chapter 3: *Remote Role-Playing*. This chapter defines the Remote Role-playing (RRP), and illustrates the possible scenarios of playing roles remotely. Afterward, it discusses the requirements of the RRP.
Chapter 4: *The Concepts of Distributed Object Teams.* This chapter proposes the Distributed Object Teams/Java framework as an implementation of the Remote Role-Playing

(RRP) technique for Java. It describes the concepts for mapping the fundamentals of OT model to distributed environments.

Chapter 5: *DOT/J Implementation*. This chapter discusses in details the implementation of the DOT/J framework. It divides the implementation process into three phases according to the discussion made in Chapter 3; namely, static or source code phase, load-time phase, and runtime phase. Then, it studies in details the specifications of implementation of each one of these phases.

Chapter 6: *The DOT/J Runtime System*. This chapter studies the runtime system of DOT/J where remote objects *must* precisely communicate with each other to achieve an accurate remote role-playing. The chapter traces, through examples, the RRP from the point of remote objects' creation until the complete remote playing of roles. In addition, this chapter further describes the DOT/J infrastructure components that are mentioned in Chapters 3, 4 and 5, and illustrates the guidelines for developing dynamically adaptable distributed applications via Remote Role-playing.

Chapter 7: *DOT/J Case Studies and Evaluation*. The DOT/J approach will be evaluated in this chapter through developing several case studies. First, the chapter emphasizes the capability of the DOT/J framework to map OT/J applications accurately in order to achieve the DADAO, and then it discusses performance issues. The chapter compares DOT/J and AWED against expressivity, performance, and other issues. Furthermore, it shows other capabilities for the DOT/J framework like supporting collaboration transparency between legacy applications, supporting expressive distributed aspects, and developing rich distributed components via augmenting remote team instances with the RRP. The Chapter compares DOT/J to DandyJ; an AOP approach for dynamic adaptation of distributed application through dynamic aspects weaving.

Chapter 8: *Related Works*. The chapter presents more detailed discussion for the related approaches to the DOT/J approach in distributed AOP, distributed applications adaptation, and other software engineering fields.

Chapter 9: *Conclusions and Future work*. This chapter discusses the achievements of this dissertation, and verifies that all the goals mentioned in this chapter are accomplished. Then, it discusses the obstacles that face the implementation of the DOT/J framework, and shows how could they be solved or diminished. After that, it documents some of DOT/J features that might require more enhancements besides other future works, and records the approach's current limitations.

Chapter 2

Object Teams/Java – The Concepts

This chapter presents the basis of the OT/J programming language (the language that implements the OT model), and illustrates by a set of examples the programming features it provides. The chapter also presents the behavioral adaptation of application objects from a perspective of the OT/J, and explains how developers can use OT/J to implement adaptable applications in a high degree of expressivity and modularity.

2.1 Introduction

OT/J is a programming language that extends the Java™ programming language with the new programming modules and concepts of the OT model. The core of OT/J is divided into two main sub-systems; the Object Teams Development Tooling and the Object Teams Runtime Environment. Currently, the implementation of OT/J has been developed on top of the Eclipse™ Java Development Tooling (JDT) [90].

On the one hand, the Object Teams Development Tooling (OTDT) is the part of OT/J that extends Eclipse JDT by a set of plug-ins (i.e. OSGi bundles), which involves mainly the OT/J compilation unit (which translates the source code of OT/J applications into Java classes), and the OT/J's user-interface perspective (e.g. Code editors, Views, etc.). On the other hand, the Object Teams Runtime Environment (OTRE) is the part that is dedicated to transform application classes by *weaving* – into team and base classes – the necessary code segments that realize the language-specific concepts especially the *playedBy* relationship (e.g. the implementation of callin- and callout-method binding [79]). This section will highlight the main features of OT/J by implementing the simple example mentioned in Chapter 1 – Section 1.3.

2.2 OT/J Entities: their Modularization and Ontology

This section defines the basic concepts and construction blocks of OT/J applications.

2.2.1 Teams

In OT/J, the term "team" is used to denote any first-class module that is defined using the keyword `team`. Programmatically, a team class is a normal java class that makes use of all the features of Java classes, i.e. it can define methods and attributes, declares class constructor(s), inherits from another team class, and can implement interfaces. For example, Listing 2.1 illustrates the declaration of the **Company** team (of Figure 1.4). Note the keyword `team` in the declaration statement in line 1.

Listing 2.1: The Implementation of **Company** team

```
1.  public team class Company
2.  {
3.   private String companyName; // attribute
4.   private String companyAddress;
5.   private ArrayList<Employee> employees;
6.   // and other attributes...
7.   public Company(String _companyName, String _address) // constructor ..
8.   {
9.      this.companyName = _companyName;
10.     this.companyAddress = _address;
11.     this.employees = new ArrayList<Employee> ();
12. }
13. public void hire(Person as Employee p) {..} // method
    :
14.}
```

A team class can extend only team class, and a child team inherits from its parent team *all* the declared members. In fact, declared classes with the keyword `team` implicitly extend the pre-generated team class `org.objectteams.Team` (the parent of all teams). In this regard, OT/J coins the term "Implicit Inheritance" [79] (§1.3) to point out that a child team inherits also role classes and the *playedBy* relationships of its super team. Thus, child teams can perform further specialization on these relationships and role classes.

2.2.2 Roles

In OT/J, there is no special keyword like `team` that developers use to define roles. Basically, any class or interface declared inside a team class is a role. Within a team class, roles are classified into *bounded* and *unbounded* [75, 79]. Bounded roles are those classes that are associated with specific base classes via the *playedBy* relationship. Listing 2.2 shows a code snippet of the implementation of **Employee** role. It illustrates how this role has been bounded to the **Person** class through `playedBy` relationship. Unbounded roles, on the other hand, are the inner classes and interfaces declared inside a team class without linking them to base classes with the *playedBy* relationship neither implicitly nor explicitly. In general, bounded roles can extend unbounded role classes, and can implement unbounded role interfaces; but not vice versa. According to the "implicit inheritance" rule, the inherited bounded roles could be specialized further (e.g. their bounded and unbounded methods could be overridden) while the inheriting role can make use of all the declared Callin Method Bindings (CIMBs) and Callout Method Bindings (COMBs) of its parent role.

In terms of role class modifiers, roles are declared with the `public` or `protected` class modifiers in conjunction with `abstract` or `final` modifiers (but not the `static`). With the `protected` modifier, roles are accessible only inside the enclosing team or the nested sub-teams (called *role-teams*). In Listing 2.2, the role **Employee** is declared with modifier `protected`. Therefore, the access to its instances directly from outside an instance of **Company** team is prohibited. In contrast to the "protected" roles, instances of role classes declared with the `public` modifier could be accessed from outside their enclosing team

Listing 2.2: Implementing the **Employee** role in the team **Company** (Lines 7 - 23)

```
1.     public team class Company
2.     {
3.      private String companyName;
4.      private String companyAddress;
5.      private ArrayList<Employee> employees;
6.      // and other attributes...
7.      protected class Employee playedBy Person
8.        base when (hired(base))
9.      {
10.       private String employeeNumber;
11.       private float  emplyeeSalary;

12.       public void setEmployeeNumber(String _empNo)
13.       {this.employeeNumber = _empNo;}
14.       public void setEmployeeSalary(float _salary)
15.       {this.employeeSalary = _salary;}
          // Callin methods ..
16.       callin String getInfo()
17.        {
18.         return "Employee Name: " + getEmployeeName() + " Emp No:" +
            getEmployeeNo() + " Salary:" + getEmployeeSalary() + " Eur.";
19.        }
          // Callout methods ..
20.       abstract String getEmployeeName();
          // callin method bindings ..
21.       getInfo <- replace getInfo;
          // callout method bindings ..
22.       getEmployeeName -> getName;
23.     }
24.     public boolean hired(Person as Employee _p)
25.     {
26.         return this.employees.contains(_p);
27.     }
28.     public void hire(Person as Employee _p, float _salary)
29.     {
30.         _p.setEmployeeNumber("E"+_p.hashCode());
31.         _p.setEmployeeSalary(_salary);
32.         this.employees.add(_p);
33.     }
34.     // other team staff.
35.    }
```

instances (e.g. client programs can invoke any of role's public methods). OT/J achieves this specification through a mechanism called *role externalization* [79](§1.2.2). This mechanism exports the confined instances of public role outside the boundaries of the confining team instance so that they could be accessed as normal objects.

Role classes in OT/J can define their own members (i.e. fields and methods), in which the private members of a specific role class *are not* visible to the enclosing team – on contrary to the normal inner classing in Java; the private members of inner classes are accessible to the outer classes. For example, in Listing 2.2, the **Employee** role declares two private fields: (**`private`** `String employeebyer` and **`private float`** `employeeSalary`), which are accessible *only* inside role instances. Conversely, all team members are accessible to all the roles it has enclosed, in spite of their visibility.

2.2.3 Application Base Classes – The Players of Roles

Classes mentioned after the *playedBy* relationships are called *base* classes. A base class could be any of application classes that are visible in the scope of team class; i.e. within the class-path of the team under implementation. In more advanced binding situations, OT/J presents a set of rules [79] which govern the binding of base classes with roles. For example, when a base class is itself a role, which has been declared in different team class, or when the bounded role has extended a role that is bounded to a specific base class.

In this chapter, however, the simple binding of role and base classes will be discussed. Actually, OT/J offers more advanced composition of role and team classes, besides advanced role playing strategies. For instance, *Nesting*, *Stacking*, and *Layering* are all advanced composition scenarios, where the base class might be a team or a role class; (this issue will be discussed in brief at the end of this chapter). Here, it is important to mention that OT/J doesn't require the source code of application base classes in order to compile the classes of team and role and the *playedBy* relationships. This feature enables application developers to apply the OT concepts on legacy Java applications; for example, to adapt the functionality of their objects and to allow them to evolve.

2.2.4 The *playedBy* Relationship

The relationship *playedBy* is a bidirectional role-to-base link that binds – at the source code – a specific base class to a specific confined role class of a team class. At this level, *playedBy* is the only clause that can give information about the relation between the role class and its base class [73]. In other words, *playedBy* explains expressively why a base class connects to a role class. At runtime, this relationship binds exactly one role instance to one base object inside the team instance. This means that a specific base object could be bound to several role instances in different or the same teams as it can play several roles at the same time. This binding opens a communication channel through which the control flow could be transmitted between the two instances (role and base) in order to perform the CIMB or the COMB bindings, or to evaluate the Base Guard Predicate (BGP) expressions. However, OT/J uses the keyword `playedBy` to bind a role class (at the left-hand side) with a base class (the right-hand side) as shown in Listing 2.2 at line 7, which binds the **Employee** role to the **Person** base class.

In respect of a base class point-of-view, binding role to base class is a kind of specialization. More specifically, a role is considered as the behavior of a specific base object inside team (i.e. the "personality" of that base object as being a participant in that team (collaboration)). In respect of the role perspective, the *playedBy* relationship is a type of inheritance; because role instances can invoke the methods of their bounded bases (via COMBs), access their fields (via fields *getters* and *setters*), and can *override* any of the base's methods (via CIMBs), which exhibit the same effects of class inheritance in Java.

2.2.4.1 Base Binding Considerations

The *base* player is the class name mentioned at the right-hand-side of *playedBy* relationship; as shown in Listing 2.2 at line 7. The validity of a base class for being a role player in OT/J is dominated by a set of *rigorous* rules especially in cases of class polymorphism and base interfaces binding. The next paragraphs will discuss more of base binding considerations.

In OT/J, role and base classes are bounded according to a one-to-one relationship. If a specific base class is bounded to more than one role in the same team class (including the implicitly inherited roles), then application developers need to resolve the conflicts of replacement CIMB. That is, when two roles are declaring two CIMBs that replace the same base method. In this case, OT/J compels application developers to declare a *role precedence* clause to determine which role has the highest priority when a CIMB is matched at their base object. Furthermore, OT/J prevents the binding of roles to role-bases of the same team or any inherited role. Also, any bounded base class must be *visible* to the team class which encloses the role. The visibility here means two things: first, the base class is not restricted by modifiers like `private` and not a system class like `java.lang.Object`. Second, it could be reached from the application class-path so that OT/J can *import* it as a base class into the team class. This is important because compiling of team classes requires the coexistence of base classes to verify their validity as eligible players and to verify their methods and fields which are linked in CIMBs, COMBs, and BGPs.

Moreover, before it has been migrated to the Eclipse™ family, OT/J restricts the binding of role classes to base interfaces due to compilation limitations [79]. On the one hand, interfaces could be implemented by different objects types. Thus, it will be a meaningless to play the same role by different types of player objects. That is, the high expressivity of the language might be dispersed. On the other hand, binding base interfaces can shred application functionality; because bind a specific role-level method to a base-level method that might be implemented differently by different base objects can cause serious deformations to the application. For example, it is impossible to declare "`after`" or "`before`" callin methods that issue a field getter to get the value of a specific base field if the bounded base object does not declare or inherit that field. In addition, it is illogical and error-prone to bind a base method, which could be implemented differently by base objects, to specific role-level *replacement* callin method. Anyway, the OTDT (from version 0.7.1 under the umbrella of Eclipse™) allows binding base interfaces to role classes if – and only if – these roles declare COMBs.

2.2.4.2 Base Guard Predicates (BGPs)

In OT/J, guard predicates are conditional expressions that comprise logical operations (<, >, ==, !=, >=, and <=), invocations of boolean methods (that return `true` or `false`), constants and local variables, and team fields. Logical operations could be further connected with the conditional conjunctions (`&&`, `||`, and `!`). Guard predicates are used in OT/J to control the effects of callins ([79](§5.4)), and they have the following general format:

[**base**] **when** (*conditional_expression*) (2.1)

Base Guard Predicates (BGPs for short) are those conditional expressions that contain one or more reference to base fields or invocations of its methods. In this regard, OT/J uses the keyword **base** (see the format above) to distinguish between normal GPs and the BGPs. The **base** keyword is a place holder for the bounded base object. Usually, BGPs are appended to the *playedBy* relationship declarations, but could be attached to the bounded role-level methods and CIMB expressions. For example, Listing 2.2 illustrates at line 8 a

BGP that determines when to bind the base objects of type **Person** with role instances of type **Employee**. In this example, only the hired persons are employees in the company.

At runtime, any declared GP expression is evaluated before the execution of role's methods; depending on the level of GP and the result of its evaluation, the control flow either continues the execution when the GP is evaluated by "true," or simply stops/overlaps executing the targeted method or CIMB. For example, the BGP expression at Listing 2.2 line 8 will invoke method `hired(...)` of team instance and will pass the base object under binding as an argument to that method. If the invocation returns true, then all the declared CIMBs become applicable and can intercept the methods of that base object.

2.2.4.3 Callin Method Bindings

Callin Method Bindings (CIMBs) are expressions that bind role-level methods and base-level methods such that the control flow is intercepted at the bounded base method and transferred into the corresponding role-level method at a specific point of execution. In OT/J, only bounded roles can define CIMBs, and a CIMB expression is a combination of six values as follows:

- The Team class that encloses the desired role class - T
- The bounded Base class - B
- The corresponding bounded Role class - R
- The Role-level method - M_R
- The corresponding Base-level method - M_B
- The CIMB modifier type - CM

Therefore, any CIMB could be represented in a formal manner as a vector of 6-tuple each represents one of these values. For example, an arbitrary binding *b* could be represented as follows:

$$CIMB_{(b)} = \langle T_b, R_b, B_b, M_{Rb}, M_{Bb}, CM_b \rangle \tag{2.2}$$

The CIMB Modifier (CM_b) specifies *when* the control flow is intercepted at a base object and forwarded to the corresponding role instance (see Figure 2.1). In OT/J, CM can have one of the following interception values:

- `before`: which indicates that the Role-level method (M_{Rb}) should be executed before the control flow enters and executes the base method (M_{Bb}). Figure 2.1 (a) illustrates this situation.

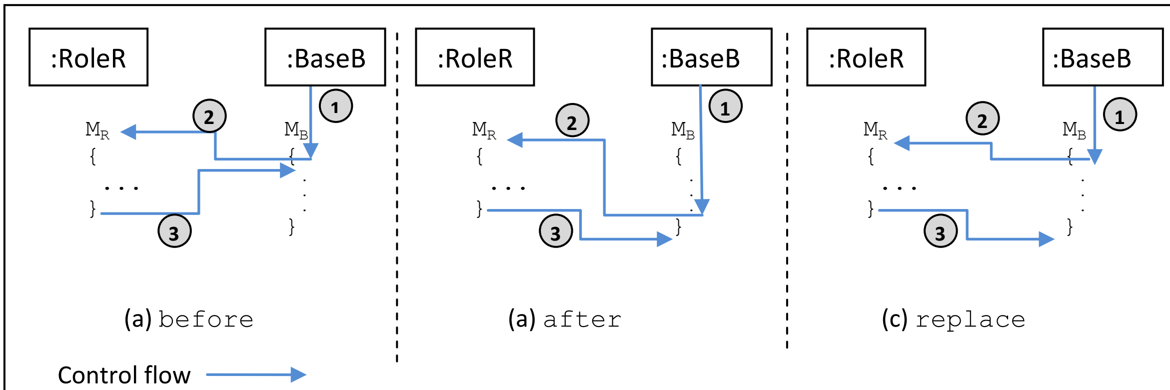

Figure 2.1. The control flow in case of: (a) `before`, (b) `after`, and (c) `replace` callin bindings.

- `after`: CIMBs with this modifier transfer the control flow into the Role-level method (M_{Rb}) after it *completes* executing the base method (M_{Bb}) and before leaving that method's boundaries, as shown in Figure 2.1 (b).
- `replace`: with this CIMB modifier, the Role-level method (M_{Rb}) is executed *instead of* the base method (M_{Bb}). CIMBs with this modifier called the *Replacement binding*s. Figure 2.1 (c) depicts the execution of replacement CIMB.

At the source code, application developers declare CIMBs according to the following format:

$$\textit{Role-level-method-designator} \;\leftarrow\; \mathbf{CM}_{\boldsymbol{b}} \;\; \textit{Base-method-designator;} \qquad (2.3)$$

The designators at both sides are place-holders for the methods' signatures. Role-level method designator can comprise, besides the name of that method, the return type and a complete parameters list. According to [79] (§4.1.c), method designator must uniquely select one method from both sides (i.e. role and base classes). Moreover, the bounded role-level methods with `after` or `before` CIMBs, could be further bounded in a replacement CIMB. Conversely, a role-level method bounded with `replace` CIMB cannot further be bounded with another `replace` CIMB in the same role class.

Listing 2.2 declares inside the **Employee** role class at line 16 the callin-method `getInfo()` which returns a string value (e.g. that might represent information of an employee). Note the keyword **`callin`** along the method declaration. OT/J uses the **`callin`** keyword to recognize those role-level methods that will be declared in the replacement callin-binding's expression. The significant point that should be mentioned here is that the replacement callin methods can't be invoked in the role instance as any other normal method (even inside the role instance itself). However, only base objects can *indirectly* invoke these methods with the intermediation of the enclosing team instance (will be discussed in more detail in Section 2.3).

2.2.4.4 Callout Method Bindings

On contrary to CIMBs, the Callout Method Bindings (COMBs) result in forwarding the control flow from a role instance to its bounded base instance. A Callout Binding binds a non-implemented role-level method (called *expected method*) with a specific base-level method. Thus, whenever a call to the expected role-method is made, the bounded method of the base object is invoked instead. The general format of COMB expression is:

$$\textit{expected-role-method-designator} \rightarrow \textit{provided-base-method-designator} \qquad (2.4)$$

The method designator could be reduced to the names of methods at both sides *if* they have two identical signatures (i.e. same method name and parameters type). In this case, a role-level declaration for the expected method is required (e.g. Listing 2.2 defines at line 20 the expected role method `getEmployeeName()` before the COMB is declared at line 22). However, developers can directly define and bind the expected role method by specifying a complete designator at both sides (i.e. return type, method names, and parameters list).

The role-level expected methods are invoked inside the role as any normal method, and called from the enclosing team instance if that method is defined as `public`. For example, at line 18 of Listing 2.2, the callout method `getEmployeeName()` is invoked from inside the callin method `getInfo`. Whenever an invocation to `getEmployeeName()` method is issued, the base method `getName()` is called instead.

Furthermore, the fields' COMBs could be used to declare expected-role methods that can access the base object's fields. In this regard, two types of callouts could be declared in role classes, namely field getter and field setter. A field-getter gets the *current* value of the field bounded in the COMB, while a field-setter set (*assigns*) a new value to the bounded field. The general formats of field-getter and field-setter declarations, respectively, are:

returnType *role-level-method-name* ( )	→ **get fieldType** *base-field-name*;	(2.5)
void *role-level-method-name* (**paraType** *paraName*)	→ **set fieldType** *base-field-name*;	(2.6)

Most likely, the *returnType*, *fieldType*, and *paraType* types are of the same type name (either primitive or class types). In this case, developers can declare abstract role methods and bound them to base's field getter or setter without declaring a specific *fieldType* name. If the type assigned to *paraType* or *returnType* is different than *fieldType*, then mechanism called *Parameter Mapping* must be used to resolve the problem of type conformation (explained in the next subsection). In concept, OT/J uses the COMBs of base fields' setter/getter to realize what so-called *Object Decapsulation* (i.e. enable the access of base object attributes in spite of their visibility), as an inverse to object's encapsulation. While it seems to be a process of object intrusion, OT/J prevents any access to the base object from outside the role instance that is played by that object if the access is not conducted by a COMB.

2.2.4.5 More Refinement on Method Bindings

The *playedBy* relationship between role and base classes is controlled (and refined further) by using the BGPs. The guard predicates, at runtime, act as locks that control the actual binding between role and base objects. However, further refinement techniques could be applied inside the bounded role class, especially at the role-level methods, to control the effects of roles callin at the level of method, and to simplify the declaration of CIMBs and COMBs. Below two strategies of OT/J that enables this *fine-grained* refinement will be discussed.

Guard Predicates inside Roles

In general, OT/J defines guard predicates at five different levels [79](§5.4). From inner- to outer-scope, these GPs are arranged as follows: GPs at CIMBs level, role methods level, role class level, base guard predicate, and team class level. This section, however, will consider the first two levels of guard predicates because they have a direct impact on the control flow channel between role and base instances at runtime:

i. Guard predicates at CIMBs:

These GPs could be attached along with the CIMB declaration expressions. The effect of these GPs impacts the callin binding itself and not the bounded method. For example, Listing 2.3 depicts a CIMB expression that leads to execute the role-level method `attachNote()` before executing the base method `getInfo()` *if and only if* the guard predicate (`notesList.size()<10000`) is evaluated to be true. Note that

Listing 2.3: An example of Guard Predicates along CIMB expressions

```
void attachNote () ← before String getInfo()
          when (notesList.size() < 10000);
```

the role-level method `attachNote()` is invoked as any normal method inside role class without evaluating this GP.

ii. Guard predicates at Role methods:
Application programmers can attach GPs to *any* of declared role-level method. The GP in this case affects the execution of the method itself; i.e. when it is evaluated as false, the method will not be executed. As an example, Listing 2.4 illustrates attaching a GP to an arbitrary role-level method. Note the keyword **`this`** which refers to the current role instance.

Listing 2.4: An example of Guard Predicates at the role-level's methods

```
public long doSomething (int x)
    when (x > 0 && this.value < 100)
{ ... // method body }
```

Parameters Mapping

In practice, a declared CIMB can bind a role-level method to abase method that can have different semantics. Programmatically, this means that the two bounded methods have different *signatures*. The same thing goes with COMB case. In order to make the role method conformed to the base method, OT/J presents the Parameters Mapping mechanism. By using this mechanism, application developers can re-allocate parameters positions or change their types in the CIMBs [79] (§4.4) and COMBs (§3.2 and §3.5.c) declarations.

Parameter mapping expressions have the general format shown in Figure 2.2 below. In case of COMBs, the → (right arrow) symbol indicates that the base method parameter will get its value from the parameter-name/expression mentioned at the left-hand-side, and the ← (left arrow) symbol is only used to map the returned value (denoted by the keyword **`result`**) of the callout according to the parameter-name/expression mentioned at the right-hand-side.

`with` {*role-side-parameter-or-expression* $\left\{\begin{matrix}\rightarrow\\ \leftarrow\end{matrix}\right\}$ *base-side-parameter-or-expression*}

Figure 2.2. The general format of Parameter Mapping Expressions.

In case of COMBs of base field getters and setters, the ← symbol maps the right-hand-side expression into the **`result`** keyword when a field getter is stated, and in case of field setter, the symbol → is used to map the left-hand-side expression into the right-hand-side expression (i.e. new field value is calculated from the left-hand-side expression).

Respecting CIMBs, the mechanism of parameter mapping is working in a similar way to the one mentioned above, except that the application developers should take care of the mapping directions. Practically, the direction of control flow in case of CIMBs moves from base objects toward their role instances. Thus, the mapping expressions always map base-side parameters to those of role-side. Additionally, CIMBs with `before` and `after` modifiers will silently neglect (at base object) any returned values from role methods; so it's meaningless to map these returned values.

2.2.5 Team Activation Mechanism

OT/J presents the mechanism of *Team Activation* [73, 75 and 79] at the team instance level to control – at runtime – the applicability of *playedBy* relationships that have been enclosed in that team instance. The effect of team activation could be explained in a simple statement as follows: "*if a team instance is activated, then all the declared **playedBy** relationships inside that team can take place, and callin methods of roles are executed once the base objects' methods are intercepted by the declared CIMBs."* As a conclusion, all CIMBs declared within role classes are inactive (or have no effects on base objects) unless the enclosing team instance is activated. This simple, but powerful, activation strategy enables developers to stop and resume aspects' effects (i.e. role playing) selectively and dynamically.

Practically, OT/J introduces two types of team activation: *Implicit Team Activation* and *Explicit Team Activation*. The first one is controlled by the runtime system of OT/J. Thus, application developers can't handle or take the control over this activation. In this type, team instances are implicitly activated whenever the control flow is crossing their boundaries or one of their roles [79] (§5.3) in. More specifically, when any of team instance methods or one of its roles method (bounded or unbounded) has been invoked, then this team instance is considered activated until the control flow leaves the called method.

The Explicit Team Activation, on the contrary, is only triggered manually by the application developers. To facilitate this, OT/J provides two methods to explicitly activate team instances: (1) the **`within`** block (see Listing 2.5), which activates the indicated team instance inside a specific range of execution, and (2) the APIs `activate()` and `deactivate()`that enable a permanent (de)activation state of team instances within the *thread* that currently serves their execution. In this direction, to (de)activate team instances for specific thread, developers can use the form `(de)activate(Thread t)`. Furthermore, developers can use `(de)activate (Team.ALL_THREADS)` to (de)activate team instances *globally* at all application threads.

Listing 2.5: The format of explicit team activation with the **within** block

```
within (teamInstanceID)
 {// team instance is activated
     :
 }// team instance is deactivated
 :
```

The order at which team instances are activated is an important issue in OT/J, especially when roles of two (or more) teams are played by the same base class. When two team instances of different team types are activated, such that both are enclosing a role class bounded to a specific base class, then the recent team to activate is the one obtaining the priority to *apply* its *playedBy* relationship first. The effect of activation order, in fact, directly impacts the application behavior, especially in case when a replacement CIMB is encountered. To illustrate this point, Figure 2.3 depicts a typical example. The figure shows three team classes each one of them confines a role class that is played by the base class **B**

(Figure 2.3 (a)). Suppose that the base class **B** declares a method named as **m**. In addition, suppose that each one of the enclosed roles declares two methods (arbitrarily named as **am** and **bm**), and defines two CIMB expressions, which bind method **m** respectively to **am** and **bm** methods as follows: **bm** is bounded to **m** with a *before* CIMB, and **am** is bounded with an *after* CIMB. The method **rm**, which is bounded via a *replace* CIMB with the base method **m**, will be added to the role **R1** of team **T1.**

To explain the effect of teams' activation order, three different team instances (say **t1**, **t2**, and **t3**) from the three team classes are created, and then activated according to the following order (**t1**, **t2**, and then **t3**). Afterward, method **m** of a base object of type **B** is invoked (as in Figure 2.3 (b)). The control flow will be transferred among the three team instances according to the order shown in part (c) of Figure 2.3.

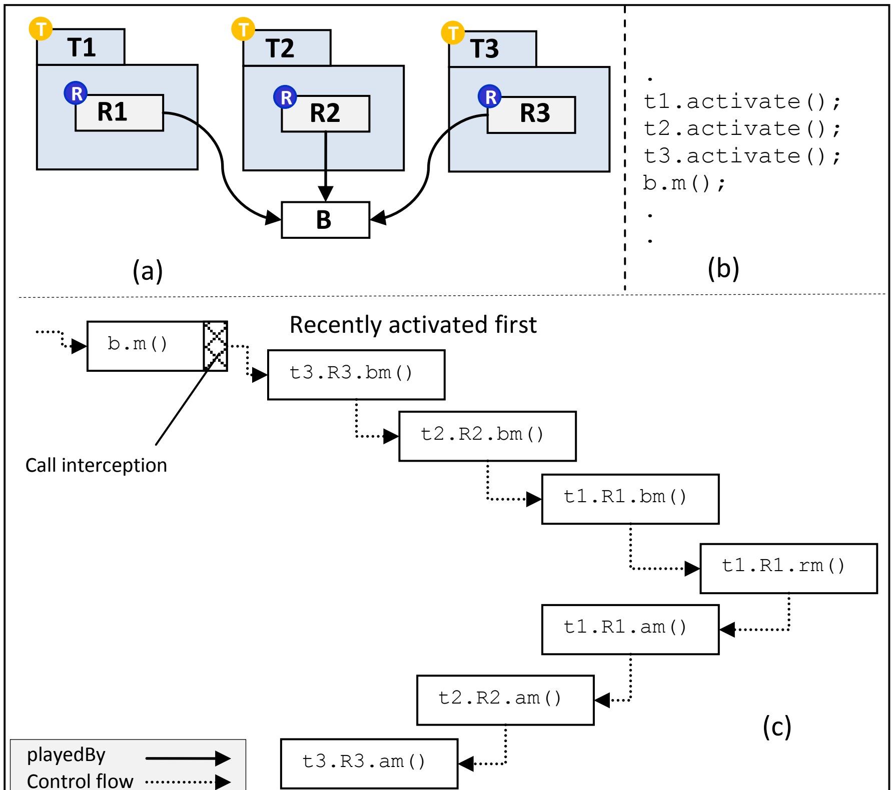

Figure 2.3. Multiple role playing design in (a), the activation order of team instances in (b), and the control flow among team instances in (c).

If another activation order plan is considered, then dispatching the map of control flow (shown in Figure 2.3(c)) will be changed accordingly. For example, if the three team instances are activated as (**t3**, **t2**, and then **t1**), then **t1** will *suppress* the effects of **t3** and **t2** roles as they have less priority, and because **R1** of **t1** declares a replacement callin. If the same activation order is used, and the replacement callin of **R1** has been removed, then all callins of roles in all the three team instances will be executed. In this case, control flow will enter **t1** first (since it is the recent to activate), then **t2**, and lastly **t3** (as it is the first to activate).

2.3 Inside the Structure of OT/J

This section will take a look inside the infrastructure of OT/J, and reviews in some detail the components of that infrastructure. As mentioned earlier in the introduction of this chapter, OT/J comprises two subsystems; the Object Teams Development Tooling (OTDT) and the Object Teams Runtime Environment (OTRE). Figure 2.4 illustrates the structure of OTDT subsystem. OTDT is responsible for the compilation process of teams and role classes as well as the *playedBy* relationships. Additionally, it includes the necessary Eclipse plug-ins that support writing correct OT/J syntax and to verify all OT/J keywords and expressions in the source code.

When compiling team classes, the OT/J compiler – depending on whether a team class encloses bounded roles or not – injects a set of special-purpose class that attributes into the bytecode of team class as well as it adds the implementation of roles and methods of the parent team class (`org.objectteams.Team`). The compiler then generates for each declared role an inner-interface in the team class under compilation along with an implementation inner-class to that interface. For those roles that are bounded to base classes, the compiler adds to team class the necessary code segments that realize *translation polymorphism* for each role type. The Translation Polymorphism is a mechanism that organizes the way a team instance should handle base and role instances once they are coupled at runtime (will be discussed in Section 2.3.1). With respect to CIMBs, the compiler adds for each CIMB declaration a Team-Level Wrapper Method (TLWM) that intermediates the access of a role-level method when the base object method intercepted by that CIMB. The compiler also generates the necessary methods that facilitate the GPs and BGPs evaluation at their different levels.

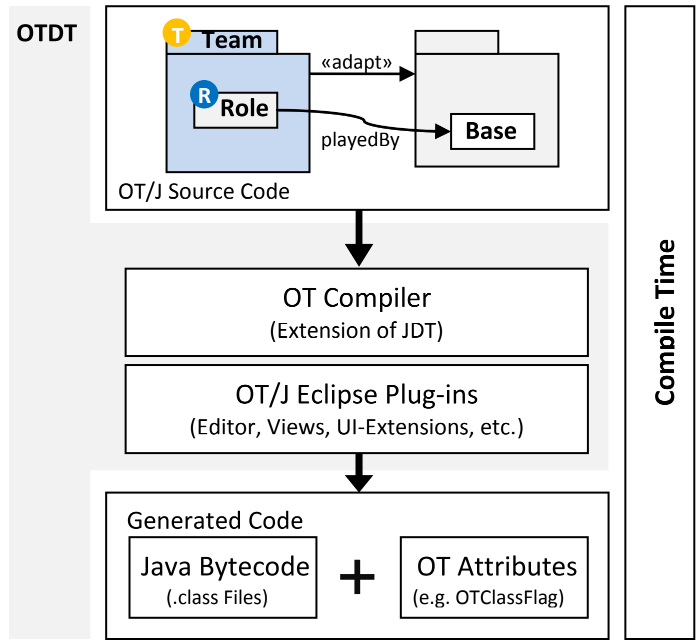

Figure 2.4. The general structure of the OTDT.

The second subsystem of OT/J is the OTRE. This part of OT/J is responsible for facilitating the execution of OT/J compiled application classes. The main operation performed by the OTRE is the transformation of bytecode of the *targeted* application classes at load-time. Figure 2.5 illustrates the structure of OTRE.

At the first implementation version of OT model, OT/J uses JMangler [29] to transform the different application classes at the load-time. JMangler is a load-time transformation system that augments Java Class-Loaders by a custom class-loader that enables developers to modify the bytecode of classes before they are actually loaded into the JVM for execution (see Appendix-B for more details about JMangler). To manipulate the bytecode of classes, OT/J uses the Byte-Code Engineering Library (BCEL) [50] in order to instrument the bytecode and then weaves the required code segments in base and team classes to prepare their instances for precise coupling, and hence accurate role playing.

In practice, the BCEL simplifies the treatment of classes' bytecode as it reclaims their descriptions first, which makes the manipulation of these classes friendlier. Then, BCEL provides a wide set of APIs that make the creation of code segments in fly so easy. Thus, OTRE implements a stack of bytecode transformers (see Figure 2.5) where the output of one transformer is the input of the next one. Currently, the recent versions of OTRE support bytecode transformation by using the Java Programming Language Instrumentation Service (JPLIS) [91]; but still support using JMangler class-loader as well. JPLIS allows the dynamic transformation of application classes, as well as transforming system classes' bytecode of Java (like `java.lang.String`), which enables their objects to play roles of OT/J applications.

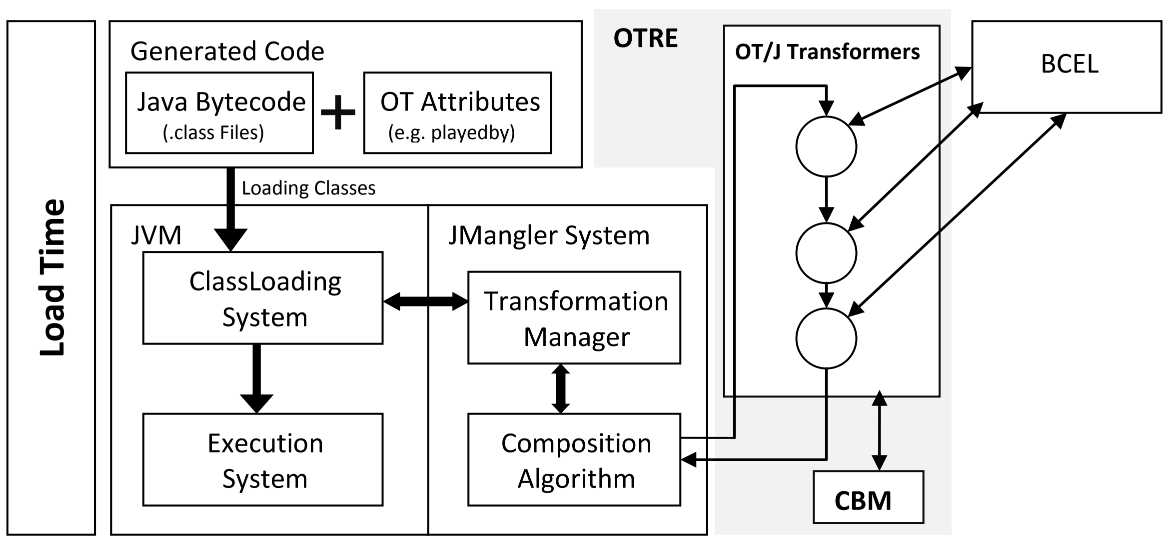

Figure 2.5. Integration of Object Teams Runtime Environment (OTRE) and the JMangler system.

Part of stack of the OTRE bytecode transformers is dedicated to transform the base classes, and the other transforms team and role classes. The most important operations the OT/J's transformers perform on the team classes are: (1) inject the invocations of the implicit (de)activation methods, (2) add the methods of registering team instances in and un-registering from base classes, and (3) assign a unique team-ID for each team class. In respect of the bounded role classes, OT/J transformers add to the role-level methods the necessary method invocations for implicit (de)activation of the enclosing team instance.

At the transformation phase, all targeted base classes are transformed by Base Transformers. These transformers mainly reformulate the code of bases' methods that are bounded to role-level methods through CIMBs. The new versions of base methods are supplied with the capability to check, at runtime, the activation state for all registered team instances of that base object, and then execute the corresponding TLWM that is associated to the required role-level callin method.

Note from Figure 2.5 that all application classes are loaded by the JMangler system, and then they pass through all OT/J transformers before they proceed for execution. This implies that information about callin methods, teams IDs, and other attributes (e.g. parameters mapping plan, player class name...etc.) need to be *stored* at a shared place so that transformers at the bottom of the stack can refer to this information as necessary. In fact, OT/J resolves this problem partially by injecting some of this information as class and method attributes inside the classes' description at the compile-time (see Figure 2.4), and completes the solution by storing the rest of information in the shared component called as Callins Binding Manager (CBM). This component is shared among all transformers in the stack, and provides them with the necessary information on-demand, like, *playedBy* relationship properties and CIMB expressions.

2.3.1 Lifting and Lowering Mechanisms [77]

In practice, role instances are not equivalent to their base objects. For example, using the Java operator **`instanceof`** will return "false" if it compares a role instance and its base object. On the contrary, at the conceptual level where role instance and its base object are considered as a single unit of modularity. In OT/J, the runtime system is adherence to couple exactly one role instance to exactly one specific base object, so that both entities are considered as one aggregated entity [78]; even if each one of them has different object identity. Moreover, it was shown in [73] that this multiple identity of objects results with the Object Schizophrenia phenomenon, which may lead to the Object Schizophrenia Problems that may impact application integrity. In practice, OT/J prevents this problem occurrence by *switching* interchangeably between role and base instances inside team instances through two harmonious operations called *Lifting* and *Lowering*. In [76] it was shown how the OT model helps to avoid problems that could arise from dynamic multiple roles.

2.3.1.1 Lifting

When the control flow is transferred from a base object towards specific team instance in case of a CIMB execution, then that team instance picks up a role instance corresponding to that base object to complete executing that CIMB. This process, i.e. *lifting the base object to its corresponding role instance*, is performed by team instances transparently without the knowledge of application programmers. Even though, OT/J allows programmers to explicitly lift base objects into role instance through so-called *Explicit Parameters Lifting* declarations. As an example, in Listing 2.2 at line 24, the expression (`Person as Employee _p`) in the declaration of team-level method `hired (Person as Employee _p)` explicitly indicates that when passing a base object of type `Person` to this method, a *prope*r role instance is lifted. To do this precisely, team instance reclaims role instances from its own cache if the specific role instance exists. Otherwise, it creates a new role instance, binds it to the passed base object, and then puts the new role instance in its cache.

To summarize, lifting a base object into a role instance is a transparent operation that is carried out by team instances whenever that base object passed the boundaries of team instance in. The actual binding of role instance to base object occurs at the first time that base object is lifted. Consequently, role instance preserves an immutable reference to its base object.

2.3.1.2 Lowering

The opposite of lifting in OT/J is lowering. This process picks the base object reference that has been preserved by the role instance whenever the control flow is to be dispatched to that base object, for example, in case of fetching the value of a base field. Mainly, a role instance is lowered into its bounded base object when the execution of specific COMB is encountered (recall that *all* COMB executions are forwarded into the base object). Lowering operation in OT/J is considered as a synonym to Java™ inheritance at objects' level (i.e. role instance inherits its base). The difference between the two inheritance mechanisms is that the role instances do not dispatch their self-reference in the delegated calls as in Java (since role instance doesn't know that a base method is invoked on behalf of its method [73]).

To sum up, the role instance and its bounded base object are conceptually considered as a single unit of modularity, but practically they are different entities. The mechanisms of lifting and lowering, together, resolve the translation polymorphism when the control flow is passing the boundary of team instance in and out during the role playing process.

2.3.2 Understand the Control Flow between Roles and their Bases

As stipulated in the Confinement Rule of roles in OT/J, role instances are accessible only inside the boundaries of the team instance enclosing them. This restriction prevents base objects to invoke any of role's methods or access any of their fields. In case of CIMBs execution, OT/J provides a transparent intermediation between the two objects (base and role) to overlap this restriction. In practice, OT/J compiler implicitly generates for each declared CIMB a Team-Level Wrapper Method (TLWM) inside the team class that comprises these CIMBs. On the contrary, through COMBs a role instance can *directly* access the methods of its bounded base object. In fact, the coupling between role instance and base object results with that role instance preserves an immutable reference to the base object while the base object preserves a reference to the team instance that confines that role (since team instances register themselves at base classes upon activation).

To explain the how control flow is dispatched during CIMB and COMB execution, let us trace an invocation of method `getInfo()` of the **Person** base class (as shown in Figure 2.6). The Figure shows three instances: a base object **b** of type **Person**, a team instance **t** of type **Company**, and a role instance **e** of type **Employee**. When the control flow enters instance **b** to execute method `getInfo()`, it will be intercepted by the CIMB `{getInfo <- replace getInfo;}`.

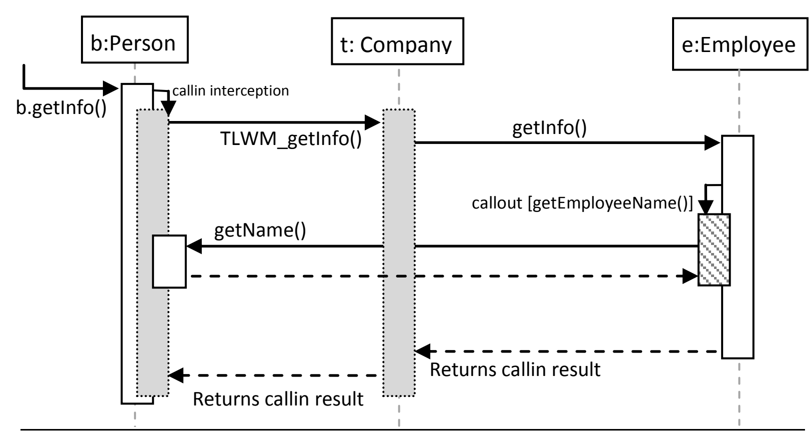

Figure 2.6. Control flow between role **e** and base **b**.

Consequently, the control flow is dispatched to team instance **t** to execute the corresponding TLWM (for simplification it will be given the name `TLWM_getInfo()` as shown in the Figure). As the team instance knows exactly which of role methods is bounded to this TLWM, it *lifts* and uses **e** instead of **b** which has been transparently passed at the interception time. Afterward, **t** invokes the required callin method. It happens in this example that **e**'s callin method `getInfo()` invokes, in turn, the callout method `getEmployeeName ()` (see Listing 2.2 line 18). Thus, the control flow must be dispatched to execute that callout method. For that, a transparent *lowering* process is carried out to forward the execution of this callout method to the bounded base object **b**. Note in Figure 2.6 that a *direct* invocation from **e** to **b.**`getName()` is performed, and the result is returned to **e**. On the contrary, the CIMB execution returns the expected value back indirectly to the base object **b** by the intermediation of **t**.

2.4 Advanced Role-playing

Besides the simple role playing scenarios which had been presented so far, OT/J enables more advanced composition scenarios for team and role classes, and more advanced role playing combinations as well. This capability enables developers to develop more complex applications. Therefore, these advanced features, namely: Nesting, Stacking, and Layering will be discussed.

2.4.1 Nesting

In simple words, nesting means that a team class can declare, to non restricted depth, other team classes (called role-teams). By this way, the inner team class will obtain both the role and the team characteristics at the same time, i.e. it simultaneously acts as a team and as a role.

Conceptually, nesting teams and roles provide advanced and complex compositions for applications. Practically, with the nesting and Java inheritance, OT/J presents a new type of inheritance called Implicit Inheritance [79](§1.3). This feature implies that the nested modules are inherited implicitly by the inheriting module; thus, they are overridden or specialized further. In this regard, the keyword `tsuper` is used in OT/J along specific type names (i.e. a qualified `tsuper`) to invoke the corresponding method or constructor of a super team to the overriding method or constructor where `tsuper` invocation appears.

2.4.2 Stacking

In a programmatic perspective, stacking means that a specific bounded role is played by a team. In this case, a player team (called base-team) acts as a normal base class in that any of its methods could be intercepted by CIMBs and accessed through COMBs.

2.4.3 Layering

This feature supports advanced role playing scenarios. Layering enables roles of specific team class to play the roles of another team class; i.e. roles of the first team are base classes. In this case, the team class, which encloses the played roles, defines a layer over the team class that comprises the base roles. In practice, layering imposes preserving a final reference of players' team in the roles' team. For example, **TeamA** in Figure 2.7 must define a final team instance of type **TeamB** as [**`final`** `TeamB teamB =` **`new`** `TeamB();`]; since **TeamA** defines a layer over **TeamB**. Layering insures that all base objects bounded to roles of **TeamA** are contained in the defined anchor instance.

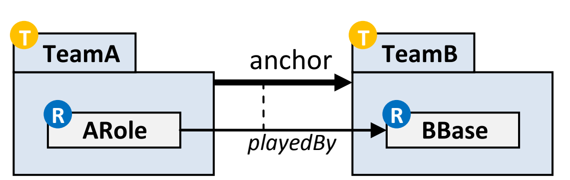

Figure 2.7. Team class **TeamA** layers over **TeamB**.

2.5 Adaptation of Application Objects: a Perspective of OT/J

The definition of application adaptability mentioned in Chapter 1 implies two main issues: *First*, application objects need to be equipped with a mechanism that enables them to adapt and change their behaviors precisely; here, precision refers to carrying out this adaptation without any deformations to the application consistency and objects structural integrity. *Second*, a modular and expressive mechanism is needed to modularize the new desired functionalities, and to specify when application objects should adapt through exploiting these functionalities and how?

In case of dynamic adaptation of distributed application objects, and in order to support adaptive applications, the middleware on which application objects execute must primarily addresses the following key questions:

- When to adapt? The ability of components to detect the proper time to adapt.
- What to adapt? Answer to this question must determine which elements and components of the application are subject to adapting their behaviors.
- How to adapt? Determines the most beneficial adapting mechanism to be applied such that the application moves from a consistency state before adaptation to another consistency state after adaptation.

Feasible answers on these questions are inspired from the capability of the OT model to support application objects adaptation (in particular, behavioral adaptation of centralized applications) in the name of role-playing. Thus, to apply the same technique on distributed applications, a precise mapping for the OT capability to distributed environments must be achieved.

2.5.1 The Essence of Roles in OT/J – A Revisit

Basically, a bounded role could be seen as the individuality of a base class inside the team class that encloses this role. This individuality might be of short-term (i.e. temporal) or long-term characteristic type. For example, "student" is a temporal characteristic of a "person" entity that cannot be a student forever. Also, the adaptability of legacy applications means that application components need to adapt their behaviors permanently or for long time. This semantic meaning of role module conforms to the role specifications mentioned in [25], in which roles in programming world considered as those classes that are not really classes, but roles. Moreover, in the contiguity between roles and the concept of *agents* mentioned in [26], roles are considered as conceptual tools to model the digital world in a similar way to how the real world is modeled.

In Java's conventions, objects can acquire new functionalities through inheritance. Therefore, objects *extend* their behaviors by delegating calls for these new functionalities to their parents. In OT/J, the integration between a role – as a module that represents the new characteristic of specific type of objects – and the process of behavior expansion of objects, enables us to conclude the following two facts of roles:

1. In terms of concept, the relationship between a role and its bounded base class merges them together to form a single unit of modularity. When zooming in this relationship, a role module, in fact, is a special individuality of that base class (e.g.

employee is a special individuality of a person). This relationship could be translated as a kind of class inheritance (i.e. roles inherit their base classes). Therefore, a role instance can *extend* its base object as it knows everything about that bounded base object. The important difference between role-base inheritance and normal Java inheritance is that in the Java language, the JVM transparently creates a parent instance at the construction-time of a child in order to handle the call delegations. Conversely, in OT/J, the existence of role instances *completely* depends on the binding between them and the already exist base objects. Then, the following OT aspects strengthen further the consideration of role-*extends*-base relationship:

a. The capability of a role instance to override *any* of its base object methods through replacement CIMBs (i.e. CIMBs with `replace` modifier).
b. The capability to access any of base object's fields through getter and setter COMBs.
c. The fact that role and its base objects have different object identities is not an issue [73].

2. According to the fundamentals of AOP, roles are aspects, i.e. they capture the crosscutting behavior of base objects over collaborations. Following the AOP concepts, roles can define advices and pointcuts using CIMB expressions (with `after`, `before`, and `replace` modifiers). These CIMBs specify when advices should be executed and how? Whereas advices specify what to execute (i.e. how objects' behaviors adapt).

   In this scope, OT/J provides – through the one-to-one CIMB – an expressive advising which is understandable and controllable more than, for example, the pointcuts of AspectJ that include the "*" wildcard [95]. Thus, it is easy to express in OT/J that a *person* must adapt his behavior when he asked about his information by telling also his information as an employee in a company as long as he has this role in that company. In AspectJ and alike, however, the same case could not be expressed. Though OT/J does not yet implements a Join Point Model (JPM) like AspectJ and JAsCo, it appreciably provides better application modularity and expressivity.

When adding the two facts together, modules (i.e. roles) that can comprise a specific functionality could be obtained, and through the *playedBy* relationship, this functionality is employed modularly at specific application objects.

2.5.2 The Behavioral Adaptation of Objects

Application self-adaptation, as cited in [34], does *not* mean that application obtains the logic to "adapt" itself while the code is pre-existed in components, i.e. embedded as a part of component description. Contrariwise, the behavior of an operation in self-adaptive application could be dynamically modified by changing the code of that operation, e.g. by replacing the way that operation executes and not the operation itself.

Thus, the dynamic behavioral adaptability of an object will be redefined as follows: "*The ability of that object to execute a new behavior instead of its original one whenever it is supposed to do that.*" The execution of new behaviors must be controlled by modularity; i.e. programmers should specify in a modular way why, when and how objects can adapt. In

AOP, separating object's core logic from the non-functional behaviors improves the modularity to accomplish functional adaptability of objects [7].

To summarize the idea, application objects can adapt and change their behaviors as they can play different roles in different teams. The realization of object adaptability through role-playing is conducted by the following viewpoints:

- *The team activation mechanism*: The process of role-playing is totally depending on the activation status of team instances. This fact helps us to determine when application objects can adapt? From an adaptability perspective, the answer of this question is related to changes of execution environment. In this regard, it was shown in [23] that monitoring the execution environment is an important fundamental to implement adaptable applications. When the monitoring service encounters specific change at execution environment, application components need to adapt adequately. From role-playing point-of-view, implementing the monitoring services as teams, and then bind the changes of execution environment to activation/deactivation operations, a modular and controllable adaptation of application objects could be obtained.

- *Shall all objects adapt or some?* It may be necessary to refine adaptability. That is, application developers should have the capability to modularly select among application objects those that can adapt and not all of them. In this regard, the BGPs can play a significant role in refining which of application base objects are able to play a specific role; thus adapt. In addition to, and depending on the level of BGPs, programmers can flexibly control the effects of role's callins; hence objects' adaptation itself.

- *The precedence of roles*. In real world, a person can have simultaneously two or more different personalities (for example, a person can be a father and an employee). However, at any specific point of time, that person behaves only according to one of these personalities. In the programming world, OT/J represents such situation and captures these personalities in two different roles that could be played in different teams. It allows application programmers to determine which one of roles acquires the highest priority, for example, in case both declare CIMBs which bound to the same base method. This capability allows us to choose among different adaptation strategies through modifying

2.6 Summary

This chapter has presented an overview of the key features of OT/J programming language (the language that implements the OT model). It has illustrated through examples how application developers could write applications using OT/J in a high degree of modularity and expressivity. The chapter has figured out how the fundamentals of the OT model could be employed to improve the implementation of the behavioral adaptation of application objects. Finally, it has paved the road (conceptually) to a modular mapping for these fundamentals to distributed environments. The next chapter will propose the outlines of this mapping and introduce the Remote Role-Playing concepts.

Chapter 3

Remote Role Playing

This chapter presents the concepts of remote role playing. The chapter discusses the process of mapping the features of the OT model into a distributed environment. It describes, through examples, the practical steps which have been followed to accomplish this mapping: First, it discusses a simple remote role-playing scenario, and addresses the requirements for mapping this scenario to a distributed environment. Second, it determines which of OT/J application components need to be enrolled in the remote role-playing process.

3.1 Introduction

The OT model is rigorous with respect to the way it handles role instances at runtime, but it is flexible in representing them at the source code level in a seamless fashion with a high degree of understandability. Therefore, mapping the fundamentals and concepts of OT to a distributed environment must not violate these features in order to make use of the different OT capabilities like the improvement of applications modularity [73, 75], the good understandability of applications' structures (hence, easy maintenance) [3], the dynamic management of aspects (roles) at runtime [2], etc.

In practice, distributed applications have different properties than single (or centralized) applications; e.g. in the way their components are represented at the source code level and then accessed at runtime. Therefore, employing the concepts of the OT model (mainly, role-playing) in distributed environments imposes the integration of the properties of distributed and single OT/J applications. Keep in mind that the main goal of this dissertation is the realization of Dynamic Adaptation of Distributed Application Objects (DADAOs) in a modular and expressive fashion. Consequently, applying the OT concepts on separated objects that are executing on different application nodes requires the *modification* of these objects in order to obey the OT role-playing specifications [3].

It is important to distinguish between the capability of using OT/J programming language to develop distributed applications and employing OT fundamentals in distributed environments. That is, application developers can implement distributed team and base objects, but they cannot connect (via the *playedBy* relationship) role and base objects that are residing on different hosts in the normal way. Hence, mapping the OT features to distributed environments is mandatorily meant to provide this connection [3].

3.2 Why Remote Role-playing?

To realize the DADAOs, objects of distributed application need to *change* their behaviors dynamically in a modular and systematic way, which can determine when to adapt, which to adapt, and how to adapt [3]? In fact, OT supports the adaptation of objects and the modular representation of this adaptation. That is, a base object is said to *adapt* and change part (or all) of its behavior when it plays a specific role, and if that role has declared a *replacement* callin method (i.e. a role's callin method with the "`replace`" modifier). This replacement of functionality is considered as a functional adaptation; because base object executes a method that is different than the original one. This functional adaptation works excellently in sequential applications but not in case of distributed applications.

The backbone of the OT model, however, is the *role-playing* concept, which is expressed by the *playedBy* relationship. This relationship could be considered as a modular adaptation toolkit for the application base objects decorated by roles [78] (decoration here means the binding of a role class with a base class). Thus, exporting this relationship into a distributed environment will bring several benefits: first, the way distributed application objects can adapt and change their behaviors could be modularized; because this adaptation is accomplished through the modular role and team, which improve applications' modularity in general. Second, it separates the original behaviors of base objects from the new desired adaptation functionality; thus it allows more control on adaptations. Third, it will improve the modularity of distributed adaptable applications through an expressive and understandable relationship. Finally, application base objects can adapt in consistent manner as they play roles simultaneously (which is equivalent to *atomic aspects weaving* [20, 51]). Therefore, the *Remote Role-playing* (RRP) [2 and 3] is introduced to achieve these benefits in distributed environments.

3.3 A Simple Remote Role-playing Scenario

This section will present a simple arbitrary OT/J application, and then explains how it could be mapped to a distributed environment. Thus, a clear concept for the RRP would be created. Consider the typical OT/J application shown in Figure 3.1. The figure shows a *playedBy* relationship between the base class **MyBase** and the role class **MyRole**. The role class has been enclosed in the team class **MyTeam**. As a first step to describe the mapping of this application to a distributed environment, the base and team classes are drawn each inside a dashed rectangle to indicate that they are *separated*, i.e. the instances generated from these classes will reside on different locations (i.e. hosts or processes).

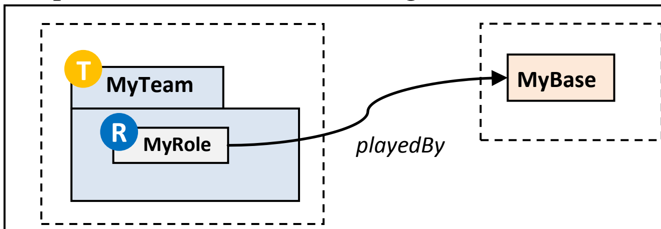

Figure 3.1. Envision of a typical OT/J application over a distributed environment.

To emphasize the RRP better, let us generate several objects from the base class **MyBase** and deploy them on several hosts[1] as shown in Figure 3.2 (a). Naturally, base objects need to *communicate* with a team instance of **MyTeam** in order to construct the

[1] Host names H1, H2, and H3 are used to indicate the physical distribution of objects. This representation could be used to indicate objects' logical distribution; i.e. the processes in which objects execute.

playedBy relationship. Now, establishing this communication in this case is impossible without deploying an individual team instance of type **MyTeam** on each host on which a base object has been created; as illustrated in branch (b) of the figure.

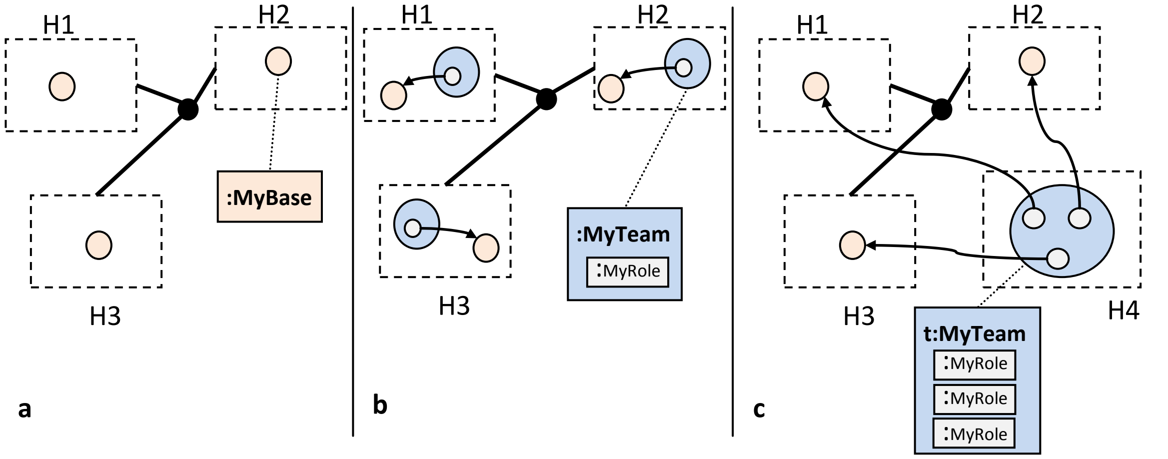

Figure 3.2. Several deployment scenarios for the objects of the application shown in Figure 3.1: (a) deployment of base objects, (b) deployment of several team copies, and (c) deployment of a single team instance while enabling playing of roles remotely.

3.3.1 Disadvantages

In practice, this solution has several disadvantages. First, the deployment of several team instance copies must be handled manually by developrs, which adds extra efforts if a slight change to the team class has been made [3]; because application developers should re-deploy team instances once again. This means that application execution must be stopped at all nodes on which team instance copies have been previously deployed. In fact, this violates the conditions of the DADAO mentioned in Definition 1.1, in particular condition number 4.

Furthermore, the state of all team instance copies (e.g. the activation status) must be *consistent* and *synchronized* at runtime all the time in order to ensure that all base objects play roles *identically* [3]. To achieve this, a mechanism is required to ensure that state changes to a team instance will be synchronized on all nodes immediately. Actually, this will complicate the design and implementation of the role-playing process, and makes the management of team instance copies very hard. Moreover, it prevents the dynamic weaving of new roles at runtime; because only those roles which are woven at the load-time level will be played.

3.3.2 A RRP Relationship

A better solution could be achieved if a single team instance is deployed, and the separated base objects are allowed to play their roles *remotely* and independent of their locations. Figure 3.2 (c) depicts this situation. In this way, the characteristic of "team" module for being a single container of role objects is preserved. In addition, the state of every single team instance could be manipulated individually rather than manipulating several states of several "sub-team" copies in a synchronous fashion; because the consistency of these copies is required. Moreover, the problem of team *Class Versioning* is avoided. Thus, when a change is made to the team class, then there is no need to redeploy multiple team instances; in which a team instance might not be redeployed on a specific node unintentionally.

3.3.3 Requirements of RRP

A precise mapping of the *playedBy* relationship shown in Figure 3.1, in order to realize the RRP relationship shown in Figure 3.2 (c), should determine which of the entities involved in that local relationship need to be *considered* in the RRP. In addition, an appropriate communication layer underneath these entities should be provided in order to realize a true RRP relationship that both preserves the semantic of the *playedBy* relationship and helps to realize the DADAO. As a first step, it is clear from Figures 3.1 and 3.2 that the following entities are identified:

1. Objects of the base class **MyBase**.
2. The team instance of type **MyTeam**.
3. Role instances of the role class **MyRole**.
4. The *playedBy* relationship; as a link between these entities.

The base objects deployed on hosts H1, H2 and H3 as shown in Figure 3.2 (c) need to acquire *transparently* a reference to team instance **t** (which encloses their roles) in order to read its activation status accurately. Also, once calls of their methods have been intercepted by specific Callin Method Bindings (CIMBs), these base objects must be able to *select* the appropriate Team Level Wrapper Method (TLWM) in team **t** that is corresponding to the roles' callin methods in each CIMB. But, as only a single team instance has been deployed on host H4, base objects must be provided with a *remote reference* to team instance **t**. Thus, the team instance **t** must be represented as a *remote team instance* in order to allow accessing it remotely [3].

On the other hand, the role instances enclosed in the team instance **t** need to *acquire* and *preserve* immutable references to their base objects in order to issue the declared Call-Out Method Bindings (COMBs) and evaluate Base Guard Predicate (BGP) expressions. Role instances of **MyRole** must always be confined within the boundaries of team instance **t** according to the *role confinement rule* [79]. Thus, to provide them with *valid* references to their base objects, these base objects must "export" themselves into the distributed environment as *remote base objects*. This means that the role instances of *remote team* **t** acquire *remote references* to their players instead of the ordinary local references.

In OT/J, the *playedBy* relationship shown in Figure 3.1 is compiled by the OT/J compiler in such a way application base objects, role instances, and the team instance are *taught* to communicate locally. That is, the *playedBy* relationship only binds local base objects to local role instances in a local team instance. Thus, it cannot bind the remote base objects shown in Figure 3.2 (c) to the role instances of remote team instance **t**.

3.3.4 Remote Communication Channel

Our efforts to realize this RRP scenario will concentrate mainly on establishing a *remote communication channel* that can connect the *remote team instance* **t** and the *remote base objects* deployed on hosts H1, H2, and H3 together. To address the requirements of this communication, the following features of the OT model, which should be taken into account in the design of the desired remote communication, have been assembled:

1. In general, the base objects of an application *must* be aware and able to detect which of team instances comprise any of their roles at runtime. In addition, they should be able to check CIMBs' matching at their methods' calls. In OT/J, this is achieved via weaving roles into base classes by the bytecode transformers of the Object Teams Runtime Environment (OTRE), which inject in the bytecode of base classes the methods and attributes required by base objects to detect the currently activated team instances. But, due to the loose coupling between the remote base objects of the application shown in Figure 3.2 (c) and the team instance **t**, a smart mechanism is required to bind the two groups of entities together *before* any of base objects' methods is being invoked.
2. Executing the roles' callin methods associated with CIMBs cannot be performed directly by the remote base objects; because of roles confinement rule. In fact this is the case in all OT/J applications; base objects *can* invoke roles' callin methods via the proper TLWMs. The invocation statements to all TLWMs are injected into base classes by the OTRE at load-time. Thus, a precise mechanism is needed for intercepting the remote base objects' functionality by the CIMBs declared in the role class of the remote team **MyTeam**. In addition, a seamless, secure and accurate execution of the roles' callin methods *remotely* should be ensured.
3. The applicability of role-playing is governed by the *team activation* process. More specifically, the CIMBs declared in the role class **MyRole** have no effect on the base objects of **MyBase** if the team instance **t** is not activated.

   Practically, it is the responsibility of base objects to detect the activation status of team instances *before* any callin method upon a CIMB interception is invoked. In case of local base objects, these objects can read the activation status of team instances easily because teams *register* themselves dynamically at base classes whenever they are activated, and *unregister* when deactivated. In detail, the transformers of the OTRE inject two *static* methods into the bytecode of each base class in order to handle the teams' registration and un-registration operations.

   To achieve this feature in case of RRP, the remote base objects of our application must be able to *read* the activation status of the team instance **t** every time one of their methods has been invoked. Thus, when the activation status of team instance **t** is changed, all the remote base objects involved in the RRP must perceive this change. Now, the problem is that the remote team instance **t** cannot register/unregister statically in the remote base class **MyBase** due to distribution; i.e. developers cannot invoke a static method of a remote class. Later in this chapter, this issue will be revisited to explain how remote base objects can read the activation status of remote team instances precisely.

3.4 Remote Team, Remote Role and Remote Base Modules

The term "physical location" of an object will be used to indicate the network node (host) that hosts the JVM on which that object is executing. Conversely, the phrase "logical location" will be used to point out to the process in the JVM in which that object executes besides other objects which reside in the same JVM but execute in different processes. Formally, the location of an object will be defined as follows:

Definition 3.1: The Location of Distributed Application Objects

The location of object **obj** is loc (**obj**) = L_{obj}; where **L** = { The physical Host name, or / The Process in which **obj** is executing.

This section will redefine – from a RRP point-of-view – the OT concepts of team, role, and base modules. In each subsection, it presents a formal definition for the remote version of each of them, and demonstrates the effects of the mapping process on them.

3.4.1 The Remote Team Module

The team module in OT has the following characteristics:

1. It is the context in which objects of an application play their roles.
2. It is a container of role objects.
3. It is a component of services in which each role can represent a specific service.
4. In the collaboration-based design, teams are collaboration modules.
5. Teams could be instantiated, and their instances could be accessed as normal objects.

In case of RRP, team classes that enclose the remotely played roles will be called *remote team classes*. The *Remote Team* will be defined as follows:

Definition 3.2: The Remote Team

A Remote Team **RT** is an OT team class which comprises at least one role class (say **R**) that is played remotely by a specific base class (say **B**), OR the team which its instances need to be accessed remotely. We say that **RT** is a remote team and **RT(R:B)** is the relationship that binds role instances of **R** with the base objects of **B** <u>iff:</u>

loc(**R**) = loc(**RT**) AND loc(**RT**) ≠ loc(**B**).

At the design level, and in order to distinguish between normal and remote team classes, the *Antenna Symbol* ( ) [1, 2, and 3] has been invented to mark remote teams (as well as all other entities that participate in the RRP). For example, to mark the team class **MyTeam** (see Figure 3.1) as a remote team, the antenna symbol is added on top of its UFA notation as shown in Figure 3.3.

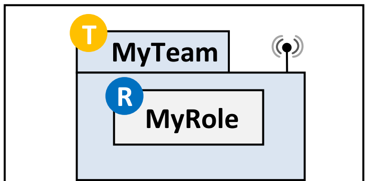

Figure 3.3. Re-represent teams as remote teams.

3.4.2 The Remote Base Module

Base classes, and hence the generated objects from these classes, are the players of roles. In OT/J, base classes could be the classes of legacy applications (in which their source code might not be available) or user-defined classes. However, mapping a base class into the distributed environment should be the same regardless of its source type. Thus, a remote base class could be any of the application classes *except* when it is a nested-team, or roles of a specific team class (called *layering compositions*). Advanced OT/J compositions are discussed later in this chapter.

The *Remote Base* class will be defined as follows:

Definition 3.3: The Remote Base

The base class **B** is a remote base class if an object **rB** generated from that class can play the role **r** within the remote team instance **rT** such that:

loc (**rT**) $\neq$ loc(**rB**), and **B** is not a nested-team or a role of another team.

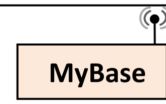

Figure 3.4. Re-represent base classes as remote bases.

At the design level, remote bases are represented as normal UML class blocks with the antenna symbol set on top of them. For example, the base class **MyBase** shown in Figure 3.1 could be redesigned as a remote base as shown in Figure 3.4.

3.4.3 A Remote *PlayedBy* Relationship

In OT/J applications, the *playedBy* relationship is the only clause that can give information about the semantic relationship between the base and role classes. Application developers declare *playedBy* expressions to ensure a secure and firm binding between the base objects and their role instances at runtime. This expressive and firm binding must be preserved in case of the RRP, especially with the loose coupling of distributed application objects. Therefore, a "remote relationship" to link between the remote base objects of a distributed OT/J application and their remote role instances is required.

At the design level, this remote relationship is distinguished in application diagrams via the antenna symbol drawn along the *playedBy* relationship of OT/J applications. For example, the diagram of the application shown in Figure 3.1 could be redrawn using the *remote playedBy* relationship as shown in Figure 3.5. Note that the antenna symbols are the removed from class diagrams of **MyTeam** and **MyBase** to indicate that zremote *playedBy* relationship binds (by default) remote bases and the roles of remote teams.

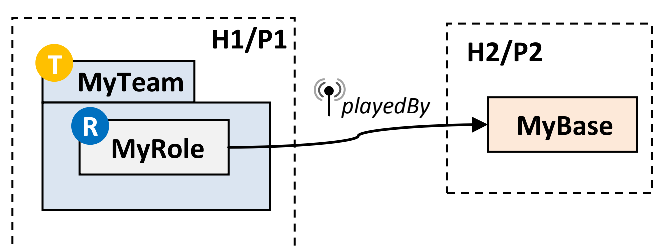

Figure 3.5. The *remote playedBy* relationship is represented as normal *playedBy* relationship with the antenna symbol.

Recall that the local *playedBy* relationship involves a communication channel that comprises two types of method bindings: the CIMBs and the COMBs. Therefore, the *remote playedBy* relationship must support *remote* method bindings (discussed later in detail). Furthermore, the evaluation of Base Guard Predicates (BGPs) must be supported in order to ensure accurate RRP; hence, accurate remote base objects adaptation. The following points discuss each of these remote concepts from a RRP perspective:

1. The Remote Callin Method Binding concept (remote CIMB):

The importance of CIMBs is not limited to intercept base objects' functionalities. Rather, the matching of a CIMB with a method call of a base object (i.e. the joinpoint matching in the AOP) is one of two situations where binding between the base object and a role instance could be established[1]. Specifically, when a base method of a specific base object is intercepted by a declared CIMB, then the base object *dispatches* itself as a parameter to the TLWM of the enclosing team instance. Consequently, team instance can easily *lift* that base object into a specific role instance; or creates a new one if no bounded role exists in the cache. Thus, it completes the binding process.

The same interception mechanism to base methods calls by CIMB expressions need to be realized in case of remote base objects. Also, transparent joinpoints matching at runtime needs to be accomplished to execute the associated roles' callin methods remotely. For example, suppose that the role **MyRole** declares a callin method named `rm`, and declares a CIMB expression, which replaces the calls of a specific base method (say `m`) in the remote base class **MyBase** as follows:

```
rm ← replace m;
```
(3.1)

Then, the RRP is requested to provide a mechanism that enables the interception of calls to `m`, and invokes remotely the method `rm` instead. Subsequently, the binding between the deployed remote base objects on H1, H2, and H3 (see Figure 3.2 (c)) and the role instances of remote team instance **t** must be accomplished in a similar way to the mechanism adopted by OT/J. But, several obstacles are facing the achievement of this mechanism:

1. In an OT/J application, all base objects know "*in advance*" the teams' TLWMs they should invoke when a CIMB is matched. For this purpose, the OTRE transformers weave into the base classes all declared roles in the correct precedence (in case of multiple teams). This technique, however, restricts the dynamic playing of roles at runtime in that only the woven roles could be played. Moreover, in RRP, the dynamic attachment of new remote team instances at runtime without interrupting the execution of the distributed application is a condition of the dynamic adaptation of objects.
2. In the RRP, the remote team instances cannot register/unregister in/from remote base classes as in OT/J when they are activated and deactivated, respectively.
3. The transformers of OT/J modify only those base class methods which are bound in CIMB declarations. The transformation process injects the suitable TLWMs' calls according to the static role-playing map that is known at the load-time. Therefore, weaving new roles could not be achieved after base classes are loaded into the JVM.

Let us define *Remote CIMB* as follows:

Definition 3.4: Remote CIMB.

A Remote CIMB is an expression that binds between the role-level method **rm** declared in the role instance **r** and the base-level method **m** of the remote base object **b**, such that:

- the role **r** is:
  - (1) Confined in the remote team instance **rT**
  - (2) Played by the remote base **b.**
- loc(**b**) $\neq$ loc(**rT**).

[1] The second situation is achieved by the "explicitly lifting parameter" declarations.

Then, binding is represented as a vector of 6-tuple elements as follows:

$$\textbf{Remote\_CIMB}_{(b)} = <rT_b , R_b , B_b , rm_b , m_b , CM_b>$$

Where:

rT_b : the remote team instance which encloses R_b.
R_b : the role class that declares the CIMB expression.
B_b : the remote base class that plays the role **R**.
rm_b: the bounded role-level method.
m_b : the method of B_b to be intercepted.
CM_b : the callin modifier of the binding (after, before, or replace)

To overcome the aforementioned obstacles, and to realize remote CIMBs according to the Definition 3.4, a *transparent* interception to method calls of remote base objects needs to be supported. In addition, all the callin methods associated with remote CIMBs in remote roles must be executed according to the accurate precedence. The dynamic employment of new roles at runtime, on the other hand, must be supported in order to achieve dynamic adaptation of application objects with the conditions of definition 1.1 (Chapter 1) satisfied.

2. The Remote Callout Method Bindings (remote COMBs) concept:

A *remote* COMB can declare an *expected* role-level method that can either invoke a specific method in remote base object, or access a specific base field (i.e. to get its value or to give it a new value). A bounded role instance cannot issue any COMB at runtime without acquiring first a valid reference to its base object. Therefore, role instances of the remote team instance **t (**shown in Figure 3.2 (c)) need to be able to issue COMBs via the *remote references* of remote base objects of the **MyBase** base class. This imposes a new mechanism to enable role instances to perform remotely the declared remote COMBs.

The *remote COMB* will be defined as follows:

Definition 3.5: The Remote COMB.

A remote COMB is the expression to declare an expected role-level method **eM** in the role class **R** and links that method with one of the following in the bounded remote base class **B**:

- a specific method **m.**
- a field getter that can read a base object field (say **f**).
- a field setter that can change the value of a specific field (say **f**).

Such that; whenever **eM** is invoked in a role instance of **R**, the control flow is dispatched to the bounded base object of base class **B**, and either calls **m**, gets the value of **f**, or sets a new value to **f**.
Where: $loc(\mathbf{B}) \neq loc(\mathbf{R})$.

3. The Remote Base Guard Predicates (remote BGPs) concept:

At runtime, the BGP expressions, which are declared usually along the CIMB declarations and the *playedBy* relationship, are evaluated by team instances just *before* the actual execution of roles' callin methods. The declared BGPs along the *playedBy* relationships are evaluated prior the *lifting* of base objects into corresponding role instances. In either case, the counterpart remote BGPs could be evaluated in similar way, but using a valid remote base reference. In this case, role instance needs to be able to issue base methods' calls and fields' access operations of remote BGP expressions *remotely*.

The *remote BGP* could be defined as follows:

Definition 3.6: The Remote Base Guard Predicate (remote BGP).

A remote BGP is a logical expression declared by specific remote teams or remote role classes, and contains either:

- a call to a specific method of remote base object, or
- an access to a specific field of remote base object, or both of them.

3.5 The Remote Team-Roles

In its concept, OT/J presents so-called *Team-role* modules (i.e. the confined role is a team) to integrate the features of team and role modules in a single module. This allows the formulation of advanced composites in OT/J applications. The team-role class can be a team class that can define additional inner roles, called unbounded team-role or *nested team*. It might be also a bounded role, which is played by a specific base class.

In practice, the implementation of "team-role" module imposes the *locality* of the generated instances (according to the Role-Confinement rule). That is, team-role instances cannot leave the boundaries of their outer team. Therefore, a precise mechanism is required to support a counterpart *remote team-role* in case of the RRP. At the same time, it should respect the *dual characteristic* of that module; i.e. as a team and as a role. Also, the dependency between the *outer* and *inner* team classes needs to be explained.

First, and according to OT/J disciplines, the inner team must obey the activation order rule of team instances, which stipulates: "*the inner team instances can activate if and only if the outer team instance is activated."* Second, the existence of inner team instances totally depends on the existence of outer team instance, i.e. the creation of new inner-team instances *must* be carried out via an outer-team instance (e.g. by invoking a specific method that generate the inner team instances), or through the "role externalization" mechanism [79].

In the RRP, a "team-role" class is considered as a "remote team-role" either if it has been played remotely by a specific remote base class, or if it comprises at least one role class which is played by a specific remote base class. To preserve the characteristics of "team-role" module in both cases, remote team-role instances need to be *manipulated* as *remote team instances* and as *confined role instances* simultaneously. In the former, instances of remote team-roles could be easily accessed as remote team instances; because they could be accessed remotely even though they are inner objects. In the later, manipulating team-role instances as role instances could be left to the OTRE; because access to these instances is handled by the outer team instance.

3.6 Advanced Remote Role-Playing Scenarios

It's easy to make things complex! Thus, design more advanced scenarios for RRP means compose team and role classes, and team and base classes in multi hierarchy levels. In this regard, *nesting* of team classes and *stacking* of bound roles are two primary composing techniques proposed in the OT model [73, 79]. These techniques have been discussed in Chapter 2. However, although such composition techniques support complicated structure of OT/J applications, the fundamentals of OT model like the *playedBy* relationship could be used as usual. For example, base objects will not perceive whether they play roles of a nested team or not.

In case of nesting, the *remote playedBy* relationship should connect role instances of an *inner* team instance with remote base objects without any extra modifications. For example, if the remote team class **MyTeam** (see Figure 3.2) has been declared as a nested team in another outer team class, then the remote base objects deployed in Figure 3.2 (c) will not be affected by this nesting and still be able to play roles normally.

Layering of team classes, on the other hand, requires the team class, which encloses the playable role(s), to hold a *final immutable reference anchor* to the team class that encloses the *base* role [79]. For example, if the base class **MyBase** (shown in Figure 3.1) was declared as a role class inside the team class **MyTeam2** as shown in Figure 3.6. Then, the team **MyTeam** should define statically an anchor instance of team class type **MyTeam2** so that only the base-role instances of **MyBase** (which are instantiated by the anchor instance) could be bound to role instances inside team instances of **MyTeam**. In OT/J, anchoring role types to final team instances is called *role externalization* [79] (§1.2.2(b)).

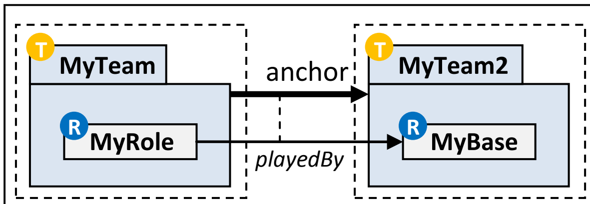

Figure 3.6. Team class **MyTeam** layers over **MyTeam2**.

Mapping of this scenario to distributed environments – according to the concept made previously for the RRP – might not be "*possible*." *First*, the role-confinement requirement in RRP will contradict with the roles' externalization of layering. *Second*, layering of team class **MyTeam** requires the anchor instance to be defined and initialized *before* any base-role instances have been created in the layered team (i.e. **MyTeam2**), which means that a *local* copy of that anchor instance must be deployed on the layering team side; while remote team instances could be accessed *remotely* in RRP.

A possible solution for mapping this scenario to a distributed environment similar to the one shown in Figure 3.2 (c) is by establishing a *remote anchoring* relationship between the layering and the layered team instances. For example, the team class **MyTeam2** might be designed and implemented in such a way it allows remote access of its instances, and enables the layering team instances of **MyTeam** to *preserve* remote references to the base-role objects of **MyTeam2** anchors. That is, the required final anchor instance by **MyTeam** needs to be replaced by a remote reference to the layered team instance of **MyTeam2**, and the created base-role objects must be *synchronized* by sending their remote references to the layering team instance once they are created. The implementation of the layering scenario is left as a future work.

3.7 Summary

This chapter has presented the concepts of Remote Role-Playing (RRP). The terms of *remote team*, *remote role*, and *remote base* have been defined. The mechanism to realize each of these terms in distributed environments has been discussed in concept. The next chapter will describe in detail the implementation of the DOT/J framework.

Chapter 4

The Concepts of Distributed Object Teams

This chapter presents the concepts of the DOT/J framework and the description of its infrastructure. The chapter describes the requirements of a precise implementation of the DOT/J components. The chapter introduces how remote CIMB, COMB and remote BGPs should be executed in a distributed environment in the context of remote role-playing.

4.1 Introduction

This chapter will present the concepts of mapping the OT model's fundamentals to distributed environments. These concepts are the basis of the Distributed Object Teams/Java (DOT/J) framework [1, 2, and 3] implementation. First, the chapter discusses the design and requirements of a precise RRP, and then presents a proposition to accomplish each of them.

4.2 Preparations for Remote Role-Playing

The Java-RMI system has been adopted as a middleware over which the RRP activities are performed. Therefore, to implement remote team, role, and base classes as discussed earlier, remote base objects and remote team instances must be *represented* as remote objects according to the Java-RMI disciplines. That is, each remote base object and remote team instance practically *must* implement at least one *remote interface*. For this purpose, the *required-provided interface* technique [89] is a practical choice to design the communication between remote base and team classes. This technique emphasizes the relationship between the remote objects better, and fits nicely with the *contract-based* design used in Java-RMI system to represent remote objects [6]. Therefore, the classes involved in RRP relationship are converted into *remote classes* [3] as shown in Figure 4.1, which redesigns the application classes depicted in Figures 3.3 and 3.4 in Chapter 3.

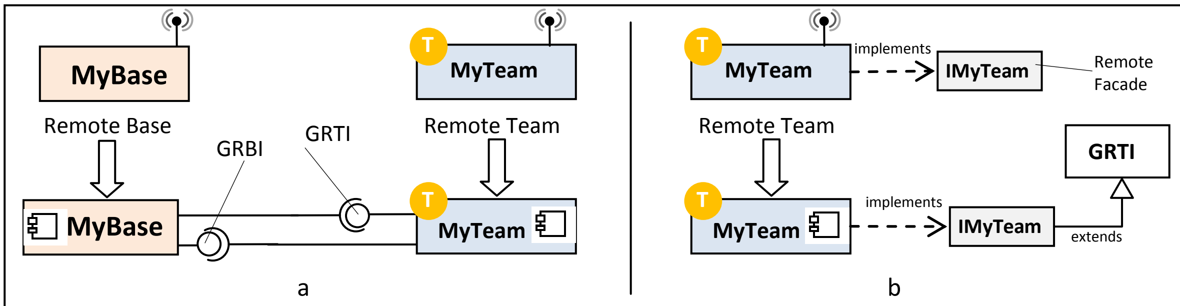

Figure 4.1. The connection between remote base and team classes as a Required-Provided relation.

4.2.1 Remote Base Classes

Remote base objects need to be able to play roles dynamically at runtime without interrupting their execution. Therefore, they should provide a remote interface with the following characteristics:

1. The original functionalities of base objects must be preserved.
2. The structural hierarchy (e.g. inheritance relationships, composition dependencies, associations, etc.) of these objects must not be deformed.
3. The interface *must* enable *any* of base object's roles to perform *any* remote COMBs; i.e. the provided interface needs to be *generic*.
4. The remote stub generated from this remote interface must be a valid remote reference to all bounded role instances which are played by remote base object. This is important to allow new role employment without reloading the remote base classes.
5. It must not prevent remote base objects to play *local* roles in other teams.

The remote interface provided by remote base objects is called the *Generic Remote Base Interface (GRBI)* [1]; as it provides access to remote base objects from any of their roles (see Figure 4.7(a)).

4.2.2 Remote Team Classes

With respect to a remote team instances, they should provide a remote interface that enables remote base objects to be bound seamlessly with their roles. The remote interface – called the *Generic Remote Teams Interface (GRTI)* [1] as shown in Figure 4.1(a) – must support the following:

1. When a remote CIMB is matched upon a base method call, the GRTI should enable the remote base object to invoke remotely the appropriate Team Level Wrapper Method (TLWM) associated to the role's callin method of that CIMB. In this regard, remote team instances must follow the disciplines mentioned in subsections 2.2.4.3 for a precise callin execution.
2. It must be *generic*; if a specific remote team instance declares several remote roles, then remote base objects must have a "single remote reference" to that team instance.
3. It should enable application developers to access remote team instances as distributed components.

Part (a) of the Figure 4.1 demonstrates that remote team and base classes could provide "*pre-prepared*" remote interfaces, and to accomplish a generic access to remote base objects and remote team instances, the Java Reflection API [5] could be used. The Java Reflection API is a standard Java library that enables application developer to invoke any of object's methods via a specific set of APIs. Therefore, these interfaces considered generic if the Java Reflection API is employed. This option facilitates a transparent communication between remote base and team objects at runtime without the knowledge of application developers. Though, application developers are obligated to access remote team instances at source code via the GRTI interface only.

To overcome this obligation, the use of user-defined interfaces as provided remote interfaces by remote teams must be supported. Therefore, the access of remote team instances becomes simple. For this purpose, the concept of "*Remote Façade*" is invented in this dissertation as a technique to enable application developers to *select* their own interfaces to access remote team instances as distributed components. This technique is used to access distributed components via interfaces in approaches like CORBA [69], EJB [46], etc.

For example, the remote team **MyTeam** can implement the user-defined interface **IMyTeam** as shown in part (b) of Figure 4.1; in which application developers can assign this interface as a provided remote interface. But, for **IMyTeam** to be a valid provided remote interface, it must *act* as a remote interface which simplifies the access of remote team instance as a distributed component, and facilitates the connection required in the RRP relationship (as shown in part (a)). To achieve this, the **IMyTeam** interface could be "*transparently*" redesigned to extend the GRTI as shown in part (b) of the figure. In this case, developers can explicitly access remote team instances of **MyTeam** via the **IMyTeam** interface and remote base objects of **MyBase** are *still* communicate implicitly with team instances via the GRTI.

4.3 The DOT/J Block Diagram

The block diagram shown in Figure 4.2 illustrates the global structure of DOT/J framework. It shows the integration between the DOT/J and the OT/J infrastructure. The figure divides the structure of DOT/J over three phases, which represent the stages of OT/J applications development; namely: the Compile-time phase, the Load-time phase, and the Runtime phase. Note that the incorporation between OT/J and DOT/J occurs at the Load-time phase; because the output of the Compile-time phase of OT/J will be reused in the mapping process. However, the following subsections will

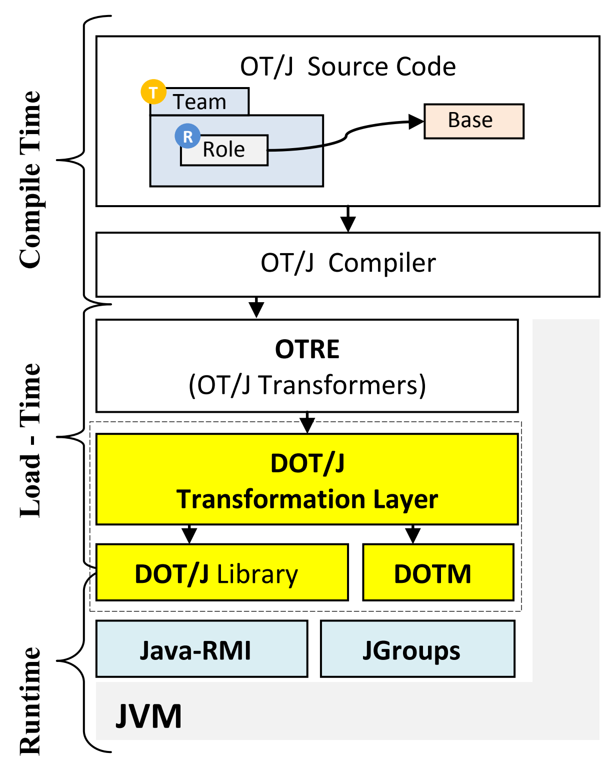

Figure 4.2. A diagram of the DOT/J Framework.

describe the conceptual DOT/J structure in detail[1].

4.3.1 Reusing OT/J's Infrastructure

This dissertation does not create a new model from scratch. Rather, it reuses the fundamental concepts of the OT model like role-playing and the activation mechanisms for teams, which have been already implemented in the OT/J programming language [79]. The main goal is to *map* these concepts and features and employ them in distributed environments in a systematic fashion.

Primarily, the OT/J's compiler will be reused to compile the remote team and role classes of OT/J application to be mapped. That is, to compile the declared *playedBy* relationships which connect remote base classes with remote role classes [3]. By reusing OT/J compiler, the base and role classes are preserved type-checked, besides the validity of the *playedBy* relationship. Thus, no new keywords or compiler directives need to be added to the OT/J language definition to realize the RRP concepts.

In specific, the mission of the proposed framework is to "*DOTJify*[2]" the OT/J application classes by mapping to a distributed environment according to the RRP criteria. In the next chapter, the concepts of this approach will be presented.

4.3.2 The DOT/J Transformation Layer

In general, the transformation of Java application classes is the process which allows *adding* new methods or fields, *removing* methods or fields, or *replacing* existing methods or fields of a specific Java class with new ones. The main goal of "class transformation" is to make application classes *conform* to the disciplines of a specific model. To map the OT/J application shown in Figure 3.1 – Chapter 3 to the distributed environment shown in Figure 3.2 (c) – Chapter 3, the base class **MyBase** must be *transformed* so that remote base objects are able to allocate and communicate with the remote team instance **t**. Likewise, the remote team class **MyTeam** and its role class **MyRole** need to be *transformed* to ensure a precise communication with remote base objects.

Most importantly in this regard, the OTRE comprises a set of bytecode transformers that use the load-time class transformation[3] in order to weave the defined roles into target base classes. In practice, transforming the bytecode of classes at the load-time provides a high degree of *transparency* with respect to injecting new code segments, and it enables the transformation of legacy Java applications. In addition, it makes it easy to evolve and maintain the OT model without affecting application classes.

The bytecode transformation of classes at the load-time needs to be adopted [1, and 3]; because the compiled and transformed classes of OT/J applications are reused. For this purpose, the diagram in Figure 4.2 illustrates the *DOT/J Transformation Layer (DTL)*. According to the figure, the remote classes of the OT/J application to be mapped must be transformed *locally* by the OT/J Transformers, and then by the DTL transformers.

[1] The implementation will be discussed in Chapter 4.
[2] We use this metaphor to indicate that we map OT/J applications into distributed environments by DOT/J.
[3] Other transformation approaches use the compile-time or the runtime to transform classes.

4.3.2.1 The Remote Base Class Transformer

The DTL comprises the Remote Base Bytecode Transformer (RBBT), which perform the following transformations to remote base classes:

1. *Prepare the remote base classes of a distributed application to generate remote base objects.* This task aims to provide remote base classes with a capability to enable the generated base objects to be *exposed* as *valid* remote players of roles. Apart from the technical details (which will be discussed in Chapter 4), it is important to ensure that the RBBT will *not* mistakenly transform remote base classes in such a way the generated remote objects are prevented to exhibit their original behavior or lose their original structure. This task mainly achieved by implementing the GRBI by *all* remote base classes of the distributed application.
2. *Equip remote base objects with a secure communication with remote team instances.* In case of RRP, the communication between remote base objects and remote team instances must be precise with respect to base objects' binding, and the carrying out of RRP relationship activities like remote CIMBs and remote COMB executions.
3. *Enable remote base objects to perceive the dynamic weaving of new roles at the appropriate execution time.* One of the fundamentals in the DADAO is the dynamic weaving of aspects. Thus, DOT/J is requested to support this fundamental as main part of RRP realization. This implies that application developers can attach new remote team instances dynamically, and the remote base objects of application are able to play the new roles of these teams. In practice, this means that the remote CIMBs defined in the new roles are able to intercept base methods' calls.

4.3.2.2 The Remote Team Class Transformer

Concerning remote team classes, the DTL includes a bytecode transformer called the Remote Team Bytecode Transformer (RTBT), which performs the following tasks:

1. *Prepare the remote team classes of distributed application to generate remote team instances.* When mapping a specific team class of OT/J application to distributed environments, it must be able to generate remote team instances that can *export* themselves as remote objects. This implies facilitating the communication between team instances and remote base objects in order to establish an accurate binding to role instances. In addition, remote team instances need to be accessed as distributed components, which could be accomplished through impelling all remote team classes to implement the GRTI.
2. Precisely bind *remote base objects* to *remote role instances* and *local* base objects to *local* role instances. That is, remote team instances must be able to work as *local* team instances if they enclose local roles besides remote roles. Therefore, the RTBT must not deform this feature. In fact, preserving this specification leads to mix local role-playing and remote role-playing, which supports a wide range of applications' modularity.

The RTBT is divided into three sub-transformers as shown in Figure 4.3:

1. The *DOT/J Team Transformer* is dedicated to transform the bytecode of remote team classes.
2. The *DOT/J Role Transformer* transforms the bytecode of the role classes which are played by remote base classes. The primary mission of this transformer is to make role instances compliant to the RRP, which implies, for example, enables roles to issue the different types of COMBs via the *remote references* of their remote base objects. Actually, the compiler of OT/J generates for each confined role class a "role interface" and an "implementation class" to implement that interface. The DOT/J Role Transformer transforms only roles' interfaces.

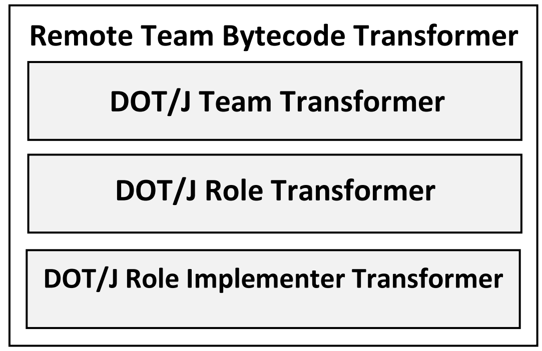

Figure 4.3. The Remote Team Bytecode Transformer and its sub-transformers.

3. The *DOT/J Role Implementer Transformer* transforms the "implementation classes" of roles' interfaces. In OT/J, when a base object has been lifted to a role instance for the first time, a new role instance is generated from the "implementation class" of that role, in which base object is passed as an argument to its constructor(s). Therefore, the implementation classes of remote roles must be transformed so that they *accept* binding *remote references* of base objects instead of local references. In Chapter 5, however, these transformers and the practical modifications they perform on the bytecode of remote base, role, and team classes will be revisited and discussed in more detail.

4.3.3 The DOT/J Library

The separation between remote team instances and remote base objects of a distributed application imposes a mechanism to help them to *allocate* each other *before* the remote *playedBy* relationship is established. That is, remote team instances need to *publish* themselves on a specific "*repository*", and remote base objects need to *look* them *up* at that repository in a proper way and "*at an accurate execution time*." To accomplish this, a specific component that could facilitate this *allocation-lookup strategy* is required, in which remote teams and remote bases can access that component. In the DOT/J framework, this is the responsibility of the *DOT/J Library* (see Figure 4.2).

4.3.3.1 The Publish-Lookup Strategy

The DOT/J Library provides model-specific support to remote objects at two levels:
1. Remote Objects' Conjunction.

At this level, a specific remote base object and a specific remote team instance need to find each other properly to establish a precise binding if that team holds a role of that base object. From remote team's side, a mechanism is needed to *publish* the necessary *contextual* information about the roles of team instance and its location. This information should simplify discovering team instances by remote base objects. In addition, the set of remote CIMBs declared in remote role classes need to be deployed on all hosts in order to allow the interception of base method calls by these CIMB; hence, accurate binding. Of course, issues of sequent order of remote teams' publication and remote CIMBs precedence must be taken into account [3].

Furthermore, remote base objects need to look remote team instances up at an early stage like their creation-time. Likewise, remote team instances need to publish themselves at an early stage of execution. Practically, in this *publish-lookup* strategy either a remote base object is created *before* the remote team instance publishes itself, or team instance publishes its information before the remote base object is created. In fact, this is an important issue because the order of publish and lookup operations determine the moment of remote role-playing. For example, in the first situation, the remote base object will miss playing the roles confined by the published team instance.

Let us explain both situations through a simple example. Consider the sequence diagram shown in Figure 4.4, which demonstrates a general concept for the *publish-lookup* strategy. In branch (a) of the figure, the remote team instance **t** of type **MyTeam** is created. It publishes its information in the "distributed layer" of the executing environment[1]. Let us call this layer as the *DOT/J Layer*. Afterward, the remote base instance **b** of base class type **MyBase** is created. This instance contacts the "DOT/J Layer" in order to look up *any* available remote team instance that might enclose any of their roles. Thus, the DOT/J Layer

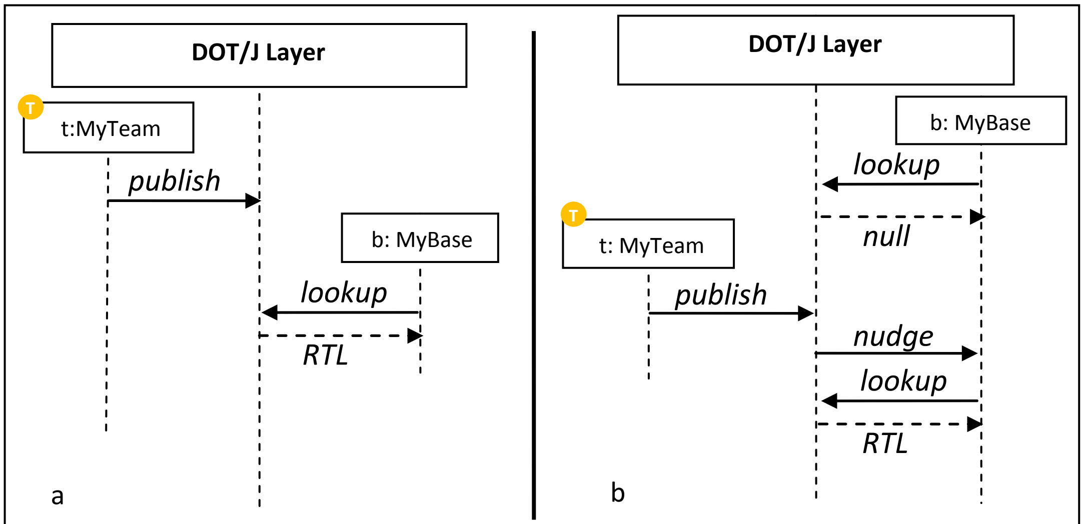

Figure 4.4. Scenarios of Publish-Lookup strategy: (a) the remote team instance **t** is published, then the remote base object looks it up, and (b) the remote base object requests the RTL before the remote team instance is published. The base instance is nudged to update the RTL.

returns the *Remote Teams List* (RTL). The returned RTL includes now information about the team instance **t**.

In this scenario, the remote base object **b** obtains the latest RTL. Conversely, in Figure 4.4 (b), the remote base instance **b** is created first. Immediately it issues a RTL look up request to the DOT/J Layer. In this case, an empty RTL will be returned as no remote team instances have been published yet. After a while, the team instance **t** is created and published. In this case, the remote base object **b** is miss-obtaining the accurate RTL; hence

[1] We proactively introduce the intermediation layer between remote base and remote team entities, which facilitates their communication. Later in this chapter and in Chapter 5 we will discuss the details of this layer.

will not play the role of team instance **t**. The base object **b** now holds a copy of the RTL which is different than the new RTL stored in the DOT/J Layer.

To ensure that remote base objects obtain the recent RTL, even if they were generated before remote team instances, a "*notification strategy*" could be used. In this strategy, the DOT/J Layer sends *fast* messages to the remote base objects affected by the publication of new remote team instances. For example, the DOT/J Layer sends a "*nudge*" to the remote base object **b** as shown in Figure 4.4 (b). Thus, the base object **b** requests the RTL once again. Now, it could obtain the recent RTL through one of two ways:

(1) embedding the RTL itself within the notification message (i.e. the DOT/J Layer sends the up-to-dated RTL as a notification), or
(2) The remote base object sends their requests to the DOT/J Layer once it receives nudged.

The second option enables a *lazy-RTL Update*, which allows the remote base object which receives notification messages to *complete* the execution of currently called method, and then contacts the DOT/J Layer to obtain the new RTL. In this way, new remote team instances could be employed without worrying about remote base objects' consistency. The first option, on the other hand, delivers the RTL to remote base objects *faster* than the second option, but adds extra overload on the communication between the DOT/J Layer and remote base objects if the RTL is large.

4.3.3.2 The Remote Teams' Registration

To achieve a precise publish-lookup process, the *Remote Team Registration Strategy* is adopted, which stipulates: "all the remote team instances of a specific application *must* register themselves in the DOT/J Layer at their creation-time". This strategy imposes, necessarily, the use of an "intermediate repository" which can store, organize, and manage the registered team instances. For this purpose, the *Distributed Objects and Teams Manager (DOTM)* component (see Figure 4.2) has been developed in the DOT/J framework to carry out this mission (will be discussed in the following subsection).

To accomplish the registration process, remote team instances should provide a bundle of information to the DOTM, which includes:

a. The contextual information about the remote CIMBs declared by roles of the registering team instance.
b. The information about the "entity" of remote team instance like its class name, and the remote stub generated from the provided interface.

2. The Remote Objects' Communication level.

This level represents the actual RRP activities. Most importantly, the declared remote CIMBs of remote roles must be *matched* at the accurate execution time by remote base objects, and the associated remote callin methods must be invoked according to the accurate precedence.

With the used publish-lookup strategy – in addition to the notification mechanism, it could be guaranteed that remote base objects can obtain the recent RTL version all the time. Consequently, calls of their methods are intercepted accurately, which means that they can play roles accurately.

From role instances' side, a precise binding must be accomplished in order to ensure that these roles have obtained *correct* references to their base objects. Therefore, they can issue remote COMBs. To achieve this, remote team instances must verify that the remote base object under binding is a valid player. The validity of a specific remote base object is subjected to two main issues: (1) base object must be generated from the *same* base class that was used by the OT/J compiler to compile the *local playedBy* relationship, and (2) remote base object must obtain a reference to the team instance from the DOTM component *only*. For example, the remote base objects deployed on hosts H1, H2, and H2 in Figure 3.2 (c) – Chapter 3 must be derived from the same base class **MyBase** which is used to compile the “playedBy” relationship in remote team class **MyTeam**. Otherwise, the binding should be refused due to base objects’ incompatibility. At runtime, base objects should obtain a remote reference to the remote team instance **t** via the DOTM only.

In fact, the communication between remote base objects and remote team instances of a distributed application must be “*sponsored*” by the DOT/J Layer in order to ensure the application integrity.

4.3.4 The Distributed Objects and Teams Manager - DOTM

The DOTM component works as a *broker* for the remote base objects and the remote team instances at runtime. Figure 3.1 – Chapter 3 has been redrawn, as shown in Figure 4.5, to illustrate the relationship between the three components; remote base objects, remote team instances, and the DOTM. In practice, the DOTM component is implemented to perform the following tasks:

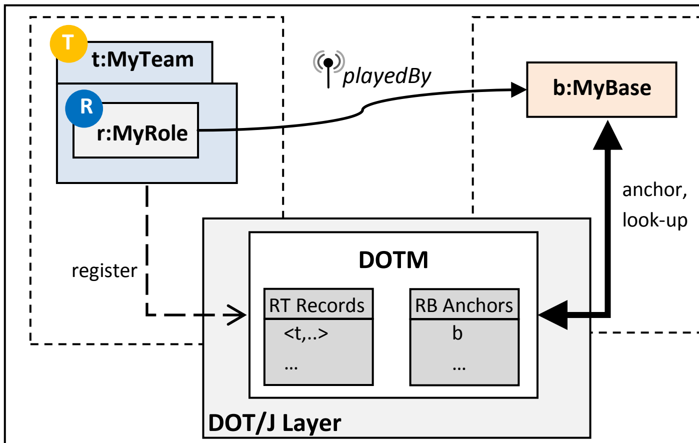

Figure 4.5. The DOTM works as a broker for team and base objects.

1. *It registers the remote team instances of the distributed application*. The DOTM is the repository of remote team instances, and *all* remote team instances must register in the DOTM. Specifically, when a new remote team instance is being created, it sends a *Registration Request* to the DOTM. Consequently, the DOTM creates a new *Remote Team-Record* in the RT-Records pool (see Figure 4.5).
2. *It stores the anchors of remote bases*. In DOT/J, the *Remote Base Anchor (RBA)* concept is invented to support a communication channel between remote base objects and the DOTM. Specifically, when a remote base object is being created, it contacts the DOTM and *registers* its RBA. This anchor will be used by the DOTM to notify the base objects of a specific remote base class with the changes to the RTL in case new remote team instances are deployed. The DOTM should notify those base objects which are affected by teams’ deployment; i.e. those objects which have playable roles in the registered remote team instances.
3. *Preserves the RTL consistent and up-to-date*. Certainly, the registration of new remote team instances leads to RTL inconsistency at some (or all) application nodes. This implies that some remote base objects hold incorrect RTL versions. Therefore, it is the responsibility of the DOTM to *re-calculate* and deliver the new RTL for each one of the

affected remote base classes. This implies that the DOTM should be able to prepare RTL for each remote base class has its BRA registered.

4. *It should provide application developers (at runtime) with information about remote team instances* like their deployment and activation status.

4.3.4.1 Detection Unavailability of Remote Teams

The DOTM is responsible for the RTL consistency in case of remote teams' *unavailability*. The unavailability of a specific remote team instance occurs when a remote base object attempts to play a role of that team instance which has no longer existed (e.g. due to node crash or runtime exception). Actually, this issue involves several aspects; The unavailability of remote team instances could be detected either during the RRP process when a remote base object attempts to invoke a specific TLWM on the unavailable team instance, or when that team instance has been accessed (as a distributed component) and a runtime exception occurs. Figure 4.6 illustrates the first situation in which the remote team instance **t** of type **MyTeam** registers in the DOTM. Suppose that the role class **MyRole** has declared a specific CIMB to intercept calls of method `m` of the remote base class **MyBase**. After it has been created, the base object **b** obtains the RTL from the DOTM, which comprises **t**. Then, for some reason, the execution of the team instance **t** is interrupted. After that the method `m` is called on **b** and intercepted by the CIMB of **r**. As shown in the figure, the base object **b** will detect the unavailability of **t** at the moment it attempts to call the TLWM on **t**.

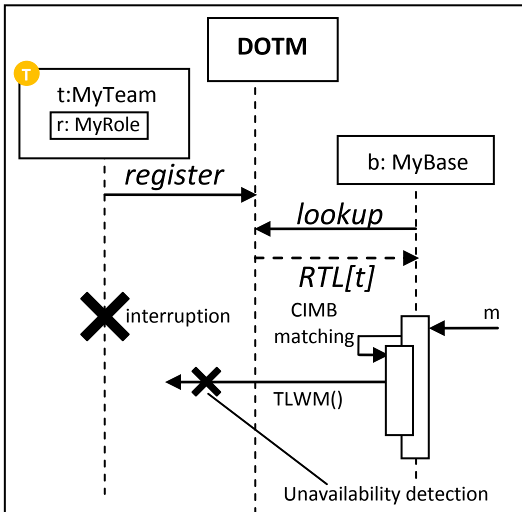

Figure 4.6. Unavailability detection of the remote team instance **t**.

The unavailability of remote team instances is critical and might impact the entire RRP process; because the RTL becomes inconsistent. Therefore, remote base objects and the DOTM should work harmoniously to keep track of remote teams' availability. This implies that remote base objects should inform the DOTM with any unavailability detection like the one shown in Figure 4.6, and DOTM should take the suitable process against that event.

Moreover, the mechanism the DOTM uses to organize the registered remote team instances must regard the order of their registration. Also, it should store as minimum sufficient remote teams' information as possible to safe space, and to provide quick notification and fast RTL delivery to remote base objects.

4.3.4.2 The DOTM Design Specifications

Remote team and base objects of a distributed application need to communicate with the DOTM at their *creation-time*. In DOT/J framework, to ensure that all application nodes are connected, a DOTM component should be deployed on each of these nodes. Therefore, remote objects can quickly grip a handle to the DOTM instance and register accordingly. The deployed DOTMs, then, *intercommunicate* with each other to preserve the RTL identical at each node *all the time*. Actually, deploying DOTM on all application's nodes adds more flexibility to the registration process of remote team instances and remote base anchors, and decentralizes the management of the registered team and base objects. Furthermore, it enables a quick RTL recovery, for example, in case that a specific node is restarted.

An alternative option to implement the DOTM, however, is to use a *single centralized DOTM*. Thus, all remote team and base instances register in a single repository. The implementation of this technique, which has been adopted in approaches like DJcutter [55], is simpler than deploying individual DOTM on all nodes; because no efforts need to be paid to establish a protocol of intercommunication between several DOTMs. Nevertheless, the choice of central DOTM has several disadvantages; *first*, it forms a single point of failure, i.e. if the node on which the central DOTM operates has been turned off or crashed, then the whole RRP process will be stopped. *Second*, it imposes the use of a fixed DOTM location, which decreases the transparency of the approach, and restricts the dynamicity and expandability of distributed application. In common, a communication overload on central DOTM can result in an application crash (due to congestion) if a large number of remote objects have been registered and a dense message transmission over the network has been encountered.

In practice, the DOTM component allows the dynamic attachment of new remote team instances at runtime, which helps to achieve a dynamic adaptation of distributed applications objects; because new roles will be played dynamically at runtime without interrupting application execution.

4.4 A Concept to execute Remote CIMBs, COMBs, and BGPs

This section discusses the disciplines followed in OT/J to realize a transparent interception of base method calls via CIMB, and then it shows how to employ the same mechanism in case of RRP. After that, it presents a concept for performing remote CIMBs and COMBs, and evaluating remote BGPs.

4.4.1 A Transparent Interception of Calls of Remote Base Methods

Basically, the following facts about intercepting the calls of base objects' methods by the declared CIMBs in an OT/J application have been enumerated:

1. The interception process is performed *transparently*.
2. Roles are woven at the load-time into the bytecode of base classes. Therefore, they are able to respond to interception and execute the associated role's callin methods.
3. The CIMBs' precedence is resolved by the OTRE transformers at the load-time during the bytecode transformation of base and team classes.
4. Roles are played only when team instances are activated.

To achieve interception "transparency" in case of RRP, remote base objects need to *detect* remote CIMB matching and *respond* to that matching transparently. Actually, due to the loose coupling between remote team and remote base objects, allocating remote team instances by remote base objects could be performed only at a late phase of application execution. This means that remote roles need be woven at runtime and not at load-time. For this reason, a list of remote team instances that are found activated at the *call-time* of a method on a specific remote base object must be calculated. Then, the matched remote

CIMBs declared by the roles of these teams should be handled accurately. For example, before the interception of a call to method `m` is carried out as shown in Figure 4.6, the remote base object **b** must recognize whether the remote team instance **t** is currently activated or not. If **t** is found deactivated at the moment of interception, then the interception should not be handled.

4.4.2 Executing the Remote CIMBs – Remote Advice Execution

The phrase "remote CIMB execution" refers to the process at runtime which includes the matching of the CIMB, and the execution of the role-level method associated with that CIMB. The DOT/J framework is requested to ensure a safe CIMB execution especially in case of multiple CIMBs matching, i.e. when method call of a specific remote base object is intercepted by several remote CIMBs declared by different roles.

For a basic interception case, consider the UML sequence diagram shown in Figure 4.7. The diagram demonstrates a simple remote CIMB execution. The CIMB is a replacement callin binding, which binds the method `m` of base object **b** (of the base class type **MyBase**) to the callin method `rm` of role instance **r** (of role class type **MyRole**). To add more explanation, the provided interface of the remote team instance **t** (i.e. the GRTI) has been interpolated in the figure.

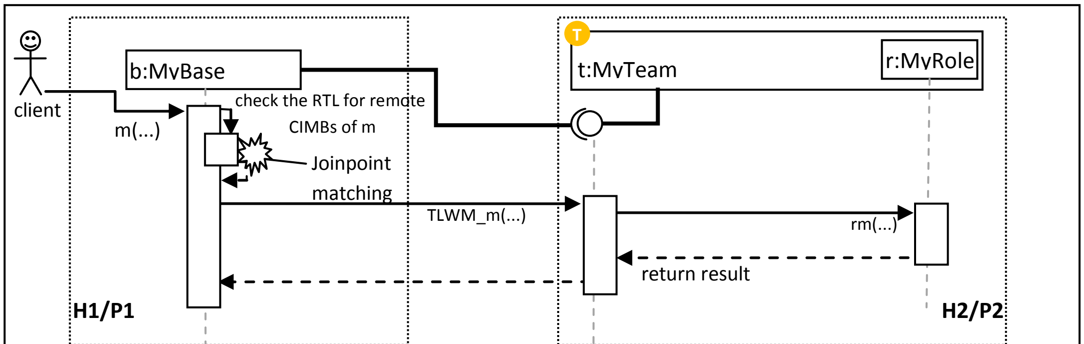

Figure 4.7. A UML sequence diagram depicts a simple remote CIMB execution.

As the figure shows, when the client invokes base method `m`, a remote CIMB is detected (because **b** has obtained the RTL just prior this detection). Thus, **b** can determine which of remote team instances are activated at the moment `m` has been invoked. Therefore, a list of CIMBs for base method `m` is prepared. Then, **b** communicates with the *activated* remote team instance **t** via the provided GRTI interface. The base object **b** invokes the TLWM of **t** (named for simplicity as `TLWM_m(…)`), which invokes (in turn) the role-level method `rm()` and returns the result. When calling the method `TLWM_m`, base object **b** *dispatches* its own *remote stub* along with the invocation; thus, the team instance **t** is able to *bind* that *remote stub* to the role instance **r**.

Synchronized Execution

To preserve the consistency of remote base objects in case of multiple method-call interceptions, the intercepted base method should *block* base object, i.e. prevents access to that object until all interceptions are performed. This includes local access as well. Then, the base object must be unblocked once the control flow is reclaimed from the remote team instances involved in the current “*interception chain*.”

4.4.3 A Conception for Executing Remote COMBs

When an expected role-level method of a specific remote COMB is invoked, the following conditions concerning the target remote base object must be satisfied:

1. If the remote COMB is a field setter, then it must be invoked *synchronously* in order to prevent concurrent access to the same base’s field.
2. If the remote COMB is a base method call, then the remote base method bounded in remote COMB must be invoked *synchronously* to prevent invoking the base method concurrently by other remote method-COMBs or local calls.

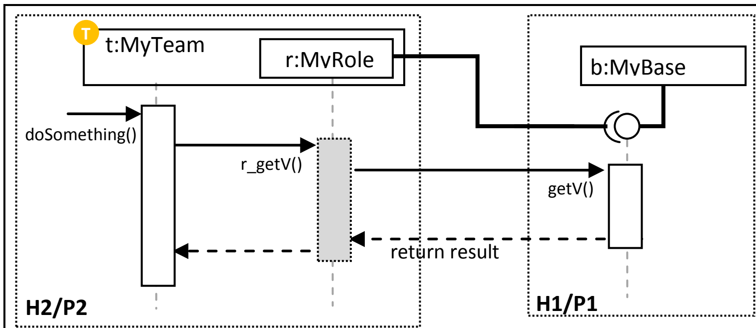

Figure 4.8. A UML sequence diagram depicts a simple remote COMB example.

To explain how a simple remote method-COMB could be executed, consider the example shown in Figure 4.8, which represents the execution of code snippet shown in Listing 4.1. As the UML sequence diagram illustrates, the control flow enters first the boundaries of the remote team instance **t** in order to invoke the method `doSomething()`. This method forwards the control flow to the role instance **r** (which has been *lifted* explicitly as shown in Listing 4.1 at line 1) to invoke the expected method `r_getV()` (drawn as a gray rectangle in Figure 4.8). At this point, instance **r** is *lowered transparently* to its base player (which is the remote stub of base object **b**) and invokes the method `getV()` *remotely*. The remote base object **b** handles the invocation locally, and returns the result of invocation to **r**.

Listing 4.1: A simple remote COMB invocation

```
   // inside remote MyTeam class
1. void doSomething(MyBase as MyRole r){
2.     r.r_getV();
3. }

4. class MyRole playedBy MyBase { // in Role class
5.    Type r_getV() -> Type getV(); // COMB
6. }
```

4.4.4 A Conception for Evaluating Remote BGP expressions

To *control* the effects of remote CIMBs by using remote BGP expressions, these BGPs must be evaluated precisely. For example, if a remote BGP has been declared along a specific remote *playedBy* relationship, then it must be evaluated *before* the team instance binds remote base objects to role instance. In addition, these expressions must be handled transparently. Nevertheless, the evaluation of remote BGP expressions is a straightforward process because remote base field access and base method invocations (involved in the remote BGP expression) could be *considered* as remote COMBs.

Let us take a simple example to explain how a remote BGP could be evaluated. Suppose that the following remote BGP expression has been declared along a remote *playedBy* relationship:

```
base when (base.x > 50 && base.getID() != this.normalID)        (4.1)
```

Then, the value of field `x` of the remote base object under interception (or under binding) must be fetched and the method `getID()` invoked. In this example, two things must be considered: (1) the keyword **`base`** appeared in expression (4.1) should be replaced by the remote reference of the base object under binding, and (2) getting the value of `x` and invoking the `getID()` method must be handled via remote COMBs to satisfy the conditions mentioned in subsection 4.4.3.

4.5 Avoiding the Problem of Remote Base Class Versioning

It could happen that application developers *unintentionally* implement two different versions of the same remote base class. Consequently, different remote base objects might be deployed on different application hosts. Suppose that two remote base objects have been generated each from one of these remote base versions, and then bound to two role instances in a specific remote team instance. In this case, a consistency problem is occurred, for example, when a remote method-COMB has been invoked by the roles played by these base objects; in which the associated base method might be implemented in different ways. Thus, different results may return to the caller roles.

4.5.1 An Example

To illustrate the problem, consider the example shown in Figure 4.9. In part (I) of the figure,

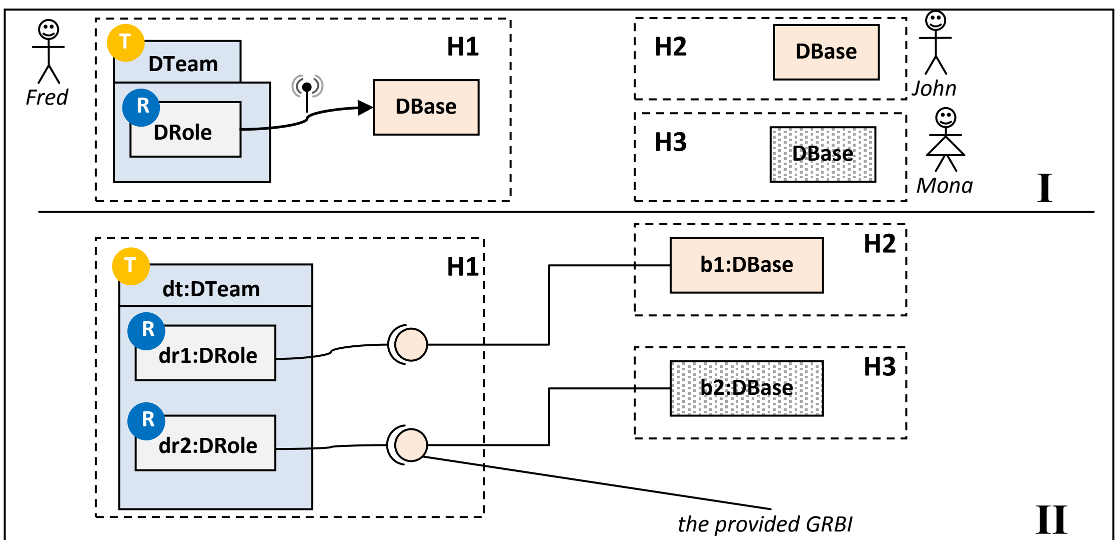

Figure 4.9. The problem of Remote Base Class Versioning.

the application programmer "Fred" has implemented the remote team class **DTeam** at host H1. This team confines the remote role **DRole**, which is played remotely by the base class **DBase**. At the host H2, "John" implements the base class **DBase**, while "Mona" implements the same base class at host H3 *but* in different implementation, e.g. a specific method has been implemented with extra parameters, or different class fields were added.

At runtime (part II of the figure), the remote team instance **dt** binds the two role instances **dr1** and **dr2** respectively with the remote stubs of the remote base objects **b1** and **b2**. At host H1, the team instance **dt** cannot recognize that **b1** and **b2** are generated from different base classes; because **dr1** and **dr2** are actually bounded with the remote stubs and not the actual base objects. In fact, the remote stubs of **b1** and **b2** will be generated from the same GRTI interface; which is provided by all remote base objects. Therefore, **dt** must has the capability to detect this incompatibility before it binds base objects to its roles.

This problem (called as *Remote Base Class Versioning (RBCV)*) is similar to the famous *Java Class Serialization* problem, which occurs when a specific object generated from a specific class version is being serialized (e.g. in case of persistency to a system file, or transferring over the network), and then reclaimed using a new class version. In the RRP context, this problem is more dangerous and could impact the RRP harmony; because remote team instances *deals* with the remote stubs of base objects. Furthermore, two different implementations of the same remote base class will generate remote stubs from the same GRBI (as shown in Figure 4.9), which makes the problem worse; as it should be carried out transparently.

4.5.2 The Class Serialization Identification

To avoid the problem of RBCV, a remote team instance must bind only remote base objects that are generated from the base class that has been *used* by the OT/J compiler to compile the *playedBy* relationship. Therefore, the remote team instance **dt** in the example of Figure 4.9 should be able to *verify* that the remote stubs bound to **dr1** and **dr2** belong to remote base objects generated from the same remote base class **DBase** which is declared in the *playedBy* relationship at H1; as shown in part (I) of the figure.

To solve this problem, a Serialization Identification strategy similar to the Java Serialization could be used. In simple words, the remote base classes at H1, H2 and H3 in Figure 4.9 (I) *must* have identical *Class Serialization ID* (CSID) values. A CSID could be a number or string to indicate the version of the class description.

At the transformation-time of classes, the remote team class **DTeam** could be provided (via the RTBT) by the CSID of base class **DBase**, which could be generated dynamically from the class description of **DBase** at H1. Thus, **dt** *can* use the generated CSIDs to check the base class version of **b1** and **b2** objects. The Remote Base Bytecode Transformer (RBBT), on the other hand, is able to generate and *inject* into the remote base class **DBase** of H2 and H3 the same CSID value. Then, the validity of remote base objects is satisfied if the two CSID values are identical; the one generated in the team **DTeam** and the one generated in each **DBase** class.

4.6 The Dynamic Deployment of Remote Teams

The natural separation between the remote team and remote base objects of distributed applications, and the adoption of the remote *provided-required* interface design, allows application developers to implement and create new remote team instances dynamically at runtime without stopping application execution or worrying about application consistency. This feature is equivalent to the *dynamic aspects weaving* strategy used in dynamic-AOP approaches like AWED [47], PROSE [9], DandyJ [51] and DyReS [21]. The dynamic attachment of new remote team instances and employ new roles at runtime is better[1] than aspects weaving and unweaving dynamically at runtime for the following reasons:

1. The performance overload required to weave/withdraw aspects into/from specific base objects (like in PROSE and DandyJ) and then reloading the modified classes is saved when RRP is applied.
2. The atomic weaving/unweaving of aspects into/from multiple distributed base objects is not required when multiple base objects play their roles inside teams; i.e. no code transmission is required and no class reloading as well.
3. The DADAO requires adaptation of objects' behavior to be controllable. Current distributed AOP approaches do not provide a suitable mechanism like remote teams' activation/deactivation, which enables application developers to control the effects of roles' callins without worrying about remote base objects consistency.

4.6.1 An Example

To illustrate the idea of dynamic deployment and employment of remote team instances, consider the simple example shown in Figure 4.10. The figure shows in part (a) a RRP relationship between the remote base object **b** and role instance **r**, which is enclosed in the remote team instance **t**. Note the remote provided-required connection between **b** and **t**. In part (b), a new remote team class (**ARTeam**) has been implemented and a new instance **q** is created. Once it is created, the team instance **q** registers itself immediately in the DOTM.

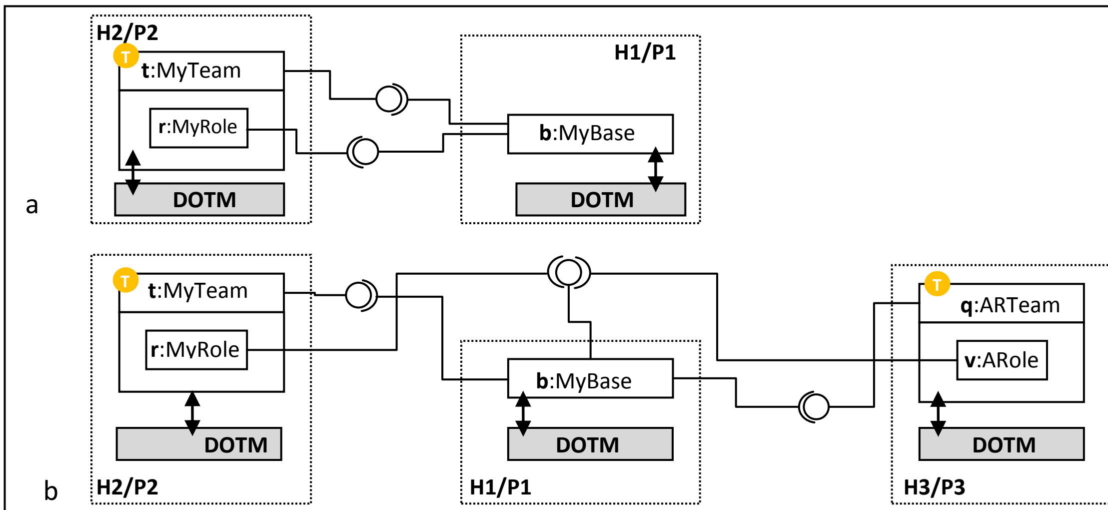

Figure 4.10. A typical single remote role-playing shown in (a), and multiple remote role-playing in (b) after the remote team instance **q** has been attached dynamically.

[1] This claim is strengthen by the results of experiments made with the RRP approach; see Chapter 6 for more details.

As the figure shows, the **ARTeam** class confines the role class **ARole** which is played by the remote base class **MyBase**. Therefore, **b** needs to play this new role. The question is: "*How can **b** play this role?*" To answer this question the following sub-questions must be answered too:

1. Which of remote team instances **t** and **q** owns the *highest priority* over the other so that its confined role played first?
2. How to *calculate* the accurate remote CIMBs interceptions map if **MyRole** and **ARole** have declared remote CIMBs to intercept the same method in **MyBase** base class?
3. When should the remote base object **b** play the new role **ARole**?

4.6.2 The Activation Priority Values

The DOT/J framework is requested to provide an adequate mechanism to organize the remote team instances registered in the DOTM. This mechanism should use a specific *precedence criterion* to resolve the priority of multiple role-playing scenarios as the one depicted in part (b) of Figure 4.10. Actually, if the remote team instance **q** has been created and registered while the remote base object **b** is playing the role **r** (i.e. the team instance **t** is currently activated), then two possible options to determine which of team instances (**t** and **q**) has the highest priority could be considered.

The *first* option is to consider the *latest* remote team instance to *register* in the DOTM is the one with the highest "activation priority." The *second* option, on the other hand, is to *force* all remote team instances to provide a specific value when they register at the DOTM, which indicates their activation priority. Practically, the first option can result in an unexpected application behavior because there is no guarantee that all remote team instances will register at the desired order. For example, suppose that the team instance **q** is desired to obtain higher priority over **t** but for a technical problem team **t** registers too late after **q** has been registered. In this case, **t** will "win" and obtain the highest priority. The second option, on the other hand, is much more practical as it enables application programmers to control the dynamic employment of remote team instances at runtime. Though, it affects the transparency of the DOT/J framework; because application developers need to assign a priority value for each remote team class explicitly.

Now, when a remote base object is notified by the DOTM due to a registration of a new remote team instance, then the base object uses the priority value assigned to that team instance and re-organizes the Remote Teams List (RTL) it owns. Consequently, the remote base object **b** plays the role **v** first if it has found team **q** was assigned a larger activation priority value than team instance **t**. In addition, if both **r** and **v** declare replacement CIMBs to bind the same base method, then the control flow should be dispatched from role **v** to role **r** precisely if **v**'s replacement callin method issues a base-call to that base method.

4.7 The Remote Team Activation/Deactivation: Types, Order and Priority

When a specific remote base object plays multiple roles in several remote team instances, then it needs to know the order according which these team instances have been activated. This is important for base objects to handle the multiple CIMBs matching accurately.

The use of *Remote Team Activation Priority* (RTAP) values helps remote base objects to resolve precedence of activation of currently activated team instances; thus, an accurate roles' callin methods execution will be ensured. The RTAP should be assigned at an early stage before application classes are loaded for execution, and need to be known at remote CIMBs matching. Thus, remote team instances may provide this value to the DOTM at the registration process.

As distributed application base objects and team instance are separated, remote team instances must be activated and deactivated *globally*; so that remote base objects could read the activation status precisely. In fact, the global activation of remote team instances is compulsory in case of RRP; because Java objects (including remote objects) are executing within local "threads" that cannot be serialized over the network. The global activation of teams is supported in OT/J via activation APIs. Therefore, the DOT/J framework could reuse the same facility in case of RRP.

4.8 Summary

This chapter has presented the concepts of Remote Role-Playing (RRP) and the conceptual specifications of the DOT/J framework. The DOT/J framework is the approach to implement the RRP concepts. The chapter has demonstrated how the features of the OT model could be mapped to a distributed environment on top of the Java-RMI middleware. The terms of *remote team*, *remote role*, and *remote base* have been defined. The mechanism to realize each of these terms in distributed environments has been discussed in concept. The next chapter will describe in detail the implementation of the DOT/J framework.

Chapter 5

DOT/J Framework Implementation

This chapter presents the detailed implementation of the DOT/J framework. It describes the implementation of DOT/J infrastructure according to the discussion in Chapters 3 and 4. The chapter presents a set of examples that strengthen the explanation of the implementation process.

5.1 Introduction

Chapter 4 has discussed the concepts of the DOT/J framework over three phases of OT/J applications development. In specific, *the static phase*; i.e. the application before it has been loaded into the JVM. At this phase, application developers are requested to *mark* the base and team classes which are involved in the RRP relationships in order to map them to the DOT/J framework. *The load-time* phase *transforms* the marked classes so that their objects can enroll precisely in RRP relationships. Finally, Chapter 4 has introduced how should remote base objects and remote team instances be supported at *the runtime phase*; i.e. after they are connected together. This chapter presents an elaborated explanation for the implementation of each phase in the DOT/J framework.

To simplify the discussion of implementation, the distributed application discussed in Chapter 3 will be reused as an example. The discussion will gradually emphasize the implementation specifications of the DOT/J framework as the implementation of this application will be described as if it was a DOT/J application.

Figure 5.1 depicts a thorough conception for the mapping of that application to a distributed environment. Starting from the top-left part, the figure demonstrates that the source code of the application (drawn as a miniature diagram to the Figure 3.1) must be *compiled* by the OT/J compiler. The compiler compiles mainly the local *playedBy* relationship and the team class. Consequently, it generates the role's interface and its implementation class. Therefore, from this stage – named as *Compilation-time* stage – the Java bytecode classes of **MyTeam** and **MyRole** classes (i.e. the Java files with the ".class" extension) are produced. As shown in that part of the figure, DOT/J has no contributions at this stage.

Afterward, application ".class" files are loaded into the JVM for execution; as shown in the "Load-time phase". Recall that the base class **MyBase** and the team class **MyTeam** have been compiled as entities of a single OT/J application. But in case of a distributed application they might be loaded into different JVMs. Anyway, both of them will pass the OTRE and DOT/J Transformation Layer (DTL) transformers. The DTL library, as shown in the figure, receives the transformed bytecode of base and team classes from the OTRE transformers[1]. By now, as the transformed team, role, and base classes are *taught* to establish local *playedBy* relationships only, the mission of the DTL is to equip the **MyBase** and **MyTeam** classes *with* new code segments that realize the remote playedBy relationship between the distributed objects of these classes.

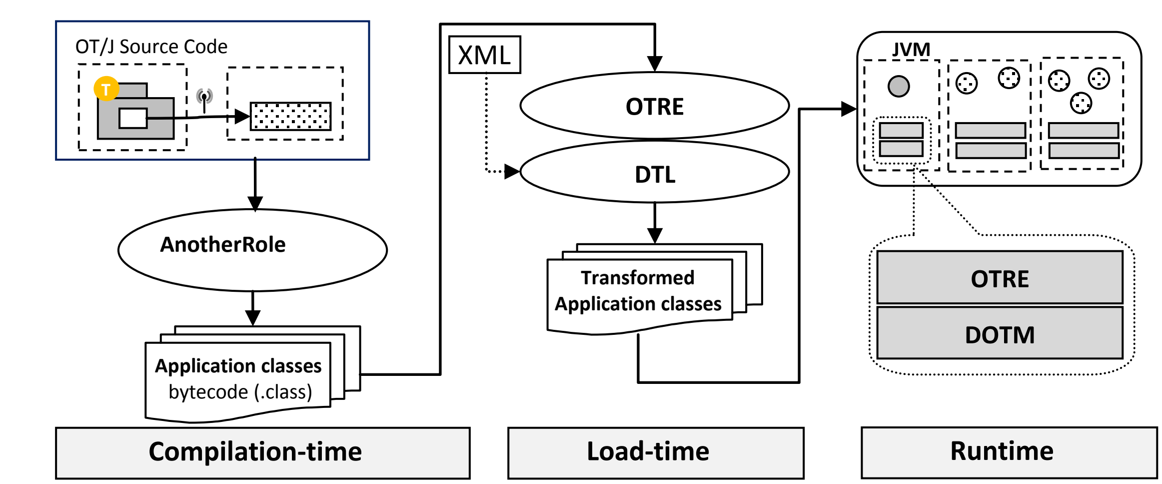

Figure 5.1. General application development process using the DOT/J framework.

Note that not all base or team classes of a specific OT/J application need to be mapped to the distributed environment. For example, **MyTeam** could comprise local roles beside the role class **MyRole**. Likewise, **MyBase** might not be the only class in the application. Therefore, a mechanism is required to *recognize* which of application classes requires a DTL transformation. To fulfill this requirement, application developers could provide a *labeling map* that enumerates those base and team classes (including the remote role classes) – which are involved in RRPs – before they are actually loaded into the JVMs. For this purpose, a simple XML-based labeling language has been developed (will be discussed in the following section). This language aims at enabling application developers to organize the remote classes of an application for the DTL transformation using simple and expressive XML entries. The developer prepares XML files, and then supplies them – as shown in the middle part of the figure – at the load-time phase to the DTL.

After the class transformation process is completed, remote team and remote base classes are loaded into the JVMs as shown in the last part of the figure (this phase called as the *Runtime Phase*). Squares with dashed borders represent application hosts, and indicate that the objects enclosed inside each one of them (shown as filled circles) reside at different hosts (hence, different JVMs). The circles filled with dots represent remote base objects, and the circle filled with solid gray represents a remote team instance. For simplicity, both the JVMs and the generated objects are drawn inside a single *runtime scope*; whereas at each host different JVM and different objects could be installed.

In order to perform a precise RRP, a DOTM component is running at each node underneath remote objects as the figure illustrates. Practically, the DOTM component must

[1] Each one of them is actually transformed in different host/process.

be deployed and operated *before* any remote base or remote team objects is created; in order to ensure an accurate binding between them as will be explained later.

The following sections will explain in detail the XML-based language, and discuss the requirements of remote base and remote team classes to generate *valid* remote objects from the *provided-required remote interfaces*. After then, they will discuss in detail the load-time transformation of remote team and remote base classes, and demonstrate how to conform them to the DOT/J criteria for RRP.

5.2 Mapping the Remote Classes of Applications to the DOT/J Framework

Labeling remote base and team classes aims at announcing which of these classes is involved in a RRP relationship. The process is *not* to map individual remote base methods to remote roles' callin methods as adopted in many distributed-AOP approaches like DyReS [21]. Also, the labeling process does not aim at binding remote base and role classes. Rather, it only maps remote classes to the DOT/J framework. This enables remote base objects to play *new* roles dynamically at runtime instead of weaving roles at the load-time as adopted by OT/J. In practice, XML can adequately capture the nesting structures of team and role classes through nested XML entries, which makes the file readable and expressive. In addition, it simplifies attaching properties to the entries in clear fashion.

5.2.1 The Structure of XML Labeling Files

Basically, application developers need to create and deploy a single XML labeling file on every application node on which RRP should be established. This file must be stored in a visible location to the DTL (e.g. within the application class-path). At each node, this file should comprise a list of entries each maps exactly one of the application's remote classes.

The XML labeling file must enclose all entries between the tags of the *root* node `<DOTJ>…</DOTJ>` as follows:

```
<?xml version="1.0"?>
<DOTJ>
    // the labeling entries ..
</DOTJ>
```

At the load-time, the DTL expects an XML file named as "`dotjConfig.xml`" in the application workspace[1]. If no XML file detected, then the DTL will suppose that no remote classes have been implemented on the current node.

5.2.2 Labeling of Remote Team Classes

Labeling remote team classes must regard the characteristic of team as a container of roles. Therefore, the XML entry which labels a remote team class should clearly express the team class and the remote roles enclosed in that team in a similar nesting hierarchy. To illustrate this idea, the example shown in Figure 5.2 – a modified version of Figure 3.1 in Chapter 3 – will be used. In the new version of the figure, the remote role **AnotherRole** and the remote

[1] This specification is adopted in the current implementation of DOT/J framewok.

base class **AnotherBase** have been added. Another modification is that the team class **MyTeam** now implements the remote façade **IMyTeamFacade**.

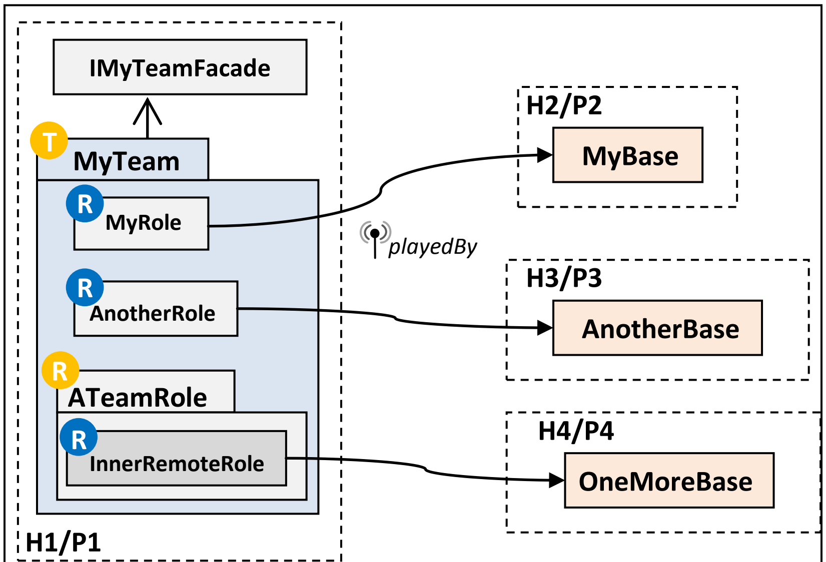

Figure 5.2. A distributed application with multiple remote role-playing.

Inside the XML file, *any* remote team class must be labeled as shown in the XML code below. The **`class`** property refers to the full-qualified name of the labeled team class, which is composed of package name of that team, and the simple team class name. For example, if the team class **MyTeam** was implemented in the package `com.business.classes`, then the remote team node in XML file will be:

```
<?xml version="1.0"?>
<DOTJ>
       <RemoteTeam class=“com.business.classes.MyTeam“>
        :
       </RemoteTeam>
</DOTJ>
```

Each remote-team-node `RemoteTeam` maps exactly one specific team class. Thus, to label another remote team, a new separate node must be created. The `RemoteTeam` node encloses the names of the remotely played roles that are confined by the labeled team.

5.2.2.1 Enumerating the Remote Roles of a Remote Team

To label the remote roles enclosed by a specific remote team class, the `RemoteRole` node is used. It takes the attribute `name` which points out to the simple class name of the labeled role. For example, the remote roles of the **MyTeam** team class of Figure 5.2 are labeled as follows:

```
<?xml version="1.0"?>
<DOTJ>
       <RemoteTeam class=“com.business.classes.MyTeam“>
              <RemoteRole name=”MyRole”/>
              <RemoteRole name=”AnotherRole”/>
       </RemoteTeam>
</DOTJ>
```

The DTL can calculate the full name of a remote role from the full-qualified name of team class and the simple role name in the attribute `name`. In case that the remote role is a team (i.e. a team-role), then it must be labeled as a normal remote role using the node `RemoteRole`. However, if the role to be labeled is a nested remote team (i.e. a remote team-role) that may enclose remote roles, then it must be labeled with the node

`RemoteTeamRole`. In this case, if the outer team class encloses another remote role beside this team-role class like **ATeamRole** as shown in Figure 5.2, then the inner remote team-role class **ATeamRole** must be labeled inside the outer team node `RemoteTeam` as follows:

```
<RemoteTeam class="com.business.classes.MyTeam">
      <RemoteRole name="MyRole"/>
      <RemoteRole name="AnotherRole"/>
      <RemoteTeamRole name="ATeamRole">
            <RemoteRole name="InnerRemoteRole"/>
      </RemoteTeamRole>
</RemoteTeam>
```

If the outer team class encloses local roles and remote team-roles only (i.e. it does not enclose remote roles), then each confined remote team-role class must be labeled in a separate `RemoteTeamRole` node as shown below. The attribute "`in`" of node `RemoteTeamRole` refers to the full-qualified name of the outer team class.

```
<RemoteTeamRole name="InnerTeamRole" in="com.business.OuterTeam">
      <RemoteRole name="InnerRemoteRole"/>
</RemoteTeamRole>
```

5.2.2.2 Labeling of the provided Remote Façade

If a remote team class provides a specific remote façade, then application developers need to explicitly label that interface inside the remote team node `RemoteTeam`. For example, as shown in Figure 5.2, the remote team **MyTeam** implements the remote façade `com.business.interfaces.IMyTeamFacade`. Thus, the labeling node of **MyTeam** will be as follows:

```
<?xml version="1.0"?>
<DOTJ>
      <RemoteTeam class="com.business.classes.MyTeam">
            .. // same as above
            <RemoteFacade class="com.business.interfaces.IMyTeamFacade"/>
      </RemoteTeam>
</DOTJ>
```

At each application nodes application developers *must* explicitly announce that remote façade interface inside the XML file deployed at that node. Actually, the purpose of labeling remote façades in the XML files is that these interfaces must be transformed by the DTL transformers (will be discussed later in this chapter in more detail). Therefore, the DTL transformer needs to distinguish between remote façade interfaces and other normal interfaces which might be implemented by remote team classes.

5.2.2.3 Assign Remote Team Activation Priority (RTAP) values to Remote Teams

As discussed in Chapter 3, application developers can assign an activation priority value (called the *Remote Team Activation Priority (RTAP)*) for every remote team class. The RTAP values are important to organize and resolve the activation priorities of team instances at runtime. An RTAP value is an integer number, and a remote team class with the highest

RTAP value is the one to acquire the highest activation priority among other teams. The RTAP value "0" (the default) indicates that remote teams are competing among each other to have the highest activation priority. In this case, the *last* remote team instance to register in the DOTM is the one to obtain the highest priority over all other team instances (which have the same RTAP value); of course, in case they are all found activated.

For example, to give team instances of the team class **MyTeam** a high activation priority, the RTAP value 1199 could be assigned as follows:

```
<?xml version="1.0"?>
<DOTJ>
    <RemoteTeam class="com.business.classes.MyTeam" activationPriority=1199>
              // same as above
    </RemoteTeam>
</DOTJ>
```

5.2.3 Labeling of Remote Base Classes

Remote base classes of remote base objects which are deployed on a specific node must be labeled in the XML file created at that node. The XML node `RemoteBases` is used to group the remote base classes on a specific node. The following XML portion illustrates how to label the remote base class **MyBase** shown in Figure 5.2:

```
<?xml version="1.0"?>
<DOTJ>
  ..
  <RemoteBases>
       <Base class="org.remoteClasses.classes.MyBase"/>
       .
       .
  </RemoteBases>
</DOTJ>
```

Note that **`Base`** entry does not give any description about the relationship between **MyBase** and the remote role **MyRole**; because the DOT/J framework will bind their instances dynamically at runtime. Here, **MyBase** is only known as a remote base class without explicitly declaring which of its methods need to be "intercepted" by roles' CIMBs. The list of remote "pointcuts" (i.e. remote CIMBs) will not be woven into the bytecode of base class; rather, will be known at runtime.

5.3 The Generic Remote Interfaces (GRIs)

The DOT/J framework adopts the Java-RMI middleware in the design and implementation of the communication between application remote objects which are involved in the RRP relationships. Therefore, the execution of remote roles' callin methods that are associated with remote CIMBs, besides remote method-COMBs invocations and remote BGPs evaluations must be performed as *Remote Method Invocations (RMIs)*.

According to the discussion in Chapter 3, the remote base classes participated in the RRP process *must* implement the Generic Remote Bases Interface (GRBI). Similarly, all remote team instances *must* implement either the Generic Remote Teams Interface (GRTI), or a specific user-defined "remote façade". This section discusses the structures of GRBI and GRTI, and explains how to employ them in applications.

But before embarking on the discussion, it is important to mention that both generic remote interfaces must obey the following criteria:

1. They should fulfill the requirements of the RRP mentioned in Chapter 3 – Section 3.8.1.
2. They should declare minimal number of methods as much as possible in order to save network bandwidth, and to reduce the time required by the Java-RMI system, for example, to serialize arguments and create dynamic proxy objects.

5.3.1 The Generic Remote Bases Interface - GRBI

The GRBI is a Java interface, named as `IRemoteBase`, involved in the DRS library as a pre-generated interface. Figure 5.3 shows the class diagram of the GRBI. The `IRemoteBase` interface should be implemented by all remote base objects. It is requested to enable the remote COMB invocations issued by the roles played by a specific remote base object. The GRBI interface extends the remarkable interface `java.rmi.Remote` of the Java-RMI system. The GRBI declares the following remote methods:

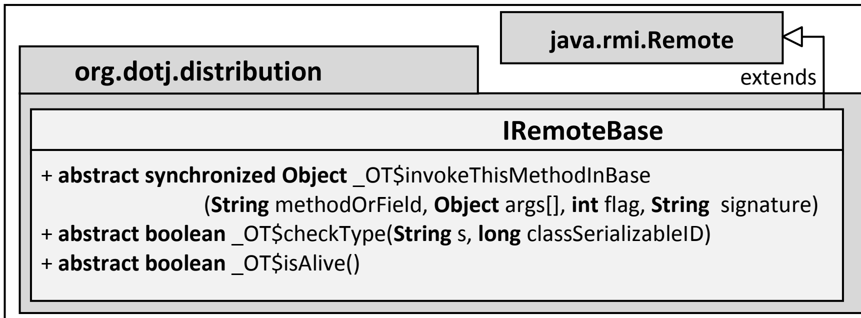

Figure 5.3 The Generic Remote Base Interface class diagram.

1. The method `_OT$invokeThisMethodInBase`[1] is a centric method that enables the execution of remote COMB requests. This method is called *Generic Invocation by Reflection Method (GIRM)*; because it could be used to invoke *any* of base object's methods or access any of its fields (via getter and setter methods) using the Reflection mechanism of Java. In addition, remote roles invoke this method to evaluate remote BGP expressions. Using a single generic method synchronously helps to handle concurrent access to remote base objects by several role instances safely [5]. When called, the following parameters must be provided:
   I. `methodOrField`: The name of base object's method being invoked, or the name of field to get or set.
   II. `args[]`: the required arguments by the base's method in case of remote method-COMB. When a field setter is encountered, then this parameter comprises the new value of the targeted field.
   III. `flag`: an integer value indicates the type of COMB invocation; specifically, 1 stands for a base method invocation, 2 for a field setter and 3 for field getter invocation. In this way, the GIRM method could be used to handle all remote COMB types instead of implementing a specific RMI for each one of them; hence the "*generic*" term in the acronym of GIRM.

[1] The prefix _OT$ has been coined in OT/J to hide language-specific methods from the user source code [80] (§A.0.3).

IV. `signature`: the signature[1] of the desired method. Passing the base method's signature is important to deal with methods overloading and to simplify arguments unboxing (if required).

2. The method `_OT$checkType` is another remote method that will be used by remote team instances to *check* the validity of remote base objects before they are bound to role instances. This method requires the full-qualified class name of the base class (from which the remote base object under binding has been generated) and its *Class Serialization ID (CSID)* value. The method returns "true" if the provided class name and the CSID value both are, respectively, similar to class name and CSID value of the base class of the base object under binding. Later, this chapter will demonstrate how to generate CSID values for remote base classes, and shows how remote team instances can obtain *identical* values at runtime; thus, the verification process could be handled accurately.
3. Finally, the method `_OT$isAlive` will be used transparently to check the availability of a specific remote base object. It always returns "true." At runtime, this method is invoked by role instances in order to check the availability of their base players *before* any remote COMBs invocation is being issued.

5.3.2 The Generic Remote Teams Interface – The GRTI

The GRTI is remote interface named as `IRemoteTeam`. Figure 5.4 shows the class diagram of the GRTI. Next paragraphs will discuss the methods declared by this interface. The figure does not show the complete list of the declared methods; only the methods needed for establishing a precise RRP are shown. The complete description of the `IRemoteTeam` interface could be found in Appendix-A.

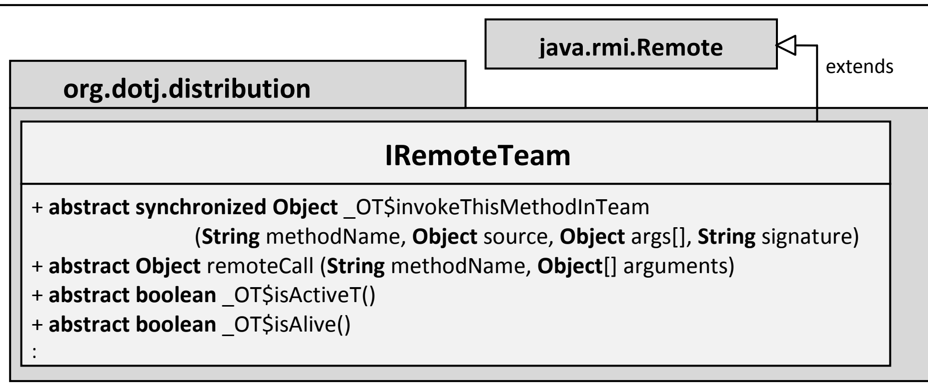

Figure 5.4 The Generic Remote Team Interface class diagram.

The GRTI defines the following methods:

1. The method `_OT$invokeThisMethodInTeam` is another remote GIRM that will be invoked by remote base objects to invoke transparently the Team Level Wrapper

[1] In Java, a method signature is the method name and the types of its parameters. In DOT/J, the JVM signatures are used. For example, "(IF)J" is a signature for method that requires two arguments of types integer and float, and returns a value of type long.

Methods (TLWMs) associated with the roles' callin methods mentioned in remote CIMBs. The caller base object supplies the *name* of the desired TLWM in the parameter `methodName`. In fact, the list of *all* TLWMs' names is delivered first to the DOTM by remote team instances during the registration process. Thus, remote base objects can easily know and select the name of the required TLWM.

The caller base object submits its *remote stub* generated from the GRBI in the parameter `source`. The GRBI stub will be used at remote teams' sides by the Java-RMI system to create "at fly" a *proxy* object of the remote base object. The new generated stub acts as a surrogate of the original remote base object; specifically in the binding process and during remote COMBs manipulations.

The arguments required by the targeted TLWM are passed in the `args` parameter. The submitted arguments are automatically "boxed" and packed in an array of objects. Finally, the caller object passes the `signature` of its currently *intercepted* method. In practice, passing the signature of the intercepted method is very important for remote team instances to designate the accurate order and type of submitted arguments. Thus, they could be easily un-boxed.

2. The method `_OT$isAlive` is invoked to check the availability of a specific remote team instance. This method, which always returns "`true`," invoked by remote base objects during the execution of remote CIMBs and (just prior) the actual invocation of the corresponding TLWMs. Checking the availability of remote team instances is important to avoid interruption of application execution due to remote exceptions. Another situation where this method could be invoked is when a remote team instance is being accessed as a distributed component. That is, before calling any of the team's methods, the availability of remote team instance must be checked first; in fact, the DOTM cannot guarantee the existence of any remote team instance because it registers only their remote stubs. For this purpose, the DOTM invokes this method to check the availability of the registered remote team instances when it prepares the Remote Teams List (RTL) upon the requests of remote base objects.
3. The method `_OT$isActiveT` will be invoked by remote base objects to check the activation status of remote team instances. More precisely, a remote base object invokes this method just before it invokes the TLWM of a remote team. This method returns the *current activation status* of the remote team instance; i.e. it returns a "true/false" value.
4. The method `remoteCall` enables application developers to invoke any public method of a specific remote team instance remotely by client program. This method helps access remote team instances as distributed components. If a remote team instance provides only the GRTI interface, then this method is the only explicit "toolkit" for application developers to access remote team instances at the source code level of their programs. This method is *visible* to the client code (i.e. it does not contain the prefix `_OT$` like previous methods). However, it requires the *name* of the desired team method to call, and the list of arguments required by that team's method. Actually, application developers need to provide a correct method name and pack the arguments in the correct order. Access of remote team instances in this way is considered verbose, though. The other alternative proposed by DOT/J framework, as will be shown later in this chapter, is to provide user-defined remote façade instead of the GRTI.

   Nevertheless, to explain how programmers could use this method in their applications, the following code snippet invokes the method `remoMeth` of a remote team instance of type **MyTeam**. Casting returned results (if any) is the responsibility of application programmers. If the result has not been casted accurately, then side-effect problems might appear.

```
IRemoteTeam remoteTeam = ... ; // get a handle to the remote team instance
result= (String)remoteTeam.remoteCall("remoMeth", new Object[]{arg1, arg2, arg3});
```

5.3.3 Implementing Remote Façades

Application developers can declare at the source-code phase a specific interface of their own, called in DOT/J framework as "remote façade", which facilitates the access of remote team instance *remotely*. To implement remote façades, the following important points must be taken into account: (1) the transparency of DOT/J framework should be preserved; i.e. hide verbose preparations to let remote team classes implement these remote façades, and (2) application developers are free to declare any interface, i.e. there should not be any limitation on the number of methods these remote façades could declare.

The following steps illustrate how application developers can implement the remote façade **IMyTeamFacade** shown in Figure 5.2:

1. *Designate remote team's methods that will be invoked remotel*y. Assume that the team class **MyTeam** (shown in Figure 5.2) has implemented the following methods, which are intended to be accessed remotely:

```
public boolean  firstMethod  (int a, String b, MyType c) {…}
public String   secondMethod (MyBase as MyRole ins, float fact) {…}
```

2. *Create a new Java interface* that declares the designated methods with the following consideration: "*convert the Explicit Lifting Parameter declarations in all methods into parameters of base class types.*" For example, Listing 5.1 shows how the **IMyTeamFacade** remote façade could be described. Note the parameters' declaration of method `secondMethod` at line 5 after the explicit lifting declaration [`MyBase as MyRole ins`] has been replaced by a parameter of class type **MyBase**. This conversion is important because OT/J does not allow declaring explicit lifting parameters outside team classes.

Listing 5.1: Declaring the remote façade interface `IMyTeamFacade`.

```
1. package com.business.interfaces;
2. public interface IMyTeamFacade
3. {
4.    public boolean  firstMethod  (int a, String b, MyType c)
5.    public String   secondMethod (MyBase ins, float fact)
6. }
```

3. Add **IMyTeamFacade** to the `implements` clause of team class **MyTeam**.

4. *Announce the implementation of* **IMyTeamFacade** *façade in the XML file* (as discussed in subsection 5.2.2.2) at the all application nodes. For example, the XML file deployed on host H2 of Figure 5.2 will look like:

```
<?xml version="1.0"?>
<DOTJ>
  <RemoteBases>
      <Base class=”com.myPkg.classes.MyBase”/>
  </RemoteBases>
  <RemoteFacade class=”com.business.interfaces.IMyTeamFacade”/>
</DOTJ>
```

5.4 The DOT/J Transformation Library (DTL)

Before embarking on a discussion of the DTL details, recall that remote base objects and remote team instances need to work in harmony during the execution of RRP activities. To preserve the harmony of communication between remote objects, their classes should be first augmented with the necessary code segments which should precisely implement the provided-required interfaces discussed earlier (the GRTI and GRBI).

In case of remote base classes, preparing them for the RRP *must not deform* their structures (e.g. class inheritance hierarchy) or distort the original behavior of their objects undesirably at runtime. In this regard, two alternatives could be used to augment remote base classes with the code segments required for enabling their objects to enroll in the RRP. Figure 5.5 illustrates the two possible techniques to prepare the remote base class **MyBase** at the load-time phase. To determine the effects of each alternative on the remote base class, the **MyBase** class has been implemented in a hierarchical structure as shown in part (a) of the figure. Now, let us discuss these techniques as follows:

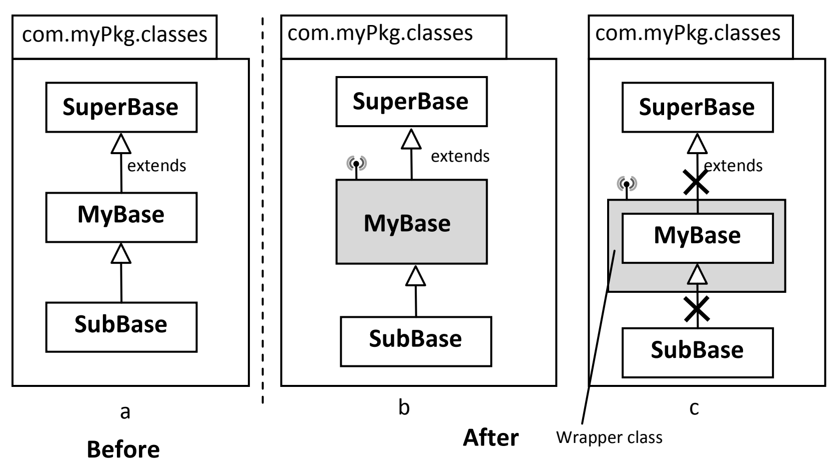

Figure 5.5. Approaches for transforming the base class **MyBase**: embedding the RRP-requirements within the base class itself (b), or wrapping the base class with a “wrapper class” (c).

1. The first technique suggests that the necessary code segments are *injected* directly into the target base class, as shown in part (b) of the figure. In this case, the base class **MyBase** acquires new code segments in order to facilitate the RRP in addition to its original code. The main *disadvantage* of this approach is that it may increase the size of class file appreciably if too much code is injected. Another drawback of this technique is that it impacts the results of, for instance, base class profiling when the Java Reflection

API is used; because new methods and fields will be added to the base class. Nevertheless, it preserves base classes' hierarchies unchanged. That is, inheritance relationships between the base class and its parent and/or child remain intact after modification. In addition, all the generated base objects will have object-IDs created from this class, which is important to avert the Object Schizophrenia Problems (OSPs)[1] [16], which include broken delegations among multiple object identities, and the Doppelgangers [76].

2. Figure 5.5 (c) shows the second option, which suggests *wrapping* the base class **MyBase** with a specific wrapper class (known as Wrapper or Adapter Pattern [4]). In this case, objects of **MyBase** acquire the capability for RRP by creating corresponding wrapper objects via specific objects factory. Unfortunately, this approach has many disadvantages; *first*, it causes many serious deformations to base class structure, and hence the functionality of the generated objects. *Second*, it violates class hierarchy; because it breaks the direct inheritance links between its parent and child. *Third*, it may result in OSPs; because team instances will bind *wrapper objects* on behalf of the actual remote base objects. *Fourth*, the delegations made from objects of the child class **SubBase** toward the super object of type **MyBase** are lost. Moreover, problems like base objects persistency can occur. The DOT/J framework adopts the first alternative, which is applied on the remote team and role classes. Some AOP approaches like TRAP/J [81] adopt the wrapper solution.

5.4.1 The Bytecode Transformation of Classes – A Quick Overview

In Java programming language, classes are the basic units of application modularity. Application developers write the description of a specific class (the source code) following the Java syntax, and then the compiler of Java translates the source code file into so-called *bytecode* file (with `.class` extension). The bytecode of a class is the actual code that the JVM can execute. However, before execution starts, the bytecode of application classes must be loaded into the JVM. Taking into account this fundament, the bytecode transformation is the mechanism that enables application programmers to perform a set of desirable modifications on the bytecode of a specific class *before* it has practically loaded into the JVM. These modifications include adding new member functions and attributes, removing existing members or attributes, implementing new interfaces, etc. Thus, the internal structure of that class could be changed.

5.4.1.1 The Java Class-Loading System [80, 5]

The process of class-loading is taking place, actually, when the JVM is started and the core libraries of Java[2] are loaded by the Bootstrap Class-Loader [5], which is part of the JVM core system itself. However, Java allows developers to load the user-defined application classes by using class loaders of their own. Specifically, Java enables the use of *Extensions class-loader* and *System class-loader*. The Extensions class-loader loads application classes from the files and JAR-libraries stored in one of the extension directories (normally the `<%JAVA-HOME%>/lib/ext`). The System class-loader, on the other hand, loads the classes found in

[1] OSPs are out the scope of this dissertation.

[2] Those stored in the `rt.jar` file, which includes the core classes of Java like `java.lang.Object`

the folders listed in the system property `java.class.path`, which maps the system variable CLASSPATH into the Java source code level.

Normally, all the user-defined classes are loaded by the default system class-loader of Java. However, Java allows developers to replace this class loader with their own loaders by extending the `java.lang.ClassLoader` class (the parent of all class-loaders), which enables the *customization* of the whole class loading process. This facilitation enables the dynamic loading of classes at runtime. Consequently, many approaches have been developed to handle the loading of application classes, intercept their loaded code in order to perform some modifications, and then dispatch these classes to the *upper* class-loaders until the JVM. JMangler [29] is one of the famous approaches in the customized Java class-loader systems; some of its features will be discussed later in this section.

5.4.1.2 Analyzing the Bytecode of Classes

The bytecode of any Java class[1] is a stream of sequential bytes impacted altogether in such a way it is very hard for application developers to read or distinguish the Java instructions embedded within that cipher. Thus, transforming the bytecode of classes at this fine-grain level is a terrible process; hence, it can result with tremendous deformations on classes and then the generated objects. To understand size of this problem, let us take a simple example. The code snippet of Figure 5.6 (a) illustrates the source code of the method `div(…)`. Figure 5.6 (c) shows the equivalent compiled code, which is the one loaded into the JVM. Clearly, it is hard to read or recognize the instructions of method `div(…)` from that code (decrypted as hexadecimal digits while in fact it is a set of binary bits). Thus, a more readable format for the compiled Java classes has been introduced in [96], which uses so-called *mnemonics of instruction "opcodes"* (stands for operation code) instead of the byte stream format. Figure 5.6 (b) shows the disassembled bytecode[2] of method `div(…)`, which is generated from the code listed in part (c). The explanation of opcode mnemonics, however, is beyond the scope of this dissertation. Anyway, it is easier for developers to cope with this mnemonic code rather than byte streams.

```
public int div(int x, int y)
  {
     if(y != 0)
        return x / y;
     else
        return -9999;
  }
```

a

```
public int div (int x, int y) {
   //   0    0:iload_2
   //   1    1:ifeq          8
   //   2    4:iload_1
   //   3    5:iload_2
   //   4    6:idiv
   //   5    7:ireturn
   //   6    8:ldc1          #16  <Int 0xfffe7961>
   //   7   10:ireturn
}
```

b

```
2F 4D 3B 01 00 03 64 69 76 01 00 05 28 49 49 29  /M;___div___(II)
49 03 FF FE 79 61 01 00 01 78 01 00 01 49 01 00  I_¿ya___x___I__
01 79 01 00 0D 53 74 61 63 6B 4D 61 70 54 61 62  _y___StackMapTab
```

c

Figure 5.6. A Simple method written in Java in (a), its mnemonic instructions list (the disassembly code) (b), and part of its bytecode (c).

Now, to achieve a seamless bytecode transformation, an adequate mechanism is required to read the bytecode of classes, and then convert that code into an accessible format that could be manipulated easily by developers at the load-time. In this field, the Byte Code

[1] Structure of Java Class files is beyond the scope of this dissertation. For additional information about Java class structure and organization read in [97].

[2] We use (DJ Java Decompiler v.3.10.10.93 © 2000-2007 Atanas Neshkov) to decompile the bytecode.

Engineering Library (BCEL) [50] imposes itself as one of the most brilliant approaches. The BCEL is a toolkit for the static analysis and dynamic transformation and/or creation of Java class files. The API library provided by BCEL could be easily used to instrument a specific *loaded* class from its physical file, and manipulate the declared attributes and methods of that class. Furthermore, the BCEL system has been implemented totally in Java and available to use for free under the GNU-LGPL license terms.

The BCEL mainly provides a generic set of modules that captures the bytecode of a loaded class. For example, the `ClassGen` module represents the decompiled Java class's description, and could be used to access the methods and fields of that class in a way similar to the Java Reflection technique. Practically, this means that developers can instantiate objects from the intercepted class dynamically. Anyway, the generation of new coarse-grain code segments (e.g. new methods) or fine-grain bytecode instructions (e.g. an instruction of specific method) is simple and "user-friendly" in BCEL. The developer can use the `InstructionFactory` class to create different types of Java instructions in a simple manner by invoking the proper method in this class. For instance, to create an "`if`" statement to compare the equality of specific value to the "`null`" value, then the developer can use the method `createBranchInstruction` of `InstructionFactory` class statically as follows:

```
InstructionFactory.createBranchInstruction(Constants.NULL,…);
```

Eventually, it is the responsibility of the BCEL system to re-compose the transformed bytecode in a class file composed only of a stream of bytes with the same class name.

5.4.1.3 Class Transformation in OT/J and DOT/J

The Object Teams Runtime Environment (OTRE) uses a stack of sequent bytecode transformers to modify the code of local base and team classes that enroll in the *playedBy* relationships. The transformers of OT/J practically use the JMangler [29] system until the OTDT version 1.4. The current version of OTDT – under the Eclipse™ umbrella – recommends the using of Java Programming Language Instrumentation Services (JPLIS) [91], but still supporting the use of JMangler. The JPLIS is an extended package of Java, which provides services that allow Java's *agents* to instrument programs running on the JVM such that the bytecode of methods of the loaded classes could be modified. However, in order to transform the classes of remote bases, and to make use of the transformed team classes, DOT/J *extends* the stack of the bytecode transformers of OT/J by appending its own transformers as illustrated in Figure 5.7.

The DOT/J transformers are integrated with the OT/J transformers in two aspects. *First*, classes transformed by the OT/J transformers are the input to the DOT/J transformers. *Second*, the transformers of OT/J and DOT/J both using (more precisely *referring to*) the repository component of callin bindings; the Callin Binding Manager (CBM). The CBM component is used by OT/J as a *shared repository* among its transformers in order to store a bundle of information about the *playedBy* relationships, callin-binding declarations, the OT/J-specific attributes of classes and methods, etc. In addition, the CBM allows the sequent transformers to request the stored information and access them on-demand. In DOT/J, the transformers of the DTL refer to the CBM mainly to inquire information about the CIMBs of remote roles.

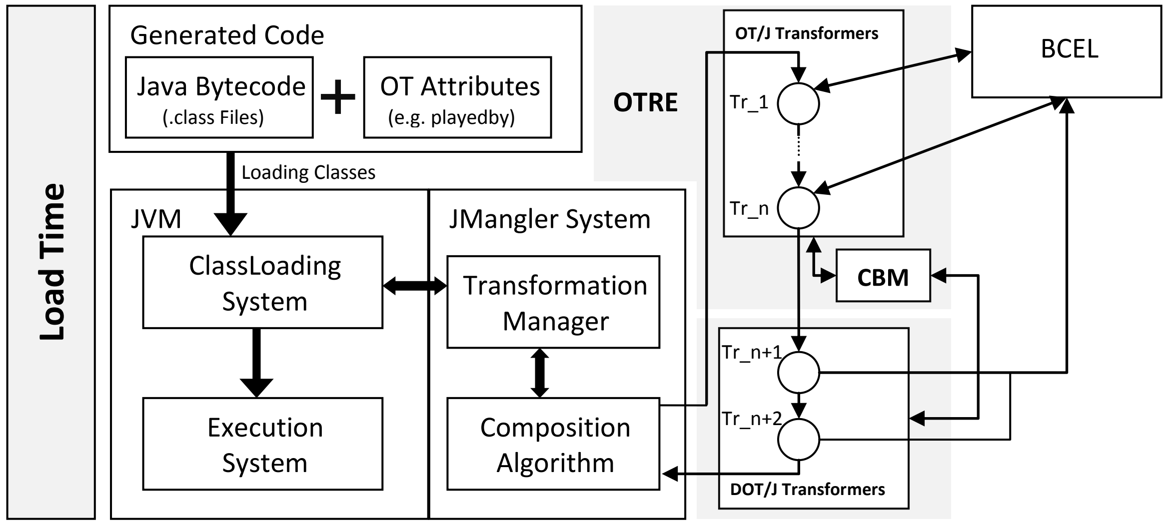

Figure 5.7. The integration between DOT/J transformers and the OT/J Transformers.

5.4.2 The Remote Team Transformer

The DTL transformers transform team and base classes *after* they have been transformed by the OT/J transformers. The Remote Team Bytecode Transformer (RTBT) transforms all the team classes labeled in the XML files. The transformation algorithm applied in the RTBT is depicted in the pseudo-code shown in Listing 5.2.

The DTL takes into account that role classes are loaded (compulsory) when the team class encloses them is being loaded. Thus, any desirable transformations on role classes must take place at this point. For this purpose, the RTBT is dedicated to transform remote team classes and to delegate the Remote Role Bytecode Transformer (RRBT) to transform both the role's interface and its implementation class as well.

5.4.2.1 The Transformation of Remote Team classes

When the RTBT is triggered for the first time by the class-loading system, it immediately parses the deployed XML file, and establishes a small database to store and organize remote team and remote role classes. Whenever a remote team class is being loaded, it will be intercepted by the RTBT. Consequently, the transformation process starts according to the algorithm of Listing 5.2. First, the RTBT checks if the loaded class is a valid OT/J team class by reading the class attribute `OTClassFlags`, which is a special attribute of teams. Then, the process of preparing that team class *for being a remote team class* is taking place.

- *Preparing team instances for being remote team instances*

Using the BCEL APIs, the RTBT adds the `IRemoteTeam` GRTI easily as shown below. The `cg` is an instance of the `ClassGen` class, which represents the decompiled class of the current loaded team class, and the `DOTJConstants` is a shared class created in the DOT/J framework to store model-specific fields and methods – see Appendix-A for more details.

```
            cg.addInterface(DOTJConstants.iremoteTeamClassName);
```

The implementation of this interface will be generated completely by the RTBT. Thus, the RTBT adds the methods declared in the GRTI interface. Consequently, the new declaration of the remote team class **MyTeam**, after this step becomes as follows:

```
    public class MyTeam extends Team implements IRemoteTeam{..}
```

However, as the team **MyTeam** implements the **IMyTeamFacade** façade (as shown in Figure 5.2), then the new class declaration of this team becomes:

```
    public class MyTeam extends Team implements IMyTeamFacade {..}
```

Afterward, the RTBT injects into the bytecode of team class a set of *DOT/J-specific* fields that could be divided into two groups: the first group comprises a set of internal-usage fields that will be used by the DOT/J Runtime System (DRS) to facilitate the communication between remote team instances and remote base objects. The second group comprises a set of fields that represent the remote state of the team instances generated from this team class. For instance, the RTBT adds the following fields[1]:

```
    public IRemoteTeam          _OT$theRemoteTeamVersion;
    public String               _OT$teamUID;
```

The first field will hold at runtime the *remote stub* that a remote team instance will generate when it exposed as a remote object (will be discussed in the next paragraphs). The second field will hold the Remote Team Unique Identifier (RT-UID) that will be given to the remote team instance after it has registered in the DOTM. Note the prefix `_OT$` in fields names, which is used to hide these fields from programmers source code.

At every team class *constructor*, the RTBT injects the necessary code segments that enable team instances to register in the DOTM. The default constructor of the team class **MyTeam** becomes as follows, where the `_OT$registerTeamAtDOTM` is the method responsible for registering the team instance (discussed below):

```
public MyTeam()
{ …
   _OT$teamUID = _OT$registerTeamAtDOTM();
}
```

- *Preparing remote team instances for a seamless registration in the DOTM*

The new generated method `_OT$registerTeamAtDOTM` shown in Listing 5.3 will perform the following important steps:

---

[1] The transformation tasks are described in brief in order to increase the process understandability.

1) Generates the *remote stub* for the instantiated remote team instance (line 5). The remote stub is generated according to the Java-RMI disciplines[1]. In the current case, the remote

Listing 5.2: The Remote-Team transformation pseudo code

Step-1:

if *the XML file is not parsed yet* ***then***

Parse it: scan entries and organize the labeled team, role and remote façade classes

Step-2*:*

If *the current loaded class is recognized as a valid remote team,* ***then***

- *Implement the* `IRemoteTeam` *interface.*
- *Add the necessary code segments allow the generating of remote team instances from this team class.*
- *Enable instances of this team class to register themselves automatically in the DOTM.*
- *Add the required methods that:*
  - *(a) Facilitate the execution of remote CIMBs.*
  - *(b) Facilitate Lifting/Lowering of remote base objects and role instances.*
  - *(c) Facilitate the invocation and execution of team's public methods remotely.*
- ***For*** *each remote role declared in the current remote team*
  Add that role to the list of role classes to be transformed by the RRBT

Else if *the loaded class is one of the labeled remote role classes* ***then***

Transform it by the RRBT as follows:

- *Prepare that role class for being bound with remote base objects.*
- *Replace local base references with remote references.*
- *Reformulate all base method invocations and field access to use the remote stub of that remote base object.*

  Else if *the loaded class is recognized as Team-Role class* ***then***
  - *Transform it first as remote team by the RTBT.*
  - *Transform it as remote role by the RRBT.*

Listing 5.3: The Registration method is injected into remote team class at load time.

```
1. public String _OT$registerTeamAtDOTM()
2. {
3.  String _OT$theName = null;
4.  try{
5.      _OT$theRemoteTeamVersion =
           (IRemoteTeam)UnicastRemoteObject.exportObject(this, 0);
6.      _OT$dom = DistributedObjectsManager.
                                 getCurrentDistributedObjectsManager();
7.      HashMap _OT$RCPList = _OT$generateRemoteCallinPointcutsList ();
        _OT$theName = _OT$dom._OT$registerRemoteTeam
8.         ("com.business.classes.MyTeam",
            (Object)_OT$theRemoteTeamVersion,
            _OT$RCPList, 1199);
9.   }
10.  catch(Exception exception){...}
11. return _OT$theName;
12.}
```

[1] More information about exporting remote objects in Java-RMI system could be found in Appendix-B.

stub will be generated via exporting the remote team instance *dynamically* by the RMI system. In line 5, the remote GRTI is used to "produce" the required remote stub. In case that a remote façade has been provided by the team class, then the RTBT will automatically detect and prepare the transformed remote team class to "produce" remote stub from that façade instead of the GRTI.

2) Grips the local DOTM instance (line 6). All the created remote team instances on a specific application node must preserve an immutable reference to the DOTM. As line 6 shows, the DRS enables remote team instances to obtain the currently running local DOTM instance via the API `getCurrentDistributedObjectsManager()`[1]. This API is visible and could be used by application developers to get a handle to the local DOTM, which allows inquiring the DOTM, for example, about the registered teams and their availability status.
3) Prepares the Remote Callin Pointcuts (RCP) list – the list of the CIMBs declared by the remote roles of the team class under transformation. This list should comprise the remote CIMB declarations (pointcuts) that will intercept the calls of remote base objects' methods. The list must be prepared in a proper way so that remote base objects can easily detect and perform the invocation of the role's callin methods associated with the remote CIMBs. The preparation of this list is carried out automatically. The method `_OT$generateRemoteCallinPointcutsList`, as shown in line 7, is woven by the RTBT. For this purpose, the RTBT refers to the CBM component and fetches the list of *local* CIMBs declarations of all remote roles.
4) Registers the generated remote team instances in the DOTM (line 8). Team instances invoke the `_OT$registerRemoteTeam` API of the DOTM, and supply automatically the necessary arguments, which are: the full-qualified class name of the team under registration, the generated remote stub, the prepared RCP list, and the Remote Team Activation Priority (RTAP) value associated to that team class (in this example it equals 1199). The value of RTAP is known during the parsing of the XML file. If no RTAP value has been specified, then "0" will be used. If the registration process performed successfully by the DOTM, then the registered team instance will get a Remote Team Unique Identification (RT-UID) value.

- *The Requirements of Remote CIMBs Execution*

The phrase "remote CIMB execution" denotes the process of matching a specific remote CIMB declaration at the call-time of a specific base object method, and executing the associated role callin method. Remote team instances must have the following capabilities in order to realize a precise remote CIMBs execution:

1. Lift the remote stub (practically, a *proxy* object) of the remote base object (which its method's call has been intercepted by the remote CIMB) into the same role instance all the time.
2. Execute the accurate TLWM associated with the desired role-level callin method.
3. Return properly the expected result (if demanded) to the remote base object in case of replacement remote CIMB.

[1] A complete description of the DOTM could be found in Appendix-A.2.

The RTBT injects into the remote team class a *new* lifting[1] method for each remote role it has confined. The RTBT does not transform the counterpart lifting methods which are created by the OT/J compiler due to the restrictions discussed in Chapter 1 – Section 1.3. The RTBT injects, for example, into the **MyTeam** team class the lifting method shown in Listing 5.4, which lifts the remote stubs of remote base objects of base class type **MyBase** into role instances of type **MyRole** class.

Before the remote team instance binds the provided remote base stub with a new role instance (if no role instance has been previously cached as line 5 shows), a *base object validity check* is carried out (line 7) to make sure that the provided stub belongs to a base object of the precise base class type. As the code at line 7 illustrates, the team instance invokes remotely the method `_OT$checkType` on the provided stub for this purpose.

Listing 5.4: A new version of Lifting method is injected into remote team class.

```
1.  protected MyRole _OT$liftTo$MyRole (IRemoteBase _OT$base)
2.  {
3.    MyRole _OT$role = null;
4.    if(_OT$base == null) return null;
5.    if(!_OT$cache_OT$MyRole.containsKey(_OT$theBase))
6.     {
7.      if(_OT$base._OT$checkType
            ("com.myPkg.classes.MyBase",<CSID>)){
         ... //create a new role instance passing _OT$theBase to the constructor
8.        }
9.        else
10.         throw new DOTJIncorrectRemoteBaseTypeException(..);
11.    }
12.   else {
        ...  // fetch the role from team's cache
13.    }
14.  return _OT$role;
15. }
```

The first argument required by this method is the full-qualified class name of base class; in this case `com.myPkg.classes.MyBase`. The second argument (the Class Serialization ID) is *calculated* for the base class under lifting by the RTBT at the load-time. Specifically, the RTBT *uses* the local description of base class (the description at the team's side) to calculate this value. It uses all the declared methods and fields to calculate this value, and then injects it as an argument to the `_OT$checkType` method.

- *The Team Level Wrapper Methods (TLWMs)*

At runtime, remote team instance needs to lift the remote stubs of remote base objects into their bound role instances at two situations: the first is when a specific TLWM is invoked (i.e. due to a remote CIMB execution). The second situation is when a team's method which has declared an *explicit lifting parameter* is invoked. In OT/J, the compiler *generates* a local TLWM for each declared CIMB in the team class as part of *playedBy* relationship interpretation. After that, the OT/J Base Transformer completes the implementation of *playedBy* by modifying the base class methods which are bounded in CIMB expressions.

[1] Lifting and Lowering mechanisms discussed in detail in Chapter 2.

The OT/J transformer injects in base methods the necessary TLWMs invocation statements. In this way, base objects detect and execute the role's callin methods once a "joinpoint" is matched. In case of RRP, remote base objects could know the necessary TLWMs corresponding to the declared remote CIMBs only at runtime; because DOT/J framework does not adopt binding base methods in remote CIMBs *statically*.

Actually, the RTBT is not able to transform the local TLWMs to fit the RRP needs; because OT/J weaves roles into base classes at load-time. Thus, OT/J transformers created these TLWMs in such a way team instances *invoke* some base object methods that are generated at the load-time with attributes and model-specific values like the team's ID, which are known only at the load-time. Therefore, the RTBT creates a *new* TLWM for each declared remote CIMB in the bytecode of remote team class, and *removes* the local counterpart TLWM. The new TLWM, however, should perform the same task of the local one but for the RRP.

The new generated TLWMs are named according to the following format:

```
_OT$<BaseClass>$<RoleClass>$<RLM>$<BLM>                                (5.1)
```

Where

`_OT$` : is a prefix used to hide the method from client code.
`BaseClass` : The simple name of base class.
`RoleClass` : the simple name of role class.
`RLM` : the Role Level Method mentioned in the remote CIMB expression.
`BLM` : the Base Level Method mentioned in the remote CIMB expression.

For example, if the role class **MyRole** declares the following remote CIMB:

```
String m() ← replace String m();                                       (5.2)
```

Then, the corresponding TLWM will be named as `String _OT$MyBase$MyRole$m$m (…)`. The implementation of the new TLWMs is woven entirely by the RTBT to perform the following tasks:

(1) Activates *implicitly* the team instance.
(2) Lifts the provided remote base stub into a specific role instance.
(3) Calls the designated role-level method `m` of the lifted role instance.
(4) Returns any expected value, and deactivates *implicitly* the team instance.

- *Preparing the RCP list*

The RTBT injects into the bytecode of each remote team class the method `_OT$generateRemoteCallinPointcutsList`, which packs the list of all remote CIMBs declared by the remote roles of each remote team as follows: "for each bound remote base class, create a list of *Remote Callin Pointcuts* (RCPs). Each RCP entry encapsulates a specific declared remote CIMB that intercepts one of base class methods." An RCP consists of the following parts:

- CM : The Callin Modifier (**A** for `after`, **B** for `before`, and **R** for `replace`).
- TLWM: The name of the TLWM associated to the designated CIMB.
- BLMS: The signature of the Base-level Method (The JVM signature).

For example, if the role **MyRole** class declares the following remote CIMBs:

```
role_m1 ← after m;                                        (5.3)
role_m2 ← before m;                                       (5.4)
```

Then, the RTBT will prepare the RCP list[1] for remote base class **MyBase** inside the bytecode of team class **MyTeam** as follows:

Base Class	Remote Callin Pointcuts		
	CIM	TLWM	BLMS
MyBase	R	_OT$MyBase$MyRole$m$m	()Ljava/lang/String;
	A	_OT$MyBase$MyRole$role_m1$m	()Ljava/lang/String;
	B	_OT$MyBase$MyRole$role_m2$m	()Ljava/lang/String;

- *Implement the methods of GRTI interface*

The RTBT weaves the implementation of the declared methods of the GRTI interface into each remote team class. The most important method to implement is the remote GIRM `_OT$invokeThisMethodInTeam`, which is the only "port" through which remote base objects can invoke the TLWMs. The implementation code of this method uses the Java Reflection API to invoke the desired TLWM method.

- *Manipulate the "Explicit Lifting Parameter" declarations*

The *visible* remote team's methods that declare *explicit lifting parameters* for remote base classes need to be transformed: when the application developer calls one of these methods remotely, she must supply a remote base object as an argument. Inside the team instance, the provided base object is lifted into a corresponding role instant. But, in Java-RMI system, when passing a remote object as an argument to a specific remote method invocation, the remote stub of that object will be passed instead *automatically*. More specifically, the called method of remote team instance will receive a remote stub of base object and not the object itself. Therefore, *all* remote team class methods that declare explicit lifting parameters for remote base classes must be compulsory transformed such that they accept the GRBI stubs instead of the actual remote base objects.

For example, if the team class **MyTeam** declares the method [`public void doSomething (MyBase` **`as`** `MyRole p, int v)`], which explicitly lifts a provided base object of type **MyBase** into a corresponding role instance of type **MyRole**. When compiling this method, the OT/J compiler reformulates this declaration to be as follows: [`public void doSomething (MyBase p, int v)`], and then it injects the necessary invocation statement to the lifting method (see Listing 5.4) exactly at the beginning of the method's implementation code. Once again, the GRBI will be passed. Thus, the RTBT changes the declaration of this method to be as follows: [`public void doSomething(IRemoteBase`

[1] The list is woven in the method `_OT$generateRemoteCallinPointcutsList`.

`p, int v)`]. In this case, the GRBI stubs could be passed safely and transparently. Also, lifting will be performed accurately since the base argument passed to the lifting method (see Listing 5.4 – Line 1) is of the GRBI type as well.

- *The Remote Team-Role Transformation*

When a remote team-role class is being loaded, then it should be transformed and equipped with the necessary toolkit that enables its instances to exhibit the dual characteristic of "Team-Role" module. In DOT/J, the DTL should transform remote team-role classes *twice*. The first round is carried out by the RTBT in order to transform these classes as *remote teams*, and the second round will be performed by the Remote Role Bytecode Transformer (RRBT) (discussed in the next sub-subsection), which transforms the team-role classes as remote roles *if* they are played by specific base classes remotely.

5.4.2.2 The Transformation of Remote Role classes

As explained earlier, OT/J realizes the implementation of roles by generating, at the compilation time, two classes for each declared roles: (1) a role interface that comprises its method declarations and (2) the implementation class which implements that interface (named as `__OT__<the simple role name>`). Consequently, the transformation process must be applied on both of them coincidently.

- *What to Transform?*

Most importantly, role instances of a specific remote team instance must be bound to *valid* base objects in order to enable them to issue remote COMBs and evaluate the BGP expressions accurately. Role classes in OT/J applications compiled in a way they preserve *local references* to their base objects, which results in a tight coupling between role and base objects at runtime. This tight coupling must be relaxed in order to enable the dynamic RRP relationships.

The technique of replacing local references with remote references in order to *split* application's functionality over the network nodes is used in several approaches like J-Orchestra [19], which is a system that enhances centralized Java applications with distributed capabilities. Other approaches are Addistant [57], which enables the adaptation of legacy Java-based applications to distributed execution on multiple JVMs, and RuggedJ [66], a transparent distribution framework for Java that distributes Java applications to run across a cluster of JVMs.

These approaches inspire the transformation of the bytecode of role classes by *replacing* the local base references with *remote base references*. Specifically, the remote role instances of a remote team instance will preserve a reference to the *proxy* objects generated from the provided GRBI stub by remote base objects. Accordingly, the *entire* local access in a remote role instance to its base object (e.g. the "*base calls*" and base fields' getters and setters) becomes *invalid*. Therefore, the Remote Role Bytecode Transformer (RRBT), which is dedicated to transform role classes, *replaces* all these local accesses with *remote method invocations* via the GRBI. The following paragraphs will explain this replacement step by step.

- *Replace the local base references in remote role classes with remote references*

Using the example shown in Figure 5.2, the compiler of OT/J generates an interface for the role class **MyRole** as shown in Listing 5.5, which declares the method [`public abstract MyBase _OT$getBase()`] (line 4) that returns an *immutable* base reference for the bounded base object. This method is transformed by the RRBT to be as [`public abstract IRemoteBase _OT$getBase()`] (line 7) in order to return the bounded remote stub of the remote base object.

Listing 5.5: Part of the compiled **MyRole** interface.

```
1. protected static interface MyTeam$MyRole
2. {
3.     /* before transformation
4.     public abstract MyBase _OT$getBase(); */
5.
6.     // after transformation
7.     public abstract IRemoteBase _OT$getBase();
       :
8. }
```

Inside the implementation class of role interface, a field of base class type is declared in order to hold the immutable reference of the bounded base object. In OT/J, the value of this field is known at the binding-time; to be exact, at the constructor of the role. Thus, the RRBT should transform the declaration of this field in order to hold the provided remote base stub instead. Listing 5.6 illustrates the constructor of the role class **MyRole** *before* and

Listing 5.6: Transforming the constructor of role class **MyRole**.

Before

```
protected class MyTeam$__OT__MyRole implements MyTeam.MyRole
{
 :
 public final MyBase _OT$base;
 public MyTeam$__OT__MyRole(MyBase base)
    {
        this$0 = MyTeam.this;
        super();
        _OT$base = base;
        _OT$cache_OT$AdvancedMath.put(_OT$base, (Object)this);
        :
    }}
```

After

```
protected class MyTeam$__OT__MyRole implements MyTeam.MyRole
{
 :
 public final IRemoteBase _OT$base;
 public MyTeam$__OT__MyRole(IRemoteBase base)
    {
        this$0 = MyTeam.this;
        super();
        _OT$base = base;
        _OT$cache_OT$AdvancedMath.put(_OT$base, (Object)this);
        :
    }}
```

after transformation. The immutable reference of the base object declared in line 4 (*before*) will be transformed to the one shown in line 4 (*after*). Note the modifications at the parameterized constructor at line 5 in both sub-lists.

- *Transformation of Base Guard Predicates*

The OT/J compiler converts each declared Guard Predicate (GP) or Base Guard Predicate (BGP) expression into a method which invoked to evaluate the logical expression; taking into account the levels of guarding (as discussed in Chapter 2 – Section 2.2.5.5). Each generated BGP's method requires a valid reference to the base object as an argument in order to access the fields and methods of that base object which are involved in the logical expression. To illustrate that, assume that the following BGP has been declared along the *playedBy* relationship shown in Figure 5.2:

```
(base when (MyBase.staticMethod() != 0 && base.fieldValue >=
MyTeam.threshold || base.getWhatever(this.x)))                    (5.5)
```

Then, the method shown in Listing 5.7 is generated in the role's implementation class to evaluate this expression. Note that the method `_OT$base_when` requires a base object reference (line 1) in order to invoke the method `getWhatever` and to get the value of the `fieldValue` field as illustrated at line 3.

Listing 5.7: The method generated for the BGP in expression (5.5).

```
1. public static synchronized boolean _OT$base_when (..,MyBase _base)
2. {
3.    return (MyBase.staticMethod() != 0) &&
            (_base.fieldValue >= MyTeam.threshold) ||
             _base.getWhatever(this.x));
4. }
```

The RRBT replaces the parameters of *local base class types* in all remote role classes with parameters of the type `IRemoteBase` (the GRBI). Therefore, the declaration of the method shown in Listing 5.7 will be transformed so that the type of the provided base parameter becomes [`IRemoteBase _base`] instead of [`MyBase _base`].

The RRBT parses the code of the BGP methods – using the BCEL instrumentation library – to detect the bytecode instructions of the method invocations and field accesses issued on the local base reference. Then, it replaces them by adequate *remote method invocations* counterparts that precisely map the *local* access into a remote access on the remote base object as follows:

1. Each call to a base method is replaced by an invocation to the `_OT$invokeThisMethodInBase` GIRM method on remote base stub. For example, the static method invocation `MyBase.staticMethod()` shown at line 3 of Listing 5.7 will be replaced by the following remote invocation:

```
((Integer) _base._OT$invokeThisMethodInBase(
                    "staticMethod",null,1,"()I")).intValue()!=0
```

Likewise, the method invocation `getWhatever(this.x)` is replaced by the following remote invocation:

```
((Boolean) _base._OT$invokeThisMethodInBase(
                    "getWhatever",new Object[]{x}, 1, "(I)Z")).booleanValue()
```

2. All access instructions to the bound base object's fields will be replaced by invocations to the GIRM method, taking into account un-boxing the fetched field value in case of remote field getter. Actually, Boxing and Un-boxing [5] (§5.1.7) of fields is important to avoid the side-effects when casting values. Anyway, the field access `_base.fieldValue` at line 3 of Listing 5.7 will be replaced by the following remote invocation (with the assumption that `fieldValue` is of reference type `Integer`):

```
((Integer) _base._OT$invokeThisMethodInBase(
                    "_OT$get$fieldValue",null,3,"()I")).intValue()
```

- *Transform the "base calls" in role's callin methods*

Inside the replacement callin method (i.e. is the role's method declared with the `callin` modifier) the compiler of OT/J performs a flow analysis to ensure that the original base object method is invoked before the replacement takes place. Therefore, the compiler expects a *base call* to the base-level method is declared. Normally, base calls are explicitly defined using the keyword `base` – the placeholder of the bound base object. To illustrate the idea, consider the CIMB expression 5.2 in Sub-subsection 5.4.2.1. Suppose that the application developer has implemented the role-level method `m` as shown in Listing 5.8. Then the compiler will replace the placeholder keyword **`base`** (at line 3) with the reference of the bound base object, and the call of base method `m` will be replaced by a call to the "chain" method of `m`.

Listing 5.8: The role-level callin method `m`.

```
1.  callin String m()
2.  {
3.    String tempS = base.m();
4.    return this.roleID + ":" +  tempS;
5.  }
```

To handle the base call properly in case of RRP, the RRBT substitutes the local "base call" with a remote method invocation to the GIRM method `_OT$invokeThisMethodInBase` as follows (casting the returned result adequately):

```
String tempS = (String) _OT$base._OT$invokeThisMethodInBase(
"_OT$doDispatch", new Object[]{ARTL,"m", ..},1, "(..)Ljava/lang/Object;");
```

In this way, the control flow during the chaining execution continues properly.

- *Transforming the remote field-COMBs and remote method-COMBs*

The compiler of OT/J practically *converts* the COMBs of base fields' getter and setter into calls of *static methods* in the base class of the bound object. For example, if the base class **MyBase** has declared the field `anyField` of type `MyType`, and the role class **MyRole** declares the following field-COMBs to get and set the field `anyField`, respectively:

```
MyType getAnyField()             → get MyType anyField;              (5.6)
void getAnyField(MyType val)     → set MyType anyField;              (5.7)
```

Then, the field getter expression (5.6) will be compiled into the role-level method shown in Listing 5.9. Note that the static method `_OT$get$anyField` shown in line 3 *will be* injected

Listing 5.9: The field getter expression at 5.6 is converted to a normal method

```
1.  public MyType getAnyField()
2.  {
3.    return MyBase._OT$get$anyField(_OT$base);
4.  }
```

in the bytecode of the base class by the OT/J transformers. However, the RRBT will replace this local static invocation with a remote invocation that invokes the same *static* method (i.e. `_OT$get$anyField`) using the GIRM. The new field getter method of Listing 5.9 after the transformation becomes as follows:

Note that the RRBT dispatches the field signature along the remote invocation, which is important to box and un-box the field value at both sides. Also, the integer value 3 (the third argument of the invocation at line 3 of Listing 5.10) is provided to indicate that the current invocation of the GIRM is a field getter.

Listing 5.10: The transformed field getter of `anyField`.

```
1. public MyType getAnyField()
2. {
3.  return (MyType)_OT$base._OT$invokeThisMethodInBase(
      "_OT$get$anyField", null, 3, <Signature of field's type>);
4.  }
```

The field setter expression 5.7 is manipulated likewise. The difference between the two remote field-COMB types is the arguments shipped to the GIRM. To illustrate this difference, the field setter expression 5.7 will be transformed by the RRBT as shown in Listing 5.11. The third argument value is 2 (at line 3 of Listing 5.11), which indicates that the GIRM invocation is a field setter. In addition, the new value of the base object's field is shipped in a single-value array of objects.

Listing 5.11: The transformed field setter of `anyField`.

```
1.  public void setAnyField(MyType value)
2.  {
3.   _OT$base._OT$invokeThisMethodInBase(
       "_OT$set$anyField", new Object[]{value}, 2, <Signature>);
4.  }
```

Concerning remote method-COMBs (which can bind any expected role-level method to a specific remote base object's method), the same transformation strategy is used. In simple words, *all* the declared remote method-COMB expressions of a role class will be transformed by replacing them with remote invocations to the GIRM. The RRBT, first, *packs* the arguments required by the base-level method and then invoke the GIRM method accordingly. For example, suppose that the role class **MyRole** declares the following remote method-COMB expression:

```
int roleCallOut(int x, float y) → String baseMethod(float b)
with {y → b, result ← x * Integer.parseInt(result)};                    (5.8)
```

This COMB declares the expected role method `roleCallOut`, which requires two parameters (the first is of type integer and the second is float), and returns a value of primitive integer type. At runtime, a call to this method will invoke the base method `baseMethod` instead, taking into consideration the mapping of the result and parameters according to the "`with`" clause (refer to Chapter 2 – Subsection 2.2.5.5 for more details about the parameters mapping of COMB expressions). The RRBT transforms this method by replacing the *direct* access to method `baseMethod` of the bound base object by the remote method invocation shown in Listing 5.12, which is a transformed version of the method generated by OT/J compiler for the COMB expression in 5.8.

Listing 5.12: The transformed remote COMB in expression (5.8).

```
1. public int roleCallOut(int x, float y){
2.  String s =
   (String)_OT$base._OT$invokeThisMethodInBase("baseMethod",
       new Object[]{y}, 1,"(F)Ljava/lang/String;");
3.  return x * Integer.parseInt(s);}
```

5.4.2.3 The Transformation of Remote Façades

The DTL library implements the Remote Façade Bytecode Transformer (RFBT) in order to convert the remote façade interfaces into compatible Java-RMI remote interfaces. Thus, a remote façade will be automatically used to export the implementing remote team instances as remote objects, i.e. the remote stubs will be generated from the provided remote façade instead of the GRTI. In addition, the RFBT replaces the parameters of the declared method of remote base class types with parameters of GRBI type, which ensures that the *received* base reference upon an invocation to that method is the corresponding remote stub of the remote base object. Thus, the lifting method could be invoked safely.

- *Conforming Remote Façades to the Java-RMI disciplines*

To implement a remote façade interface compatible to the remote Java-RMI, that interface should either (1) extend the remote interface `java.rmi.Remote`, or (2) extend a user-defined remote interface (or another remote façade) which in turn extends the remote interface `java.rmi.Remote`. If it is found that the remote façade under transformation does not extend any remote user-defined interfaces, then the RFBT adds to the "`extends`" clause the GRTI (i.e. `org.dotj.distribution.IRemoteTeam`), which is already a remote interface. This composition will enable the implicit and explicit access of remote team instances at the same time.

For example, if the remote team class **MyTeam** shown in Figure 5.2 implements the remote façade declared in Listing 5.1, then the application developers can access a specific team instance at the source code from host H2 as illustrated in the following snippet code.

```
IMyTeamFacade iremfac = (IMyTeamFacade) DOTM.getRemoteTeam("…");
iremfac.firstMethod(5,"..", new MyType());
iremfac.secondMethod(new MyBase(), 3.5f);
```

The remote façades are implemented by remote team classes only. Thus, a runtime exception will be raised if a remote façade has been implemented by a class that is not remote team; because the remote façade might be transformed. To make sure that this condition is always satisfied, the RFBT transforms the remote façade interfaces during the transformation of the remote team class which implements that remote façade.

5.4.3 The Remote Base Transformer

The remote base objects generated from remote classes should *exhibit* their original behavior if they *stop* playing remote roles; specifically, when remote team instances are deactivated or become unavailable. Thus, transforming the bytecode of remote base classes must regard this specification. In addition, it should not prevent operations like objects persistency, objects migration from one node to another, and object polymorphism (i.e. casting an object into one of its parents).

In practice, remote base objects need to use two types of communications: (1) communicating with remote team instances in order to realize the remote *playedBy* relationship and (2) communicating with the DOTM in order to obtain, all the time, the latest RTL. To facilitate both communication types, remote base objects need to *provide* two interfaces as illustrated in Figure 5.8 where the remote base class **MyBase** is connected to the remote team class **MyTeam** via the GRBI (`IRemoteBase`), and to the DOTM via the Remote Base Anchor (RBA) interface (named as `IBaseAgent`). The RBA will be used locally by the DOTM to notify the registered base objects for any RTL updates.

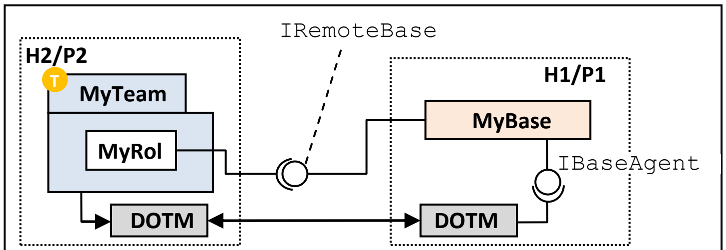

Figure 5.8. Remote base classes provide two interfaces; `IRemoteBase` GRBI, and `IBaseAgent`.

A crucial question is: "*How could it be determined which of remote base class's methods need to be bound to callin methods via remote CIMBs?*" Well, from DADAO's point of view, *all* the methods of specific remote base class are *targets* of the remote CIMB interceptions. Thus, all methods of the remote base classes *must* be transformed and prepared in such a way remote base objects can *anticipate* any remote CIMB interception dynamically at runtime.

The DTL comprises the Remote Base Bytecode Transformer (RBBT), which transforms the remote base classes of application that have been labeled in the XML files. The RBBT uses the algorithm shown in Listing 5.13 to modify these base classes.

- *Preparing Base Classes for being Remote Base Classes*

All the remote base objects generated from remote base classes should be able to *expose* themselves as remote objects according to the Java-RMI disciplines. This goal could be achieved simply once these objects implement the GRBI `IRemoteBase`. Thus, the RBBT adds the `IRemoteBase` GRBI to the declaration clause of all remote base classes by using the BCEL APIs as follows:

```
cg.addInterface(DOTJConstants.iremoteBaseClassName);
```

Listing 5.13: The Remote Base transformation Algorithm – A Pseudo code

Step-1:

__if__ the XML file is not parsed yet, __then__

- *Parse it: scan the file entries and extract the remote base classes and remote façades.*

Step-2:

__If__ the current loaded class is labeled as remote __then__

- *Implement the* `IRemoteBase` *and* `IBaseAgent` *interfaces.*
- *Add the necessary code segments, which enable the remote base object's generation from this class.*
- *Enable the created objects to register their anchors in the DOTM.*
- ***For** each declared or inherited method **do***
  - *Transform the original method*
  - *Dispatch the control flow from this method into the Methods Dispatcher.*
- ***For** each declared or inherited field **do***
  - *Create a new getter method*
  - *Create a new setter method*
- *Add a set of model-specific and internal-usage methods to support remote base objects at runtime.*

Else

__if__ the loaded class is one of the labeled remote façades __then__

- *Transform that interface class using the Remote Façade Bytecode Transformer.*

Afterward, the RBBT injects the implementation of the GRBI's methods into the base class under transformation. The most important method to implement is the GIRM method `_OT$invokeThisMethodInBase`. Recall that the remote base GIRM is used mainly to facilitate methods invocation of remote base objects by the role instances in case of remote COMBs. Listing 5.14 illustrates part of this implementation. Note the `switch` structure at line 4, which guides the control flow to invoke *reflectively* the proper base method according to the value of parameter `_OT$flag` (e.g. if `_OT$flag` equals 1 then a normal base method must be invoked as method-COMB is detected).

In fact, before the desired base method is invoked, the arguments provided in the parameter `_OT$args` are manipulated by the GIRM (not shown in Listing 5.14) in order to un-box the arguments required by the target base method (in case of method-COMB) or the field setter (in case of field-COMB).

Listing 5.14: The implementation of the `_OT$invokeThisMethodInBase` GIRM.

```
public Object _OT$invokeThisMethodInBase(
  String _OT$methodOrFieldName, Object _OT$args[],
  int _OT$flag, String _OT$methodSign)
{
  Object _OT$result = null;
  switch(_OT$flag)// invocation type ..
  {
    case 1: // normal COMB .. Invoke a base method
       :
       :
       _OT$result = getClass().getMethod(_OT$methodOrFieldName,..)
            .invoke(this, _OT$args);
        break;

    case 2: // field setter ..
       :
     getClass().getMethod(_OT$methodOrFieldName,..).invoke(this, _OT$args);
       break;

    case 3: // field getter ..
      _OT$result= getClass().getMethod(_OT$methodOrFieldName,(Class[])null)
            .invoke(this, null);
   }
   return _OT$result;
}
```

The generated objects of a specific remote base class *expose* themselves by generating remote stubs automatically from the GRBI at the construction time (in addition, they register their anchors in the DOTM). To perform both tasks, the RBBT injects at the constructor(s) of remote base classes as the method `_OT$registerBaseAtDOTM()` (as shown in Listing 5.15). At line 4, the base object under creation generates a new remote stub by using the Java-RMI class `UnicastRemoteObject`, and then registers its anchor as depicted in line 6.

Listing 5.15: The method of remote base objects registration in the local DOTM.

```
1. public void _OT$registerBaseAtDOTM()
2. {
3. try{
4.   _OT$theRemoteBaseStub =
           (IRemoteBase)UnicastRemoteObject.exportObject(this, 0);
5.   _OT$dom = DistributedObjectsManager.
               getCurrentDistributedObjectsManager();
6.   _OT$dom._OT$registerRemoteBase("com.myPkg.classes.MyBase",
                                         (IBaseAgent)this);
7.   }
8.   catch(Exception exception){...}
9. }
```

- *Implement the* `IBaseAgent` *interface*

Remote base objects implement the local interface `IBaseAgent` to keep a communication channel opened with the local DOTM. The remote base objects implement this interface to enable the DOTM to *nudge* them in case of remote team instances registration or unavailability.

The *Anchor Interface* `IBaseAgent` declares, basically, the method `_OT$nudge` that will be invoked by the DOTM only to inform base objects that a new change on the RTL contents has been occurred. The RBBT injects the implementation of this method as shown in Listing 5.16. A call to this method will set a *flag* (named as `_OT$RTLexpire`) to "true." This flag works as an ON/OFF switch to represent the status of the consistency of the *local* RTL (i.e. the RTL owned by the remote base class).

Listing 5.16: The implementation of the `_OT$nudge` method.

```
public void _OT$nudge()
{
    _OT$RTLexpire = true;
}
```

The `_OT$RTLexpire` flag is read each time a base object's method is *trapped* (as will be shown later) in order to investigate for any matched remote CIMB interceptions. The "true" value indicates that a new RTL is available in the DOTM; thus, the base object *must* request this new RTL at the next method trapping.

- *Transforming the remote base classes' methods*

The RBBT *reformulates all* the methods of remote base classes in such a way that the control flow is intercepted in these methods at runtime to handle any declared remote CIMBs. To achieve this, the control flow is intercepted in the original method and then forwarded into a *chained* version of that method. The term "*all methods*" points out to both the declared and the inherited methods of remote base classes. The RBBT traverses each method and performs the following modifications:

1. Rename the method by appending the postfix `$orig`[1] to its name. For example, if the base method under transformation named `baseMethod`, then the new name will be `baseMethod$orig`.
2. Create a new method with the original method name and signature, and generate the implementation code as follows:

    a. Pack the arguments provided to this method in an array of objects to box them.
    b. Bring the current Activated Remote Teams List (ARTL) corresponding to this method.
    c. Insert a *trap* invocation to dispatch the control flow from this method towards the *Methods Dispatcher (MsD)*. The MsD (will be discussed the next paragraphs) is a method that is invoked on behalf of the trapped method in order to detect any

[1] This specification is adopted in OT/J.

matched remote CIMBs. Then, it sparks a chain of invocations to execute the remote advices associated with the matched CIMBs according to their modifiers in an accurate precedence. The MsD in DOT/J replaces the *method chaining strategy* adopted in OT/J, in which the interception of a base method sparks a chain of team TLWMs invocations. In case that base class is playing *local* role in which current method is bound to a specific role callin, then the trap invocation is woven into that base method such that *local* role is played first. The remote role could be played if no local roles have been detected in current activated teams.

Therefore, the new version of `baseMethod` method, for example, becomes as shown in Listing 5.17 after transformation. The invocation of the MsD method (named as `_OT$doDispatche`) is shown at line 5, which requires list of currently activated remote teams (ARTL), the name of the intercepted method, its arguments (packed in an array as shown in lines 3 and 4), and its signature, and an index value that determines the level of chaining; precisely, the index of remote team instance currently in hand.

Listing 5.17. The new version of method `baseMethod` after it has been transformed.

```
1. public String baseMethod (float b)
2. {
     // packaging parameters...
3.   Object _OT$args[] = new Object[1];
4.   _OT$args[0] = Float.valueOf(b);
5.   ARTL = _OT$getActiveTeamsList(“baseMethod”, “(F)L..String;”);
     // dispatch control flow to the MD...
6.   return (String)_OT$doDispatch
       (ARTL, “baseMethod”, _OT$args, “(F)Ljava/lang/String;”,0);
7. }
```

In fact, transforming *all* the methods of remote base classes *may* inflate the bytecode of these classes, especially if they implement and/or inherits a large number of methods from its parent(s). For example, suppose that the application developer declares a remote base class (say **MyFrame**) that extends the **JFrame** class (`javax.swing.JFrame`) to provide a simple Graphical User Interface (GUI) for clients. Then, **MyFrame** will inherit about 341 methods from **JFrame** and all of its parents. The RBBT then *must* inject 341 new methods in the **MyFrame** class in order to trap the execution of each of them by the MsD. In addition, it must inject another 341 methods that will delegate the control flow into the parent object (in case that some methods are not overridden).

The experiment has implemented an empty **MyFrame** class (i.e. it defines **MyFrame** to extend **JFrame** only without declaring any methods or fields). The size of the compiled `MyFrame.class` file is *288 bytes* only, while the size after the file has been transformed by the RBBT becomes *133 KB!*, which means that the size of **MyFrame** class is decreased in a percentage of about 472%. Anyway, it should be clear (in most of the cases similar to **MyFrame**) that application classes extend **JFrame** to provide primarily a GUI facility and not to extend the functionality of **JFrame** class itself. Therefore, the inherited methods should not be transformed because they do not form (in this example) a centric part in the base class functionality.

Thus, DOT/J should be able to between the two situations to avert the tremendous increasing of file sizes. In order to *tame* the transformation process of remote base classes, the RBBT has been created to apply different *Transformation Strategies*. Specifically, it uses two transformation strategies:

1. The *INVASIVE* transformation strategy (the default). According to this strategy, the RBBT simply transforms all the declared methods, as well as all the inherited methods.
2. The *NONINVASIVE* transformation strategy. In this strategy, the RBBT transforms only the methods declared by the remote base class under transformation.

The application developers can *apply* any of these strategies *per remote base class* by specifying the desired strategy along the XML entry that labels the base class in the XML file. For example, to apply the “NONINVASIVE” transformation strategy on the **MyFrame** class discussed earlier, it should be labeled in the XML file as follows:

```
<RemoteBases>
  <Base class=”org.remoteClasses.classes.MyFrame” trafoStrategy=”NONINVASIVE”/>
      :
</RemoteBases>
```

However, the implicit “INVASIVE” strategy realizes the transparency that DOT/J framework seeks. In addition, it helps realizing the dynamic adaptability of application objects in case application developers cannot anticipate the entire role playing at the design phase. The “NONINVASIVE” strategy, on the other hand, provides a balanced transformation option, which realizes the dynamic adaptation only for the original functionality of application objects.

- *Inject the implementation of the remote fields’ getters and setters*

To complete the RRP requirements, role instances must be able to de-capsulate their remote base objects via the field-COMBs. For this purpose, the RBBT injects into the bytecode of a remote base class setter and getter methods for each one of its declared or inherited fields independent of their *visibility* (i.e. whether they are private, public or protected fields).

A field setter is a method that changes the value of a specific field to the provided value. For example, if the base class **MyBase** defines the field [`private MyType anyField`], then the RBBT will create a setter method for this field similar to the one shown in Listing 5.18. Recall that the created method should be compatible with the invocation of Listing 5.11 (line 3), which realizes the remote field setter declared in expression (5.6).

Listing 5.18: The field setter method of `anyField`.

```
public void _OT$set$anyField(MyType value)
{
  anyField = value;
}
```

With respect to fields’ getters, the RBBT creates a new method for each field to get the value of that field. For example, Listing 5.19 illustrates the field getter method generated for getting the `anyField` field.

Listing 5.19: The field getter method of `anyField`.

```
public MyType _OT$get$anyField()
{
  return anyField;
}
```

- *Create the Methods Dispatcher (MsD)*

The Methods Dispatcher (MsD) is a centric method that will be injected by the RBBT in each remote base class to trap the calls of its methods. In practice, the MsD is implemented to perform the following tasks:

1. Receive the control flow from *all* the base object methods.
2. Traverse among the ARTL records, and invoke remotely the TLWMs of remote team instance according to the matched CIMB's modifiers.

Listing 5.20 shows a partial *pseudo* implementation of the MsD, which requires the following parameters:

1. The name of the intercepted base method.
2. The required arguments for executing the corresponding TLWM.
3. The intercepted method signature of the base class; in order to facilitate the process of arguments un-boxing.

Listing 5.20: Partial MsD implementation.

```
1.  public Object _OT$doDispatch
     (List ARTL, String methName, Object[] args, String methSig, int dx)
2.  { Object _OT$result = null;
3.    if(ARTL == null || dx ≥ ARTL.length)
4.         return the result of the original base method.
5.    if( ARTL != null) // CIMBs have been matched
6.    {
7.      if( CM.before ){...}
8.      if( CM.replace)
9.        {_OT$result= result of a call to the TLWM}
10.     else
11.       {_OT$result= _OT$doDispatch(ARTL,methName,args,methSig,dx+1);}
12.     if( CM.after ) {...}
13.   }
14.   return _OT$result;
15. }
```

The MsD is designed to precisely *reflect* the mechanism used in OT/J to handle the CIMB interceptions according the accurate order and type. Note at lines 10 and 11 that if no replacement CIMB has been detected to declare by current team's role, then the chaining of interception continues by calling recursively the MsD method with pointing to the next activated team in the ARTL list. This will ensure a precise chaining and dispatching of control flow among activated remote team instances. The MsD will be revisited in Chapter 6 in more detail.

- *Transforming the Remote Façades*

As mentioned earlier, remote façades must be explicitly declared in the XML files at all application nodes that will access remote team instances via these interfaces. Therefore, at each one of these nodes, the declared remote façade interfaces must be transformed according to the mechanism discussed in Section 5.4.2.3 to ensure the compatibility of the implementation of remote façades at all application nodes.

- *Calculate and Inject the Class Serialization ID – CSID*

To avoid the problem of remote base classes versioning (see Chapter 4 – Section 4.5), the RBBT calculates the CSID value for each remote base class. This value is generated automatically prior the transformation activities in order to generate a CSID value *identical* to the one generated by the RTBT for the same base class. The RBBT adds the Java interface `java.io.Serializable` to the "`implements`" clause in every remote base class (if not already implemented), and *stores* the calculated CSID in the field [`private static final long serialVersionUID`] (or overrides the existing value).

5.5 Summary

This chapter has presented in detail the implementation of the DOT/J framework. It has discussed its design specifications and implementation alternatives. Then the chapter has explained through examples the realization of the Remote Role-playing (RRP) along the three phases of the application development. The infrastructure of the DOT/J framework has been presented and studied in detail. Throughout the chapter, the obstacles which faced the implementation have been addressed. The chapter has illustrated how these obstacles resolved. The next chapter will discuss the DOT/J Runtime System, which completes the "big picture" of the DOT/J framework.

Chapter 6

The DOT/J Runtime System

This chapter presents in detail the runtime system of the DOT/J framework. It traces the life cycle of the entities involved in a remote role-playing (RRP) relationship; starting from the point of their creation until the binding and communication between them. This involves the different activities which take place in the RRP relationship like remote callins and callouts execution, and BGPs evaluation. In addition, the chapter describes in detail the components of the DOT/J framework that support remote objects during application execution.

6.1 Introduction

The execution of distributed applications is more complicated than the execution of sequential applications. When executing a distributed application, several requirements and execution environment properties need to be ensured in order to perform precise execution; Security, Availability, Concurrency, and Fault-tolerance are examples of such requirements. Regarding the Remote Role-Playing (RRP) as a programming technique for supporting adaptable distributed applications, execution of these applications requires additional requirements that ensure an accurate performance of the RRP activities.

Basically, the DOT/J framework should support application objects through: (1) helps remote base objects to detect the deployed remote team instances efficiently, (2) ensures a seamless binding between remote base objects and role instances, and (3) makes sure that the remote *playedBy* activities are applied accurately and their effects appear on the target objects. To achieve these primary points, DOT/J should work as a middleware to connect the remote objects of application altogether through RRP. At this point, it should be clear that DOT/J must not participate *directly* in the RRP activities; rather, it should facilitate the *enrollment* of remote objects in these activities. Therefore, this chapter will highlight this fact through examples.

Consider the distributed application shown in Figure 6.1. The figure illustrates that the RRP relationships are connecting several remote team instances and remote base objects on top of the DOT/J middleware. Actually, the application shown in the figure could be considered a typical DOT/J application.

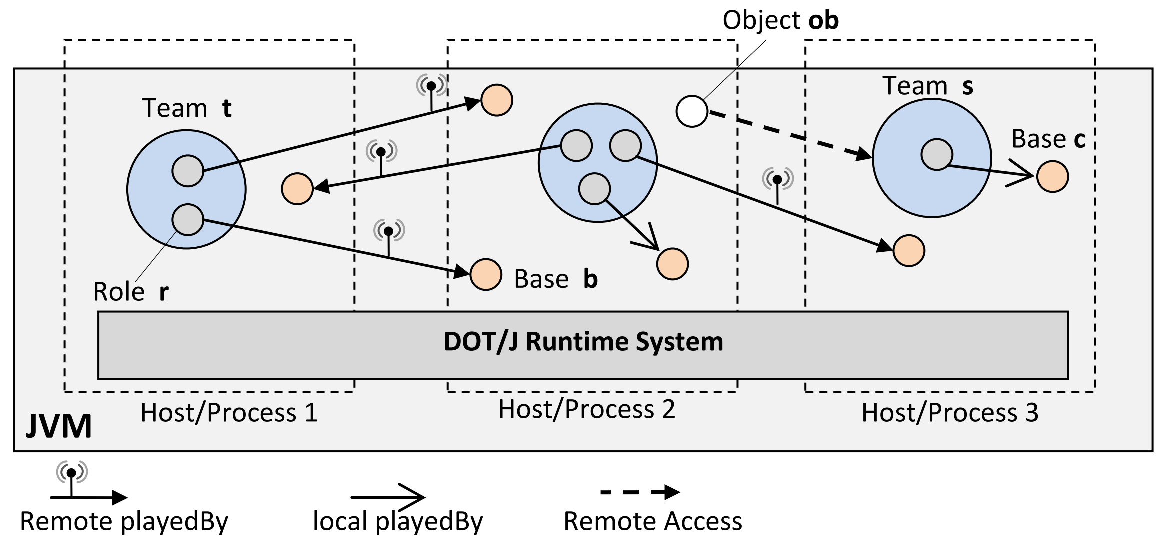

Figure 6.1. The DOT/J Runtime System supports the remote role-playing.

The figure shows different communication types between distributed and local entities. These communications could be distinguished as follows: *first*, remote team instances can confine roles that are played by *local* base objects. For example, in Host/Process 3, base object **c** plays a role in team instance **s**. *Second*, remote team instances could be accessed remotely by a specific object as illustrated in Host/Process 2, in which object **ob** access remote team **s** remotely. *Finally*, a typical RRP relationship is illustrated in the figure between remote base object **b** (in Host/Process 2) and the role **r** which is enclosed in remote team **t** (in Host/Process 1). However, a remote team instance may revolve all these communication types; i.e. it can enclose both local and remote roles, and could be accessed remotely as a distributed component. In all cases, the DOT/J framework resides underneath application objects as a "*distributed layer*" in order to facilitate all these communications. The next sections will discuss how can the DOT/J framework supports all these communications, and the proper DOT/J infrastructure implementation for fulfilling that.

6.2 The Runtime System in DOT/J: Components and Protocol

According to the discussion in Chapter 4 – Section 4.3 and the introduction of this chapter; in addition to achieving a transparent RRP, two primary runtime phases should be considered with respect to the remote objects of a distributed application:

1. *The Registration-Lookup phase*. In this phase, remote team instances register themselves in the DOTM in a proper way; so remote base objects are able to detect them *before* any of base methods is invoked. It is important to mention here that this phase is continuous. That is, new remote team instances can register dynamically in the DOTM at any execution time. This will realize the dynamic playing of roles, which is a primary requirement to accomplish *continuous* application adaptation [20].
2. *The phase of Remote "playedBy" Relationship Activities*. This phase represents the communication between a specific remote base object and a specific remote team instance in the purpose of performing the activities of the RRP, which include: (1)

intercept method calls of the remote base object according to the declared remote CIMBs of the team's roles and executing the associated callin method, (2) execute any demanded field- and/or method-COMBs, (3) evaluate remote BGP expressions. In addition, to perform the remote invocations of public remote team methods.

6.2.1 The Distributed Objects and Teams Manager (DOTM)

The Distributed Objects and Teams Manager (DOTM) is the core component of the DOT/J Runtime System (DRS). Practically, the DOTM component *simplifies* the connection between remote base objects and remote team instances (in particular, those teams enclosing the roles of these objects) before binding, and *supports* them after binding. The DOTM structure consists of three units as shown in Figure 6.2. The Remote Teams Repository (RTR), which is the part responsible for registering remote team instances. The RTR organizes remote team instances in records. Each record holds information about a specific remote team instance (Section 6.2.2 will explain this information along the discussion of the registration process of remote teams in more detail). The purpose of remote teams' registration is to enable the DOTM to replicate them at all nodes, and to simplify tracking them to ensure their availability.

The second unit in the DOTM component is the Remote Bases Repository (RBR). This unit is responsible for storing the "*anchors*" of remote base objects. Primarily, the RBR acts as a *broker* which informs remote base objects in case if a modification has occurred to the Remote Teams List (RTL)-map they hold. Consequently, remote base objects communicate *indirectly* with the RTR via the DOTM to obtain the latest RTL version.

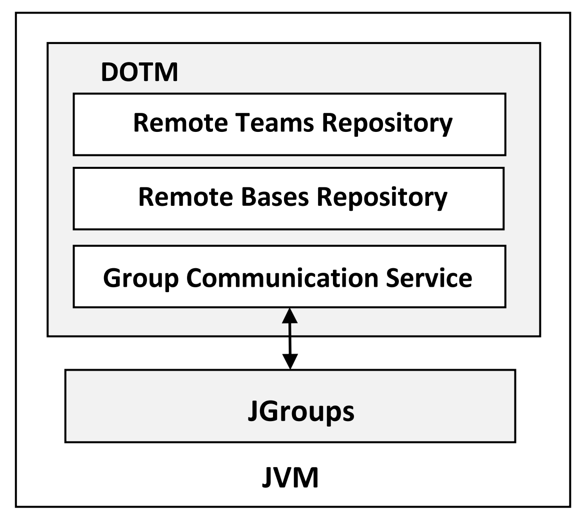

Figure 6.2. Structure of the DOTM.

The third unit, which completes the DOTM structure, is the Group Communication Service (GCS). As mentioned earlier that a single DOTM instance must be deployed at each Host/Process of distributed application in order to facilitate the communication between remote bases and teams objects. But the remote base objects deployed on a specific node need to know the complete list of remote team instances which enclose their roles *independent of* their locations; thus, a precise binding could be accomplished. This means that *all* DOTM components must have *identical copy* of the RTL at any execution time. In other words, the DOTMs must have the capability to communicate with each other to preserve a consistent copy of the RTL all the time. The GCS offers this capability as it acts as a middleware over which the DOTMs communicate. Furthermore, the GCS implements a message-based synchronization protocol using the reliable multicast communication system JGroups [86] (see Appendix-B.2).

To better understand the idea, consider Figure 6.3, which shows the organization of remote base objects and remote team instances inside the DOTMs on two different application nodes. Note from the figure that the DOTMs of Host1/Process1 and

Host2/Process2 have *identical copies* of the RTL (represented in the figure as normal rectangles), but different remote base anchor groups (represented as rounded rectangles).

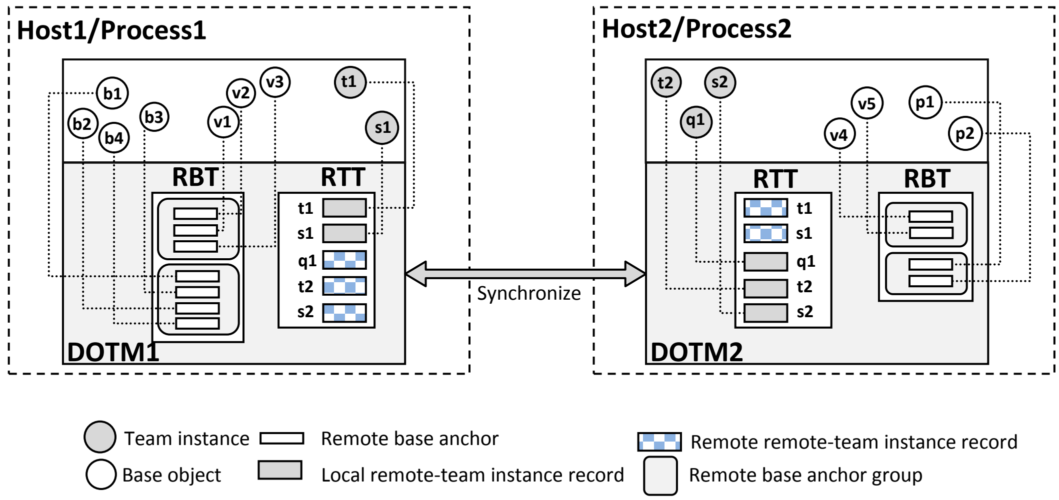

Figure 6.3. Two different DOTMs might hold different remote base anchors, but identical remote teams.

The RTR unit comprises a referential table called the *Remote Teams Table (RTT)*, as Figure 6.3 illustrates. The RTT holds the remote stubs of *all* currently *available* remote team instances that have been created in the distributed application. The RBR, on the other hand, comprises the *Remote Base Table (RBT)*, which holds the anchors of remote base objects that have been created at the host/process in which the current DOTM resides. The RBT classifies the anchors of remote base objects according to the *base class type*. For example, all the anchors of remote base objects of class type **B** will be grouped together and registered in the RBT under the class name of **B**. After then, the DOTM selects an anchor of one of the *available* remote base objects to notify the entire group of with any RTL's changes. Thus, all base objects will obtain the same new RTL copy accordingly. The Remote Base Bytecode Transformer (RBBT) injects at load-time a declaration of a *static* field into each remote base class to store the RTL; thus, all remote base objects of a specific class have the same value.

Before the distributed application is being executed on a specific node, a new DOTM instance will be created automatically in the backstage. Thus, all remote objects will be able to communicate with the created DOTM instance properly. Once created, the new DOTM instance establishes a communication channel using JGroups API [86] to join the group of the DOTMs. In this regard, the DOTM has been implemented so it joins the DOT/J communication group automatically; this group is named as "DOTJ." If the generated DOTM is the first to be created in this group, then it will be considered, according to JGroups specifications, as the group *coordinator*. The group coordinator basically holds *the map* of all members of group, and replicates that map automatically and reliably in case new members/old members join/leave the group. After then, the installation of the DOTM instance is completed. Thus, all remote objects that will be created later on its node can contact it properly.

If the created DOTM is not the first member in the "DOTJ" group, then it allocates the group coordinator by using JGroups APIs, and sends a Teams Table Request (TTR) message to the coordinator in order to obtain the current RTL copy. The creation of a new

DOTM is considered completed if it gets the latest version of RTL. If the instantiation process ends improperly (e.g. due to a runtime exception), then the execution of application on that node will *not* proceed. Once the new generated DOTM obtains the recent RTL, it starts acting *locally* as a broker for both remote team and remote base instances, and acting *remotely* as a member of the "DOTJ" group.

When a new remote team instance is being created, a registration process must take place (discussed in details in the next section) to register that team instance in the RTT. After registration, the mission of the DOTM in which the team has been registered is to *broadcast* this "registration event" to the members of the "DOTJ" group. The broadcast process not only aims to notify all DOTMs with the new team registration, but also to *deploy* the record of that team instance (called *team entry*) on each application node. Thus, all other DOTMs will receive a New Remote Team Entry (NRTE) message, which includes the complete record of the registered team instance. Consequently, all DOTMs will update their current RTL, and start a notification process to remote base classes, which their objects are affected by the update, through their anchors. Here, "affected" directly means those remote base objects which have new remote roles to play inside the new remote team instance.

To explain the idea, consider the example shown in Figure 6.3 once again. When the team instance **s1** is being created at Host1/Process1, it will be registered in DOTM1. Then DOTM1 sends a NRTE message to every DOTM in the group, so they update their RTTs by adding **s1**'s entry. In this case, DOTM2 will receive that message and immediately adds **s1**'s entry to its RTT. If the base objects of classes **P** and **V** at Host2/Process2 are players of specific roles in **s1**, then DOTM2 will notify them to update their RTLs.

The disadvantage of synchronizing the RTL in all DOTMs is that it duplicates the RTT at each node. This means that some DOTMs may preserve remote team entries that will never be used; for example, because the remote base objects on the node of DOTM have no roles to play in these team instances.

6.2.2 The Remote Teams Registration

All the remote team instances created at a specific node must be registered in the RTT of the local DOTM instance. When a remote team instance is created, it obtains the handle of the local DOTM transparently and automatically. Then, it sends a *registration request* by invoking the appropriate API of the DOTM (as explained in Chapter 5 – Section 5.4.2.1). In its registration request, the team instance provides the following information:

1. Its full-qualified class name.
2. The generated remote stub (either from the GRTI or a specific remote façade).
3. The remote CIMB list (called as the Remote Callin Pointcuts (RCP) list)
4. The Remote Team Activation Priority (RTAP) value assigned to the team class.

The first information helps the DOTM to organize teams' entries in the RTT, and makes the process of allocation and management of these entries simple and efficient. The subsequent three values are required by remote base objects. First, the remote stub of team instance will be used to establish the communication between the remote base objects and that team instance. Second, the RCP will be used by the DOTM to prepare the precise RTL for each one of the anchored remote base classes; thus, remote base objects will obtain only the RTL with the remote team instances that enclose their roles. Once remote base objects obtain the RTL, they can know which of calls to their methods will be intercepted with a specific CIMB. Finally, the RTAP value helps remote base objects to determine the accurate order of remote teams' activation; therefore, they regard the CIMB precedence.

When this information arrives, the DOTM immediately dispatches them to the RTT. The RTT creates a new record after it generates a new *serial number* for the registering team instance. Then it stores that record. The serial number represents the index of the registering team instance in the group of similar remote team instances (i.e. team instances that are created from the same team class type). For example, if two remote team instances of type **MyTeam** have been created at two different hosts[1], then the first team instance could be assigned the index "0" and the second the index "1". Finally, a unique identifier – called as *Remote Team Unique Identification (RT-UID)* – is generated and returned to the registering team instance. The RT-UID is composed from the full-qualified class name of the remote team instance and its serial number.

To summarize the remote teams' registration process, consider the UML sequence diagram shown in Figure 6.4, which illustrates the registration of the remote team instance **t1** of type **MyTeam** (the one diagramed in Figure 3.1 – Chapter 3). The registration process is carried out during the construction time. After the team instance **t1** grips the handle of the DOTM, it invokes the registration API on that DOTM supplying the aforementioned information; denoted in the figure by the letters **a**, **b**, **c**, and **d**. A replication process of team's entry is taking place between DOTMs at Host1 and Host2. Finally, a generated RT-UID is returned to **t1**. After **t1** has been registered, the RTT of all DOTMs must look like as the one shown in Figure 6.5.

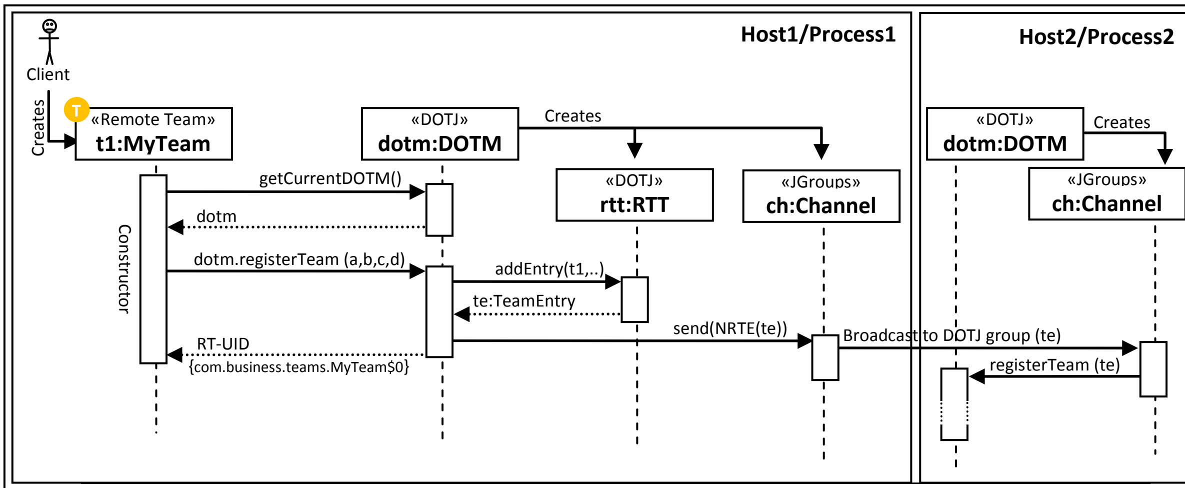

Figure 6.4. A UML Sequence Diagram depicts the remote team registration process.

The team entries of the RTT are added according to last-Register last-Appended (LRLA) strategy. Here, preserving the order of teams' registration is very important in order to resolve the role-playing priority in case that the values of RTAP of two or more team instances found to be equal. Thus, a remote base object can determine the precedence of roles to play easily. The DOT/J framework has been implemented to differentiate between two team instances depending on the order of their registration in the RTT. Therefore, application developer needs to assign RTAP values for remote team classes carefully to

[1] In practice, there is no need to create two remote team instances of the same team type; unless availability or fault-tolerance issues are considered.

avoid the problem of out-of-order role-playing. The RCP list is organized according to the criteria adopted mentioned in Chapter 5 – Subsection 5.4.2.

Remote Teams Table (RTT)			
RT-UID	**Remote GRI stub**	**RCP**	**RTAP**
com.business.classes.MyTeam$0	remGRI: IRemoteTeam	MyBase	1199

CIM	TLWM	BLM	BLM Signature
R	`_OT$MyBase$MyRole$m$m`	m	`()Ljava/lang/String;`
A	`_OT$MyBase$MyRole$role_m1$m1`	m1	`()Ljava/lang/Float;`
B	`_OT$MyBase$MyRole$role_m2$m2`	m2	`()V`

Figure 6.5. A snapshot of the RTT after the remote team instance **t1** has been registered.

6.2.3 The Consistency of the Remote Teams Lists - RTLs

It is very important to preserve the RTT at all DOTMs consistent. The RTT consistency means that all DOTMs must have the *same* copy of the RTT at any time of application execution. In a relevant way, all remote base classes at all nodes must obtain the accurate RTL at an appropriate execution time. To achieve this requirement, a study to those events that directly or indirectly impact the RTT consistency (hence, inconsistent RTLs) must be made. During the execution of application, the following events affect the consistency of the RTT have been encountered:

1. *The registration of new remote team instances*. As mentioned earlier, all new remote team instances must register themselves in the RTT of their local DOTM. At the moment of team instance registration, two situations are considered: *first*, if there is no base object requesting the RTL from a DOTM, neither a specific remote callin method is under execution. In this case, all the RTTs become consistent because all the DOTMs will receive the NRTE message and append the new registering team entry properly. *Second*, if the NRTE message has arrived at a specific DOTM at the moment at which a specific remote role's callin method is executing (with the assumption that the new registering team instance encloses a role to be played by the base object on which the callin has been triggered), then the DOTM re-organizes its RTT and sends a *nudge* message to every affected remote bases through their anchors. This sets the RTL's expiration-flag to "true," which means that the current RTL copy held by affected base objects is expired. After the execution of current callin method is completed, base objects request the new copy of the RTL at the next call of any of their methods. Thus, employing new remote teams will not application stop execution.
2. *Detect the unavailability of a specific remote team instance*. When a specific remote team instance could not be accessed anymore, then it is considered *unavailable*. In practice, several reasons could prevent the accessibility of a specific team instance

like host crash or shutdown, or time-out due to network congestion. However, two relevant issues must be taken into account by the DOT/J Runtime System (DRS) regarding remote teams' availability. First, the mechanism to detect the unavailability of remote team instances. Second, how should it react after the detection? The subsequent paragraphs will discuss both issues in detail. Actually, detecting the unavailability of remote team instances has a direct impact on the RTTs consistency, which might affect application integrity. Therefore, it must be treated accurately.

Remote Teams Unavailability Detection

The *coordinator* DOTM; i.e. the first member in the "DOTJ" group, performs an early detection to remote teams unavailability. For this purpose, it performs *periodic availability check* process. In this process, the DOTM checks the availability of remote team instances by invoking the `_OT$isAlive()` method periodically at every specific amount of time (currently 5 minutes). The unavailability is assured when *three* successive invocations to that method for each remote team instance comes out with a runtime exception[1]. The other situation in which the unavailability of a specific team instance could be detected is when the DOTM receives an Unavailable Remote Team Inquiry (URTI) request from a specific remote base object. The remote base objects send URTI requests by invoking a specific DOTM's API if a specific remote team instance has no longer response during the preparation for remote callin method execution (will be revisited in detail later). The DOTM then performs availability check for the inquired team. If it does not receive any response, then the remote team instance considered unavailable.

In both cases, the DOTM must reorganize its RTT and notifies the other DOTMs with this detection, so they reorganize their own RTTs as well in order to preserve the consistency of the RTLs. More specifically, the DOTM instance that detects the unavailability of a specific remote team instance *removes* the record of that *unavailable* team instance from its RTT, and then it notifies remote bases registered in its RBT to update their RTL copies. At the same time, it sends an Unavailable Remote Team Detection (URTD) message to the "DOTJ" group members. Each DOTM member will receive this message and, accordingly, reorganizes its own RTT by removing the record of the unavailable team instance. Then, it nudges the registered remote bases in its RBT to update their RTLs.

6.2.4 Registration of Remote Bases

Basically, the remote base objects of a distributed application must register themselves each in its local DOTM. The registration of remote base objects differs from that of remote team instances; both in the purpose and the mechanism of registration.

Primarily, remote base objects register in the local DOTM in order to simplify updating the RTL of these objects. Figure 6.6 illustrates the process of registering the remote base object **b1** as a UML sequence diagram. The DOT/J framework facilitates remote base objects registration through using the interface `IBaseAgent`. All remote base objects need to implement this interface; as discussed in Chapter 5. This interface is the anchor of remote base objects in the DOTM; hence, the DOT/J framework.

[1] This method is implemented to always return "true" value. Otherwise, a remote exception is thrown.

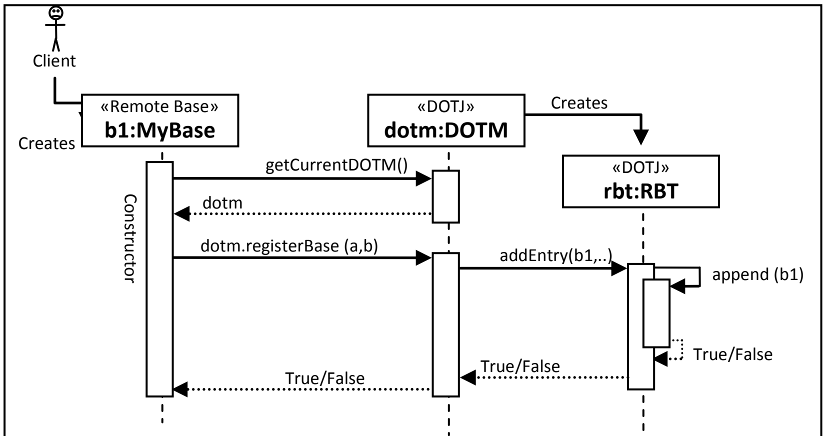

Figure 6.6. A UML Diagram for remote base object registration.

At the construction time, the new created remote base object **b1** grips a handle to the DOTM, and then sends a registration request. It provides its full-qualified class name and the anchor interface it has implemented. The DOTM forwards the request to the RBT as shown in the figure, which creates a new remote base entry, and appends it to the table. If **b1** is the first base object of its class type to register, then the RBT opens a new *Base Record* under which all remote base objects created – at the same node – from the same class of **b1**'s class in the future will be appended. If **b1** is not the first of its class type to register, then the RBT searches for its Base Record, and then appends the *anchor* of **b1**to that record.

The idea of grouping base objects' anchors has been illustrated in Figure 6.3, in which the rounded rectangles represent the Base Records of similar remote base anchors. For example, the RBT of DOTM1 comprises two Base Records; the first is grouping the anchors of base objects of **V** class, and the second one is enclosing anchors of base objects of class **MyBase**. As shown in Figure 6.6, the DOTM returns a "boolean" value (i.e. "true" or "false"), which determines whether the registration process has been performed successfully or not.

6.2.5 Base Methods Dispatching and RTLs Downloading

Once remote team and remote base objects have been registered in the DOTMs adequately, the actual RRP activities can take place. The RRP is triggered at the first invocation of any of remote base object methods, *or* when that base object is passed as an *explicit lifting parameter* to one of remote team's methods. In practice, the first case (i.e. intercepting base method calls by CIMBs) is the common to occur; it is the transparent interception of base objects functionality in all AOP approaches. Therefore, this subsection concentrates on this case. However, binding remote base objects through the explicit lifting declaration will be discussed in Section 6.7.

In DOT/J framework, in order to handle the interception of method calls of a remote base object precisely, that base object must obtain the accurate RTL. The RTL includes the necessary information to spark and perform that interception. At this point, two operations could be recognized; *first*, obtaining the RTL and, *second*, performing the interception. The following paragraphs will merge the discussion of both operations together as they should be carried out correlatively.

The first event in the RRP process takes place when one of remote base object methods is called. Then, the new method code (i.e. after bytecode transformation) is invoked, which forwards the control flow to the Methods Dispatcher (MsD) in order to check for any remote CIMBs interceptions among the current RTL. The called method provides the MsD with its name, the arguments list, and its signature. The MsD, in turn, performs the following tasks sequentially:

1. "Boxing" the arguments of intercepted method according to the provided signature. The Boxing process converts values of primitive type to corresponding values of referenced types [5]. For example, a value of type "`int`" will be converted to the object of class type "`Integer`".
2. The MsD *fetches* the matched CIMBs to the intercepted method call. The list of all matched CIMB is called Activated Remote Teams List (ARTL), and is prepared at the beginning of each base method call according to algorithm of Listing 6.1. The ARTL is organized according to the activation precedence of remote team instances (the RTAP values) and types of CIMBs (their modifiers).

Listing 6.1: Algorithm for ARTL Preparation.

```
Input -   methodName: the name of the intercepted method.
          methodSignature: the signature of the intercepted method.
Output - methodARTL: the list of the relevant remote CIMB of methodName .
Begin
1   if current RTL is null or the RTL is expired then
2        RTL ← DOTM.getRemoteTeamsList( base class name)
3        Set RTL expiration flag to false.
4   Else
5     if the RTL is not empty then
6        for each record in RTL do
7          if remote team instance t is alive and activated then
8            for each remote CIMB record (remCIMB) concerning methodName  do
9              switch (remCIMB.Modifier Type)
10             {
11                After: add  remCIMB  to the after-CIMB-list;
12                Before: append remCIMB to the before-CIMB-list;
13                Replace: add remCIMB to the replace-CIMB-list;
14             }
15         Add t and its lists (before- , replace-, and after-CIMB-lists) to methodARTL.
16    else
17           methodARTL ← null.
End.
```

At line 2 of Listing 6.1, the base object contacts the DOTM in order to *download* the current RTL. For this purpose, base object invokes the API `_OT$getRemoteTeamsList("<base class name>")` of the DOTM, and provides the full-qualified of its class name.

When the DOTM receives an RTL-download request from a specific base object, it forwards the request to the RTT, which generates a list of CIMB records. Each one of these records comprises: (1) the RT-UID of team instance, (2) the remote stub of team instance, (3) the RTAP value, and (4) the Remote Callin Pointcuts list (RCP). The RTT packs the RCP for the requester only according to the provided base class-name. The returned RTL is stored *statically* in each base class so that all base objects of that class obtain the same RTL copy. The RTT prepares the RTL list independent of remote teams' activation status.

If the returned RTL is not empty; i.e. there are roles for the base object in hand to play, then it will be traversed by the base object to prepare the ARTL of the dispatched method. Note from line 7 of Listing 6.1 that only those CIMBs in team instances which are *currently alive* and *activated* will be packed in the ARTL. Lines 11, 12, and 13 of the listing illustrate how the ARTL records are organized.

3. If the generated ARTL is not empty, then the MsD will spark a chain of invocations to execute the remote callin methods of roles corresponding to the RCP's records. The MsD invokes the Team Level Wrapper Method (TLWM) associated with each remote CIMB in the ARTL. However, if the returned ARTL is empty (i.e. no remote CIMBs intercept the base method call), then the *original* method will be invoked instead using the Java Reflection API. If an exception occurs during the interception of base method call (e.g. unavailable team instance has been detected), then RTL expiration flag is raised. In this case, in any sequent call, the MsD will fetch a new RTL from the DOTM as illustrated at 1 line of Listing 6.1.

6.2.6 The Dynamic Updating of the RTL

The DOTM plays a primary role to support remote base objects after they have registered in the RBT. Specifically, the DOTM ensures that all the remote base objects registered in the RBT will always obtain the latest RTL version. As mentioned earlier, a remote base object needs to update its RTL in two situations: when a new remote team instance has been registered, and when the unavailability of a specific remote team instance is detected. To explain these situations and their consequences on the RRP activities, consider Figure 6.7, which illustrates a UML sequence diagram for the communication between the remote base instance **b1** of type **MyBase** and the DOTM. At the same time, the remote team instance **m1** of type **MyOtherTeam** is registered. After the registration of **m1**, the current RTT of the DOTM should comprise – in addition to remote team instance **m1** – the remote team instance **t1**, which has been registered earlier, see Figure 6.4.

After the DOTM has registered **m1** and broadcasted its entry to all other DOTMs (not shown in the Figure), it delegates the RBT to "nudge" all the affected remote base objects to update their RTLs. Suppose that the current RBT has registered 100 anchors for remote base objects of type **MyBase**. Then only one anchor for a valid base object is selected and used to induct the RTL update for **MyBase**. A "valid base object" means an active base object that has not yet been "garbage collected." The RBT checks this validity by invoking the anchor `_OT$isAlive` method transparently on base objects. When a client invokes the base method "`n`", the new RTL copy will be re-fetched as the expire flag is raised.

The second event, which compels remote base objects to update their RTLs, is the unavailability detection of a specific remote team instance (shown in part **B** of Figure 6.7). While the base object **b1** is preparing the ARTL at the call of its method "`m`," it discovers that the remote team instance **t1** has no longer responded. Then, **b1** sends immediately a new URTI request to the DOTM. Meanwhile, **b1** waits for a half second (500 ms) and then attempts to contact **t1**. If it fails to get a response from **t1,** it drops it from its own RTL. The DOTM, on its side, checks the availability of **t1**, and if it fails to get a response, it removes **t1**'s entry from its RTT. Then it notifies all other remote bases except **MyBase** to update their RTLs, and broadcasts the "removal process" by sending a new URTD message to the "DOTM" group.

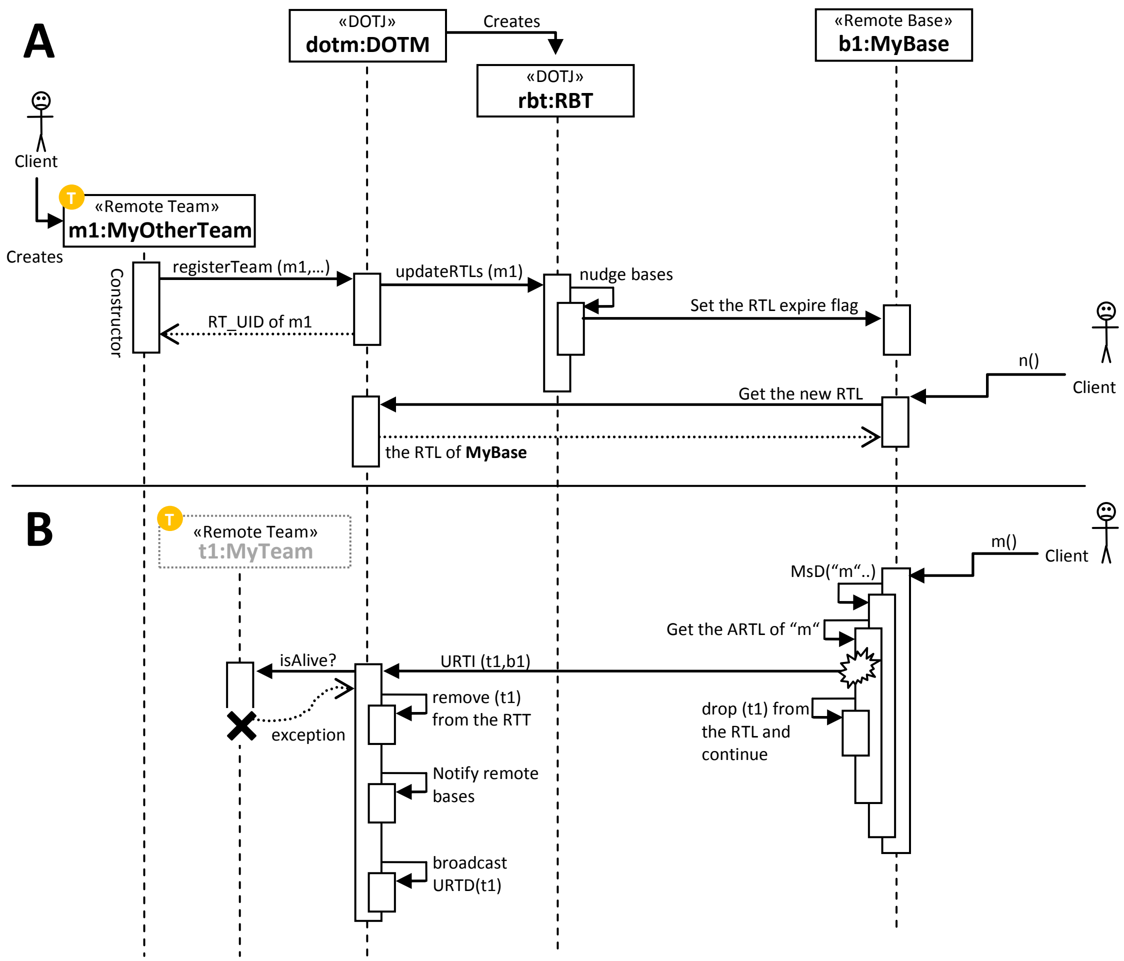

Figure 6.7. A UML Sequence Diagram for the dynamic updating of RTLs.

6.3 The Remote Teams Activation Mechanism

In the RRP, it is important to distinguish between two main processes: *first*, the binding of a role instance to a remote base object, and *second*, the communication between them to perform the remote *playedBy* relationship activities; i.e. executing the remote callins and COMBs, and evaluating the BGPs. The binding process does not require the remote team instance which encloses the role instance to be *activated*. But, the applicability of remote *playedBy* relationship, on the other hand, totally depends on the activation status of the remote team instance. That is, all the declared remote *playedBy* relationships in a specific team instance are disabled if that instance found to be deactivated.

In DOT/J framework, a remote team class *can* declare both *local* and *remote* roles. Thus, a conformance between local and remote role-playing with respect to the activation status of remote team instance at runtime must be ensured. Recall that a team instance in OT/J could be activated and deactivated through three different mechanisms (see Chapter 2 – Section 2.2.6). In practice, remote team instances (as Java objects) must reside in a *local*

Thread-based environment [5]. In fact, Java-RMI system does not support yet distributed threads. Therefore, *all* remote team instances must be activated and deactivated *globally*. In other words, the DOT/J Runtime System (DRS) requires that remote team instances *must* be activated for all threads (including the threads of the Java-RMI proxies) so that their activation status could be read remotely by remote base objects. For this reason, application developer needs to explicitly or implicitly activate remote team instances using the method `activate (thread)`, providing the OT/J's constant `Team.ALL_THREADS` to that method to indication achieve a global activation. To perform a global deactivation, application developer needs to invoke the method `deactivate (Team.ALL_THREADS)`.

6.4 Executing the Remote Roles' Callin Methods

This section, and the subsequent two sections, will discuss in detail the main activities which take place during the RRP between a remote base object and a remote role instance. In particular, it will discuss how remote CIMBs, remote COMBs, and remote BGPs are handled at runtime? A remote CIMB has a distinctive importance because it is one of two mechanisms for binding remote base objects to role instances. In addition, the remote CIMB is the key concept to accomplish the dynamic adaptation of application objects.

Basically, the phrase "execution of a remote CIMB" will be used to refer to the process of matching a remote CIMB declaration and executing the role's callin method associated with that CIMB. The process is triggered at the remote base objects; specifically, when a call to one of their methods is intercepted by one or more remote CIMBs. As mentioned earlier, all calls of a base object's methods are intercepted by the Methods Dispatcher (MsD), which takes the control over execution of remote CIMBs. To illustrate the complete story of a remote CIMB execution, the following paragraphs will trace step by step a specific base method invocation, and illustrate the interception of this call by a specific remote CIMB. In addition, the subsequent operations, which are taking place behind the scene, in the DRS in order to complete the remote CIMB execution will be shown.

Suppose that the base class **MyBase** plays two remote roles, namely: **MyRole** and **MyOtherRole**, which are confined in **MyTeam** and **MyOtherTeam** team classes, respectively. Figure 6.8 demonstrates the relationships between the remote base class and these remote roles. The figure shows the remote CIMBs that are declared by the role class **MyOtherRole**. In total, the list of declared remote CIMBs is the one shown in Figure 6.8, and the remote CIMBs of the role class **MyRole** which were declared in Chapter 5 in expressions 5.2, 5.3, and 5.4 and are listed below to simplify the explanation:

```
String m() ← replace String m();
role_m1 ← after m;
role_m2 ← before m;
```

To complete the scenario, the remote team class **MyTeam** will be assumed to has the RTAP value 10, and that of the team class **MyOtherTeam** is 20 (i.e. instances of **MyOtherTeam** team class have a higher activation priority over **MyTeam**'s instances).

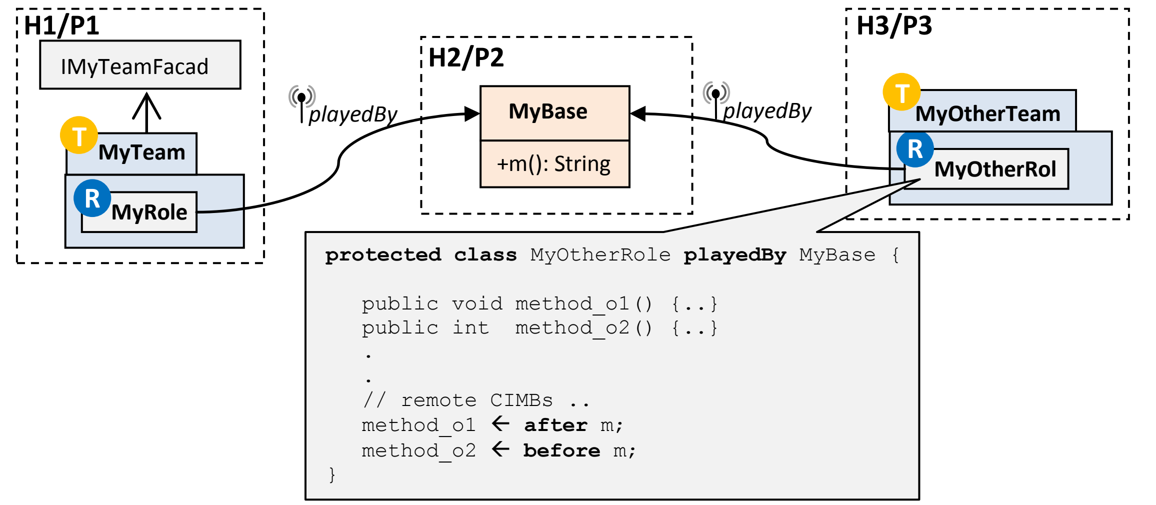

Figure 6.8. The remote base class **MyBase** plays the remote roles **MyRole** and **MyOtherRole**.

Suppose that the user of application invokes the method `m` (e.g. `[s = b.m ()]`, where `b` is an instance created from the remote base class **MyBase**), then the call will be dispatched into the MsD. Figure 6.9 illustrates the complete interception process as a UML sequence diagram starting from calling method `m` until the result of execution is returned and stored in the variable `s`. Note that the figure out-stretches the execution of remote CIMBs (enclosed in a rectangle) each in individual execution sequence to clarify the idea; actually, the MsD perform that in a loop with exceptions management. The ARTL of method `m` is shown in the table enclosed inside the callout shape. Note that in case of executing a remote *after* or *before* CIMB, the returned result (if any) will be ignored by the MsD. For simplicity, the GRTI of the remote base object **b1**, the GRTI of the remote team instance **t1**, and the remote façade of the remote team instance **q** are removed from the figure. That is, the control flow is mutually exchanged among the remote stubs of remote objects.

It is important to note the absence of the DOTM during the execution of remote CIMBs, which is true because the RRP is a direct relationship between remote base objects and remote roles (via team instance). The task of DOTM is limited to support this process *transparently*. Though, DOTM has main role, for example, in case of unavailability detection of a remote team instance.

6.5 Evaluating the Remote Base Guard Predicates – BGPs

The Guard Predicates (GPs) are conditional expressions used in OT/J applications primarily to control the effects of callin bindings at runtime. If one of these expressions comprises at least a method call or a field access of a specific base object, then the expression is called Base Guard Predicate (BGP). Regarding remote BGPs, they could be evaluated straightforward. This section explains how a remote BGP could be evaluated at runtime.

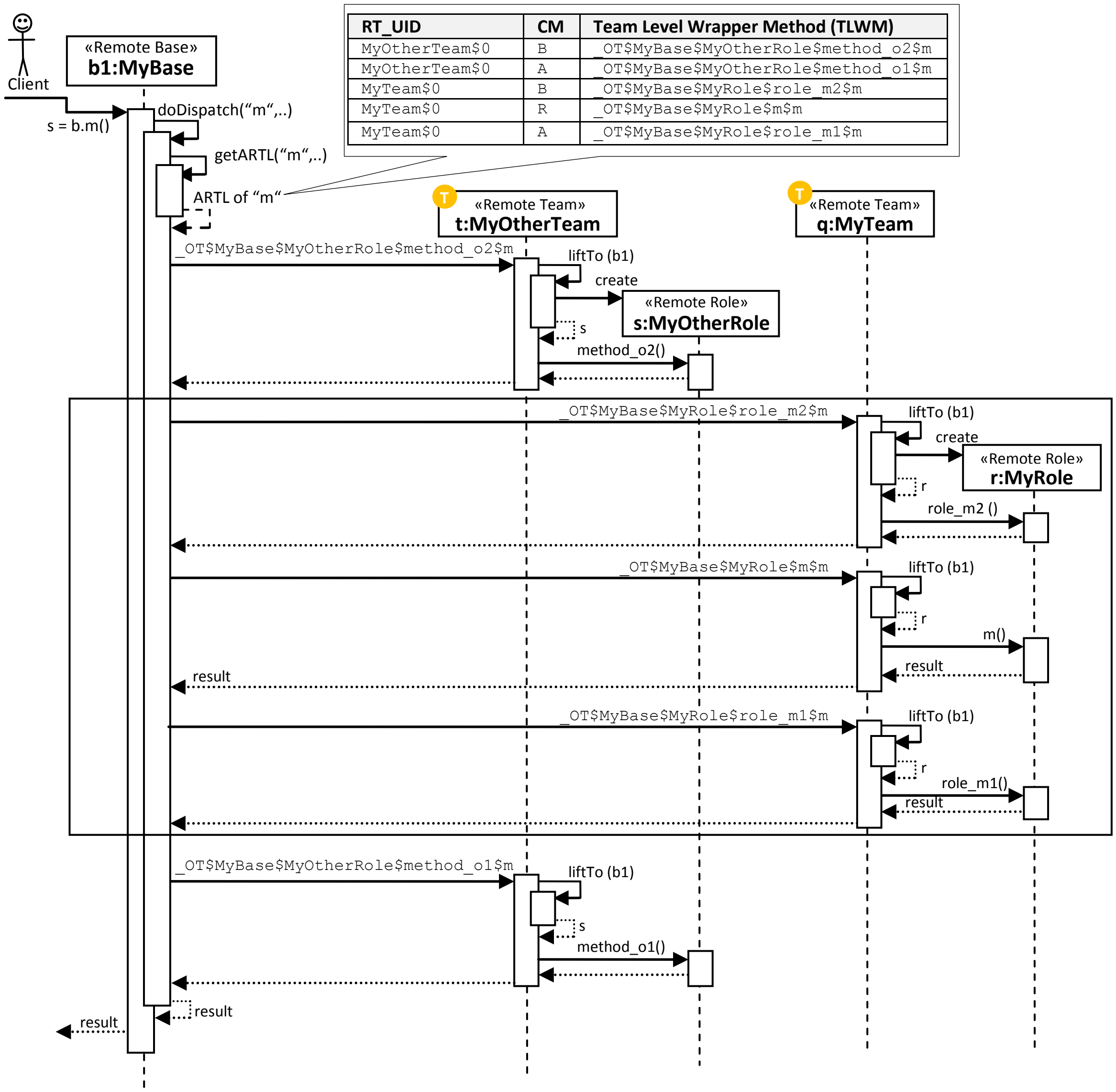

Figure 6.9. Intercepting a call to method `m` of the remote base object **b** by remote CIMBs.

Consider, as an example, the BGP method `_OT$base_when` (in Listing 5.7 – Chapter 5, Subsection 5.4.2.2) after it has been transformed by the DTL transformers. Recall that this method was generated by the OT/J transformers to realize the BGP expression 5.5 listed in the same subsection. When the base method `m` invoked, it will be intercepted as explained earlier. The first remote CIMB of role **MyRole** is declared with the modifier "`before`" – see the ARTL shown in Figure 6.9. Then, the remote BGP will be evaluated as shown in Figure 6.10. The control flow is forwarded from the role instance **r** into the base object **b1** in order to invoke the method `staticMethod` (as part of the BGP expression) by calling the Generic Invocation by Reflection Method (GIRM) on the exposed GRBI stub of **b1**. At this point, the Java-RMI System blocks the role instance **r** (suspends its execution), and unblocks it once the result of invocation is returned. The remaining parts of remote BGP evaluation are performed in the same way; i.e. by invoking the GIRM method as necessary (not shown in the figure). For example, the GIRM method will be invoked once again to get the value of the field `fieldValue` via calling the field-getter of the `fieldValue` field.

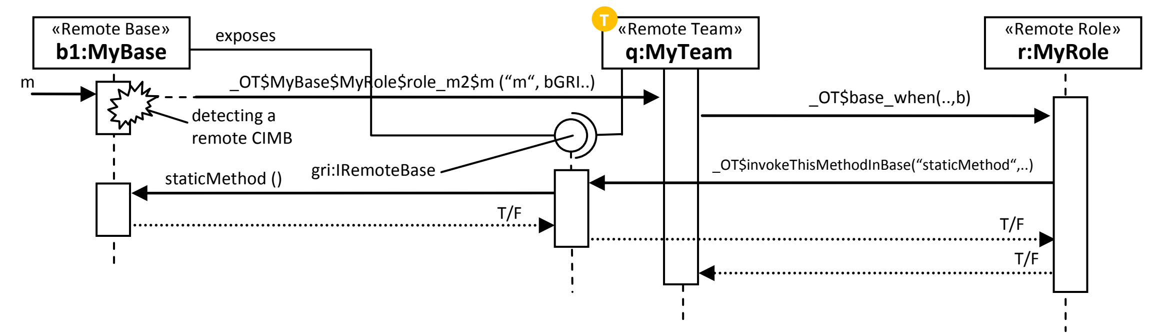

Figure 6.10. A UML sequence diagram for evaluation of a remote BGP expression.

6.6 Executing the Remote COMBs

Executing the remote COMBs is a straightforward process. A role instance can access the methods and fields of its base object via remote COMBs.

However, the execution of the remote COMBs in expressions 5.6 and 5.7 (which respectively declare field getter and method COMB) – see Chapter 5 – Subsection 5.4.2.2 – will be illustrated. In Figure 6.11, the UML sequence diagram illustrates how the role instance **r** executes these two remote COMBs once its method `ARoleMethod` (shown in Listing 6.2) is invoked.

Listing 6.2: The Role method `ARoleMethod` invokes two remote COMBs.

```
1.public void ARoleMethod()
2.{
3.   MyType f = getAnyField();
4.   int res = roleCallOut(f.getv(), 5);
5.}
```

At line 3 of Listing 6.2, the control flow is directed to invoke the expected role's method `getAnyField` (created by the OT/J compiler). As Figure 6.11 illustrates, the

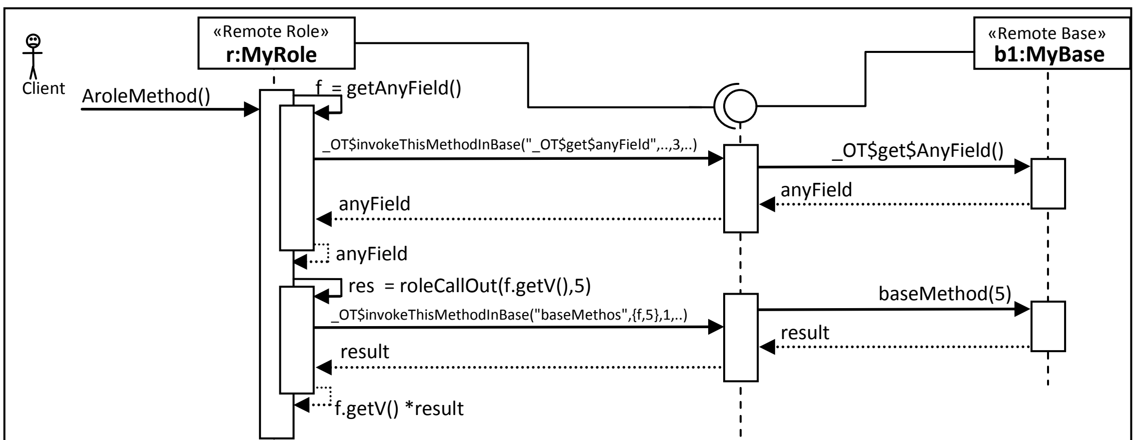

Figure 6.11. A UML sequence diagram for executing the remote COMBs of field getter and base method invocation, respectively.

invocation of this method is converted by the DTL transformers into an invocation to the GIRM method `_OT$invokeThisMethodInBase` with the *flag* value “3” to indicate that a field-getter invocation must be considered by the remote base object **b1**. The same mechanism is followed when executing line 4 of Listing 6.2 as depicted in the figure.

6.7 Access Remote Team Instances as Distributed Components

So far, remote team instances have been discussed and accessed only as containers of role objects. A remote team instance, however, can have another characteristic; i.e. remote team instance as distributed component[1]. This section will discuss how developers can employ and access remote team instances at the source code level of application. This facility will add more flexibility to the implementation of distributed applications in the DOT/J framework.

In this regard, the DOT/J framework enables application developers to access the remote teams via two mechanisms: (1) the GRTI remote interface, which declares a set of generic APIs, or (2) a user-defined Remote Façade, which can declare a selective set of team’s methods. In this subsection, the use of both mechanisms in detail will be explained.

6.7.1 Access of Remote Teams via the GRTI

In a running distributed application, the DRS allows application developers to *grip* a handle to any generated remote team instance at any node. The realization of this specification is achievable because *all* remote team instances are compelled to register each in its local DOTM. The DOTM, then, broadcasts the team entry to all other DOTMs. Therefore, it is easy to obtain the remote stubs of remote team instances.

The access of a specific remote team instance, then, implies the capability to invoke any of its *public* methods remotely. To illustrate the idea, suppose that the application developer has created at host H1 an instance of the remote team class **MyTeam** (see Figure 6.8), and that **MyTeam** implements the arbitrary public method `String doSomething(int a, int b, MyType m)`. Then, this method could be invoked from another different host of the application (say H4) as demonstrated in the code snippet shown in Listing 6.3. At line 1, a handle to the local DOTM is gripped first. Then, line 2 invokes the API method `getRemoteTeam` on the gripped DOTM instance **d**. This method returns a remote stub of

Listing 6.3: Invoke the remote team’s method `doSomething` via the remote team GRI stub.

```
1. DistributedObjectsTeamsManager // grip a handle of DOTM
   d = DistributedObjectsTeamsManager.getCurrentDistributedObjectsTeamsManager();
2. IRemoteTeam remT =(IRemoteTeam)d.getRemoteTeam(“com.business.classes.MyTeam”);
3. String res = null;
4. if(remT != null){
5. res = (String)remT.remoteCall(“doSomeThing”, new Object[]{6,8,new MyType()});
6. }
```

any available remote team instance of type `com.business.classes.MyTeam`. The method could return a remote stub of an *activated* or *deactivated* remote team instance. To return a remote stub of an activated team instance, the DOTM implements another API function (see Appendix-A for more details).

[1] The discussion of distributed component models in detail is out of the scope of this dissertation.

However, line 5 shows a call to the method `remoteCall`, which is declared by the GRTI to allow explicitly invoking any public method of a specific remote team instance. This API requires the name of the desired method to call and the list of arguments which should be packed in an array of objects. For example, line 5 illustrates how the arguments `6`, `8` and the object of `MyType`, respectively, have been organized in an array of objects.

This mechanism is adequate if few access operations to remote team instances are required; actually, it is inconvenient to pack the list of arguments of methods manually each time a method is called. Moreover, the developer of application needs to, for example, cast the returned value accurately. In addition, the returned primitive values need to be "un-boxed" manually; because the API method `remoteCall` returns an object of type `java.lang.Object`.

6.7.2 Access of Remote Teams via Remote Façades

Remote Façades are remote interfaces that could be implemented only by remote teams in order to simplify access their instances remotely at runtime. Normally, a remote façade comprises the team's implemented methods which need to be invoked remotely. Conversely, the GRTI is implemented implicitly by remote teams, and imposes a single way of access.

By using remote façades, the remote accessibility of team instances becomes easy and more modular. Chapter 5 has explained how application developers can implement remote façades in few steps. Suppose that the method `String doSomething(int a, int b, MyType m)` has been declared in the remote façade **IMyTeamFacade**, which is implemented then by **MyTeam** team (as shown in Figure 6.8). At runtime, this method can now be easily invoked as shown in line 4 of Listing 6.4. Also, the method's arguments are passed without any further manipulation. Line 2 grips the remote stub generated from the remote façade **IMyTeamFacade** for the remote team instance of **MyTeam** class instead of the GRTI stub. In addition, the application developer does not have to cast the returned value (as shown at line 4).

Listing 6.4: Invoke the remote team's method `doSomething` via remote façade.

```
1  DistributedObjectsTeamsManager d = // grip a handle to DOTM
   DistributedObjectsTeamsManager.getCurrentDistributedObjectsTeamsManager();
2  IMyTeamFacade mtf =(IMyTeamFacade)d.getRemoteTeam("com.business.classes.MyTeam");
3  String res = null;
4  if(mtf != null){res = mtf.doSomeThing(6,8,new MyType()); }
```

The disadvantage of adopting remote façades is that application developers need to announce these façades in *all* application nodes (i.e. inside the XML labeling files); because the remote teams which implement these façades will be registered in the DOTM with the remote stubs generated from the remote façades and not the GRTI. This means that remote façade classes should be loaded into all JVMs of a distributed application.

6.8 Securing the Remote Role-playing – RRP

The DOT/J framework implements the RRP, which is a remote relationship between the physically or logically separated remote base objects and remote team instances. Therefore, the DOT/J framework is requested to provide a *firm* intermediation for distributed objects in order to enable them to establish a secure relationship. Accordingly, remote base objects can accurately play remote roles without shredding their original functionality, and can reclaim this functionality if, for example, a remote team instance became unavailable.

Basically, the DOT/J framework has addressed two levels in which the security and accuracy issues must be regarded in order to preserve the integrity and consistency of application objects. These levels are:

1. *The Bytecode transformation level*. At this level, the DOT/J framework must answer the following question: "Did the DTL transformers transform the bytecode of remote team, remote role, and remote base classes accurately?" A precise bytecode transformation of remote classes is achieved if the satisfaction of the following issues has been ensured:
    a. Methods of a specific remote base class should be intercepted by the declared local and remote CIMBs only. In this direction, remote base classes are transformed by the DTL such that their methods are dispatched to the MsD method. The MsD is a central GIRM method implemented in each base object. Thus, the MsD is the only *authorized* method to detect the matched remote CIMBs, and perform remote callin invocations.
    b. A remote team instance must always bind the accurate and valid remote base objects with appropriate role instances, and must be able to *lift* each bounded base object to its exact role instance every time. To fulfill this requirement, the DTL transformers inject the necessary code segments into team classes that enable their instances to *check* the validity of the remote base object under binding before it has been actually bound. The DRS, on the other hand, ensures that the base object under binding has the accurate class name (mentioned after the "`playedBy`" keyword) and the *assured* serialization ID, which are the required values for a remote team instance to validate the base object. Both values are known by both remote base and team instance at runtime.

       Note that two different remote base objects compulsory implement the same GRBI interface, but each one of them "exported" different remote stub. Practically, the Java-RMI Runtime System creates at the *client-side* new instance from the `Proxy` class, which is created at fly. Therefore, different proxy objects will be created at the same team side. At the end, role instances will be bound to distinctive base objects.
2. *The Runtime Communication level*. At runtime, three different communication paths could be opened to support the different types of interactions between remote objects of a distributed application (see Figure 6.1). *First*, a transparent channel between remote base objects and remote team instances should be opened in order to perform the remote CIMBs and COMBs executions, and to evaluate the remote BGPs. *Second*, a communication path is required to enable the DOTMs of the "DOTJ" group to preserve the consistency of the RTL. The *third* channel is required to access explicitly a specific remote team instance.

   At this level, the DOT/J Runtime System (DRS) uses the Java-RMI system to perform these communications as remote method invocations (in specific, through the GIRM methods). The Java-RMI system, in turn, imposes a *Security Manager* [83] to be installed before it starts marshalling/un-marshalling and transferring data between application nodes. The security manager is used to provide a secure interaction at runtime, and it is controlled by defining a set of *permissions* (relevant to application resource usage; e.g. read/write files and connect/close communication channels (sockets)), which are stored in *security policy* files.

   Furthermore, remote team and remote base objects expose their remote stubs via Java-RMI system, which enables the generation of stubs by using a customized *socket* *factory* (since Java 1.2) [83, 6]. For example, socket factories to generate the remote stubs of base and team instances could be customized by embedding the SSL/TSL security layer in order to secure the communication channels at runtime [62].

6.9 Summary

This chapter has presented the concept and specifications of the DOT/J Runtime System (DRS). The chapter explained through examples the different activities involved in the RRP relationship step by step. In addition, it has described in detail the infrastructure of the core components in DRS; specifically, the Distributed Objects and Teams Manager (DOTM). It has discussed the different tasks this component has been dedicated to perform. Finally, the security issue has been discussed in brief. This chapter has completed – with the previous two chapters – the “big picture” of the DOT/J framework.

Chapter 7

DOT/J Case Studies with Evaluation

In this chapter, the DOT/J framework is evaluated. Specifically, the chapter verifies that the remote CIMBs declared by remote roles are woven and employed in distributed applications accurately. In addition, it verifies that the other features of OT model have been mapped to distributed environments properly; including: remote COMBs, remote BGPs, the accurate remote teams' activation order, and the unavailability detection of remote team instances. The chapter develops several case studies to demonstrate the applications of DOT/J, and introduces a performance analysis for each of them. The chapter compares the performance of the DOT/J approach against AWED via a case study implemented with both approaches. The developed case studies include: distributed aspects, supporting the DADAO, and a transparent white-box approach for supporting legacy applications collaboration.

7.1 Verification of Remote CIMB, Remote COMBs and Remote BGPs

In this section, an arbitrary OT/J application will be developed. The mechanism to map the application to a distributed environment using the DOT/J framework will be explained. This is to verify that remote CIMBs are accurately deployed and be able to modify the behavior of remote base objects. Also, the section will ensure that remote COMBs and remote BGPs could be handled precisely.

7.1.1 Case Study: A Simple Distributed OT/J Application

Consider the arbitrary application shown in Figure 7.1, which depicts in part (a) the base class **MyBase** playing the **MyRole** role in the team class **MyTeam**. Part (b) of the figure shows an arbitrary "deployment plan" for the application. Then, two base objects of class **MyBase** have been deployed on nodes H1 and H2, and a team instance of **MyTeam** has been created at H3 as the figure illustrates. The implementation of this application will not be listed; a complete implementation could be found in Appendix-C.1.

The role class **MyRole** declares the following remote CIMB, which has the effect of replacing the base method `whoAmI` with the role callin method `whoAmI`:

```
whoAmI <- replace whoAmI;                    (7.1)
```

In addition, it declares a remote field-getter to get the value of `myID` base field as follows:

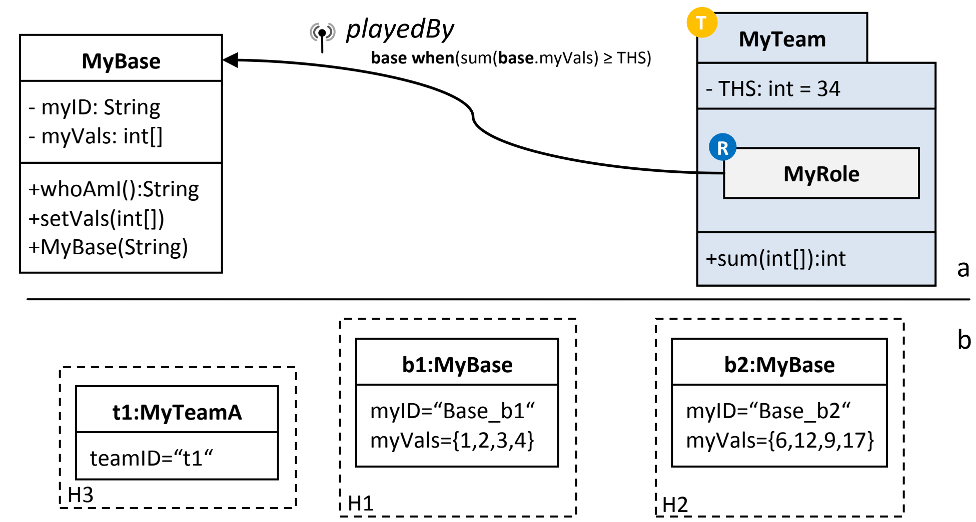

Figure 7.1 A simple distributed OT/J application. In (a) the remote *playedBy* relationship binds **MyBase** with **MyRole**, and in (b) a deployment plane.

```
String getBaseId() -> get String myID;
```
(7.2)

The base class **MyBase** implements method `whoAmI` as follows:

```
public String whoAmI()
{
   return "Instance of MyBase class";
}
```

The role's callin method `whoAmI` is implemented as follows:

```
1. callin String whoAmI()
2. {
3.   String postFix = RoleA.class + " in team " + MyTeamA.class;
4.   return base.whoAmI()+ " ID=" + getBaseId() + ", play role" + postFix;
5.  }
```

At line 4, the base-call "`base.whoAmI()`" will issue a remote COMB to invoke the base version of method `whoAmI`. At the same line, a call to the role method `getBaseId()` is issued to get the value of field `myID` of the bound base object as stipulated in expression (7.2).

The *playedBy* relationship is constrained with a BGP as shown in the figure. The expression requires the value of `myVals` field of the base object under binding. The field type is an array of integers, and an implicit remote COMB field-getter will be issued to get that field.

Now, this application could be mapped to the deployment environment of Figure 7.1 (b) as follows:

1. *Label the remote classes:*

At hosts H1 and H2, developers can label **MyBase** class as a remote base class as follows:

```
<?xml version="1.0"?>
<DOTJ>
  <RemoteBases>
    <Base class="de.evaluation.c1.bases.MyBase"/>
  </RemoteBases>
</DOTJ>
```

At host H3, **MyTeam** and its role **MyRole** could be labeled, respectively, as remote team and remote role classes as follows:

```
<?xml version="1.0"?>
<DOTJ>
 <RemoteTeam class="de.evaluation.c1.teams.MyTeam">
   <RemoteRole name="MyRole"/>
 </RemoteTeam>
</DOTJ>
```

2. *Execute the team application deployed on H3:*

The team class **MyTeam** declares the method `main`, which creates an instance of this team, and enables user to switch between activation and deactivation of this instance as follows:

```
while(true)
{
 (new BufferedReader(new InputStreamReader(System.in))).readLine();
  if(t1.isActive(ALL_THREADS))
   t1.deactivate(ALL_THREADS);
  else
   t1.activate(ALL_THREADS);
  System.out.print("-Activated? -" + t1.isActive(ALL_THREADS));
}
```

3. *Execute the base applications deployed on H1 and H2:*

The method `main` of base applications at H1 and H2 creates a remote base object of class type **MyBase**. It enables the user to invoke the method `whoAmI` on that base object. In this experiment, two modes to invoke this method will be used:

I. *Discrete invocation*: in this mode, clients invoke the method `whoAmI` after they press the “Enter” key. In this way, the activation status of team instance is ensured before the base object plays its role.

II. *Intensive invocation*: the method will be invoked 1000 times continuously without any delay. The experiment will be invoked 3 times at each one of hosts H1 and H2 while keep the team instance on H3 activated.

The following code snippet illustrates part of the `main` method implemented by base application at node H1, which depicts mode-I:

```
while(true)
{
  (new BufferedReader(new InputStreamReader(System.in))).readLine();
   s = b1.whoAmI();
   System.out.println(round + ":[" + time + "]: I am " + s);
}
```

To emphasize the effect of remote CIMB expression declared in expression (7.1) on base objects, the method `whoAmI` has been invoked *before* the team instance is activated. Then, in mode-I, after invoking `whoAmI` method at H1 and H2 for 2 times, the remote team instance on H3 has been activated. After that, the method has been invoked 10 times at each node. In mode-II, all invocations to `whoAmI` method are made after the remote team instance is activated.

7.1.2 Results of Execution

Mode-I: After executing the applications at hosts H1 and H2, the outputs shown in Figure 7.2 have been recorded.

```
Case Study 1 - Node : H1
 #Press ENTER to know who am I?

1:[16:16:20:21]: I am instance of MyBase class
2:[16:16:20:464]: I am instance of MyBase class
3:[16:16:47:388]: I am instance of MyBase class
4:[16:16:49:752]: I am instance of MyBase class
5:[16:16:51:400]: I am instance of MyBase class
6:[16:16:53:416]: I am instance of MyBase class
7:[16:16:54:937]: I am instance of MyBase class
8:[16:16:55:70]: I am instance of MyBase class
9:[16:16:55:680]: I am instance of MyBase class
10:[16:16:55:812]: I am instance of MyBase class
11:[16:16:55:995]: I am instance of MyBase class
12:[16:16:56:178]: I am instance of MyBase class
```

a

```
Case Study 1 - Node : H2
 #Press ENTER to know who am I?

1:[16:16:32:828]: I am instance of MyBase class
2:[16:16:37:712]: I am instance of MyBase class
3:[16:17:09:360]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
4:[16:17:10:421]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
5:[16:17:11:492]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
6:[16:17:12:743]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
7:[16:17:15:756]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
8:[16:17:19:857]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
9:[16:17:20:375]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
10:[16:17:20:540]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
11:[16:17:21:103]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
12:[16:17:22:219]: I am instance of MyBase class, ID= base_b2 playing role MyRole in team MyTeam.
```

b

Figure 7.2 The output of executing the application branches deployed at H1 and H2.

The first two invocations (labeled by "1:" and "2:" as shown in Figure 7.2 (a) and (b)) at both hosts call the original `whoAmI` method of the remote base objects; because the remote team instance deployed on node H3 is not yet activated. The time stamp of each invocation has been appended to the output to trace application execution.

Afterward, the remote team instance on H3 is activated, and the `whoAmI` method invoked at both hosts for 10 successive times. In Figure 7.2 (a), the output lines labeled from "3:" to "12:" illustrate that all these invocations have *not* been *intercepted* by the declared remote CIMB. This result is correct because the remote BGP expression *evaluates* to "*false*" each time. That is, when the remote team instance evaluates the BGP expression, it sums up the values of the array `myVals` of the remote base object created at H1. The sum is "15" which is less than the value of the teams' field `THS` "34". Thus, the remote base object will *not play the role*. Conversely, in part (b) of the figure, the output lines labeled from "3:" to "12:" indicate that the `whoAmI` method has been *intercepted* successfully. This means that the remote base object, which deployed on H2, has been bounded with a role instance in the remote team; because the remote BGP evaluates to "true" (the sum of `myVals` base field is 44, which satisfies the condition).

Mode-II: In this mode, the remote team instance deployed on node H3 is activated, and then the method `whoAmI` is invoked intensively for 1000 times at each of hosts H1 and H2. The experiment has been repeated 5 times to ensure robustness. At all attempts, the method `whoAmI` of the remote base objects deployed on H1 and H2 has been intercepted with the remote CIMB of expression (7.1).

7.1.3 Performance Analysis

In Mode-I, the runtime values of invocations of method `whoAmI` has been recorded. The results are shown in Table 7.1. Part (a) of the table shows the runtime recorded at host H1, and in part (b) the runtime values of host H2.

Table 7.1. The runtime values of intercepting the method `whoAmI` on H1 and H2 (Mode-I).

	a – host **H1**	b – host **H2**
No.	**Runtime (in milliseconds)**	**Runtime**
1.	5	3
2.	2	1
3.	123	73
4.	15	13
5.	13	13
6.	13	10
7.	7	10
8.	15	10
9.	12	10
10.	15	8
11.	12	10
12.	13	7
Avg.	**20.4**	**14**

Note that in both cases the first two runtime values are smaller than the rest runtime values because remote base objects did not detect any activated remote team instances yet. Then, at row 3 in both cases, the value of the runtime suddenly jumps to a large value (in relative to the rest values). In the 3^{rd} invocation, the remote team instance is found activated. In part (a), the runtime value at row 3 is 123ms, and in (b) 73ms. This runtime primarily includes:

- The time for trapping the base method call by the MsD.
- The time for preparing the Active Remote Teams List (ARTL).
- The time needed by the Java-RMI system to create the necessary proxy objects from the GRBI of remote base objects and the GRTI of the remote team.
- The time required for serializing and de-serializing the parameters needed in the evaluation of the remote BGP expression.
- The time for lifting a new role instance and invoking its callin method.
- The time required to return the result of callin method back to the intercepted base method.

The sequent calls in both cases (from row 4 to row 12) are executed in average runtime of 12.7ms for node H1 and 10.1ms for H2, respectively. The runtime values are reduced because proxy objects are created for one time. Also, the role instances associated to remote base objects are created for one time.

The recorded runtime values for Mode-II are shown in Table 7.2. At each row, the average runtime value for invoking the method `whoAmI` 1000 times is listed. All experiments have been executed on the environment with the specifications in Table 7.3.

Table 7.2: The runtime of intercepting method `whoAmI` on H1 and H2 in Mode-II.

	a – Runtime at H1		b – Runtime at H2	
No.	**Runtime of the first invocation**	**Avg. Time of 1000 invocations**	**Runtime of the first invocation**	**Avg. Time of 1000 invocations**
1	89	2.488	49	1.593
2	56	1.956	43	1.408
3	59	2.029	55	1.358
4	64	2.494	42	1.382
5	96	2.287	68	1.634

Table 7.3: The environment of experiments.

Specification	**Value**
Operating System:	Windows 7 Professional 32-bit SP1
IDE:	Eclipse™ 3.5.2 Galileo – Java 1.6
OT Development Tooling:	1.3.3
Processor:	Intel® Core™ 2 Duo 2.1 GHz CPU with 3 MB cache and 4GB RAM

7.2 Dynamic Adaptation of Distributed Application Objects [3]

This section will demonstrate how the RRP could be employed to dynamically adapt a Java-based client-server messaging application (which might be a legacy application). In specific, it employs a simple *encryption-decryption* algorithm: clients send their messages encrypted according to a specific algorithm, and the server will decrypt these messages when arrived and just before it processes them. It will explain how DOT/J can help to implement this new requirement and enables client and server objects to adapt at runtime without modifying their source code.

7.2.1 Case Study: Dynamic Encryption/Decryption

Figure 7.3 (a) shows the class diagram of a simple messaging client-server application. Clients send plain-text messages after they connect to the server. The server manipulates these messages and sends its response. The client and server objects use Java *sockets* to establish the communication (the complete implementation could be found in Appendix-C.4). Part (b) of the figure illustrates application deployment on different hosts (nodes/processes).

Now, clients and server objects need to adapt and change their behaviors so they can perform the new requirement (i.e. encryption/decryption). The DOT/J framework can modularize this requirement as shown in Figure 7.4, where:

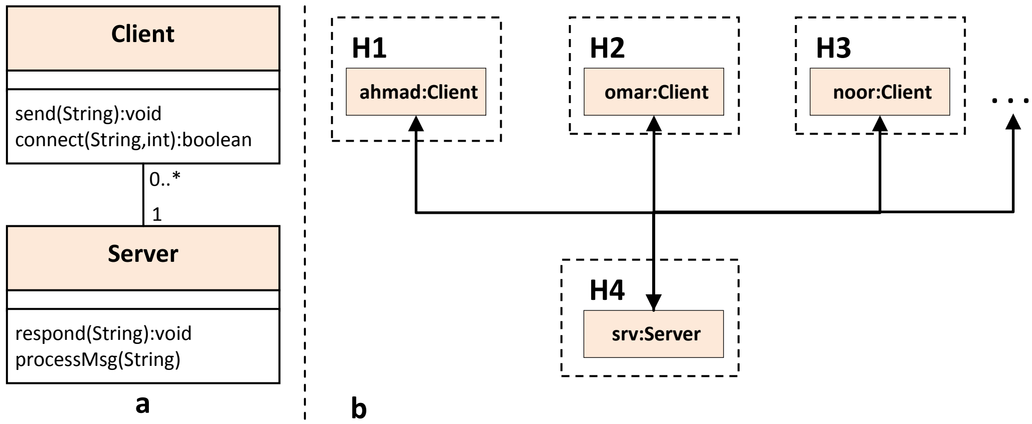

Figure 7.3. A class diagram of simple client-server messaging application (a), and a deployment scenario of that application in (b).

- Base objects of **Client** class play the **EncryptedClient** role in the **EncryptionTeam** team. The replacement CIMB (send ← **replace** send;) intercepts all calls to clients' "`send`" method.
- Base objects of **Server** class play the **DecryptedServer** role in the **DecryptionTeam** team. The replacement CIMB (processMsg ← **replace** processMsg;) intercepts all calls to "`processMsg`" method of a server object.
- The **Client** base objects *should not* start encrypting their messages (i.e. play the **EncryptedClient** role) unless a team instance of type **DecryptionTeam** team (hereafter called *decryptor-team*) has already been *deployed* and *activated*.
- To synchronize encryption and decryption algorithms, the decryptor-team must not be activated *before* the encryption process is sparked (i.e. before a team instance of type **EncryptionTeam** (hereafter called *encryptor-team*) is deployed and activated). In other words, the decryptor-team must be activated only when the encryptor-team has activated. This could be easily expressed in DOT/J as follows:

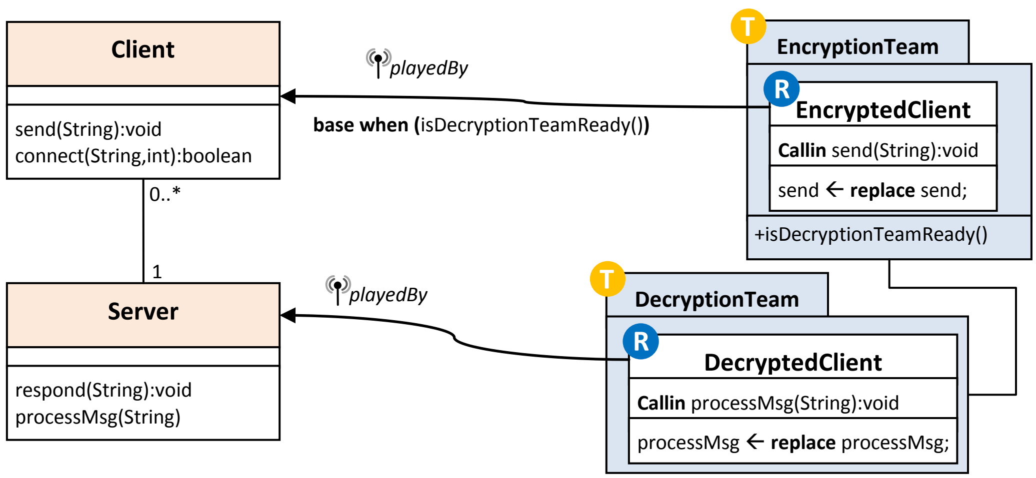

Figure 7.4. Adapting client-server messaging application through playing **EncryptedClient** and **DecryptedServer** roles.

- At the execution of encryption-side application, create and activate the enryptor-team. Then, make sure that the decryptor-team is deployed. If so, then activate it remotely. This could be done by inquiring the DOTM component (which holds information about all remote team instances), like this:

```
// at the EncryptionTeam .. see Appendix C.4
IRemoteTeam decryptor = DOTM.getRemoteTeam("a.b.DecryptionTeam");
if(decryptor != null)
{ // the decryptor-team is deployed .. activate it
    decryptor.remoteActivate();
    ..
}
```

- The Base Guard Predicate (BGP) [**`base when`** `(isDecryptionTeamReady())`], shown in Figure 7.4 along the *playedBy* relationship, calls the method `isDecryptionTeamReady` to check if the decryptor-team is still deployed and activated. Recall that the BGP expression will be evaluated each time callin methods of roles is being invoked; thus, the encryption/decryption consistency is ensured.

Both steps guarantee that no messages will be encrypted and decrypted disorderly [3].

7.2.2 A Simple Encryption Algorithm

The encryption algorithm used in the **EncryptedClient** role is simple: "the message's halves are exchanged, and a special stamp is appended. At the **DecryptedServer** role, a reverse process is performed before the server processes that message according to its original method."

7.2.3 Execution and Performance Analysis

To evaluate the performance of DOT/J, a set of experiments has been performed. Two laptops connected in a LAN are used as network nodes. The specifications of these laptops are shown in Table 7.4. To measure the overhead of mapping the client-server application using DOT/J framework (without RRP), the server-side application has been executed on "Sony" host and a client-side application on "Toshiba". The experiment has been repeated twice; one without running the DOT/J framework and the other with the DOT/J framework being operated under objects. In both cases, the client object is sending 1000 messages with a delay of 100ms between each two successive messages. The charts of runtimes in both cases are illustrated in Figure 7.5.

Table 7.4: The environment of experiments of Dynamic Adaptation.

Specification	Sony	Toshiba
Operating System	Win7 Pro 32-bit SP1	Win7 Pro 64-bit SP1
CPU	Intel® Core™2 Duo 2.1GHz	Pentium dual-Core 2.0GHz
RAM	4 GB	
Java	1.6	
Eclipse SDK	Galileo 3.5.2 Build id: M20100211-1343	
OT Development Tooling	OTDT Version: 1.3.3	

The average runtime of sending the 1000 messages and receiving their acknowledgement from the server in case of executing the application without the DOT/J framework is 1.134ms. The average runtime of sending the same 1000 messages in case of activating the DOT/J framework (i.e. application is running on top of DOT/J) is 1.882ms, which demonstrates that DOT/J adds small overhead. The added runtime consists of the time of dispatching "send" calls to the Methods Dispatcher (MsD) and check for any available remote CIMBs. In case of executing application over the DOT/J framework (part (b) of Figure 7.5), the first message to send takes 328ms (not shown in the figure for clarity). This time is consumed by the MsD to fetch the Remote Teams List (RTL) of **Client** base class from the DOTM for the first time.

Then, the dynamic adaptation of client-server application has been evaluated by running on "Sony" host the server-side and the decryptor-team applications. After that, one client-side and the encryptor-team applications have been executed on "Toshiba" host. This experiment has been repeated three times with different delay time values each time between the sent messages (respectively 100, 10, and 1ms). This measures the RRP performance in different intensive scenarios. Parts of the results of sending 1000 messages are shown in Table 7.5, and in Figure 7.6 the charts of these results are shown (charts did not show the first runtime value for clarity).

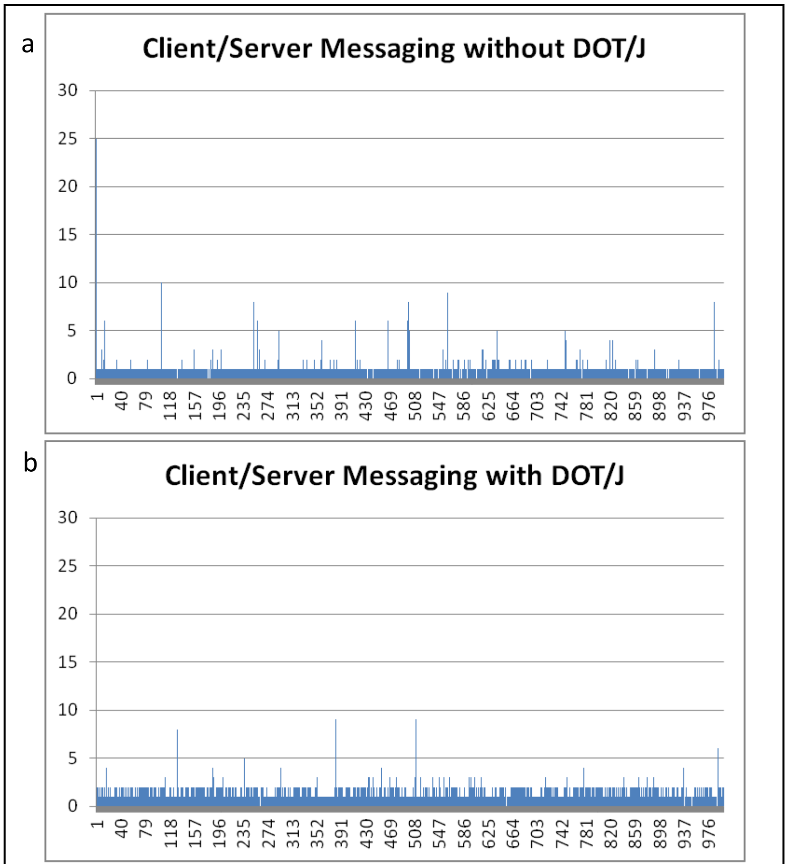

Figure 7.5. Executing the client-server messaging application to measure the overhead of using DOT/J framework: in (a) without DOT/J, and in (b) with using DOT/J.

The runtime (RT) here is the time needed to perform a complete invocation to the method "send" on a client object, which is calculated from the following formula [3]:

$$RT_{comp} = MsD_{Time} + ARTL_{Time} + TLWM_{Time} + Enc_{Time} + Send_{original} + Dec_{Time} + Ack_{Time}$$

Where:

MsD_{Time}: time needed by the Methods Dispatcher (MsD) to trap the invocation.

$ARTL_{Time}$: the time required to prepare the currently activated remote teams list. For the first call to "send" method, this time includes the runtime required to communicate with the DOTM to fetch the RTL list for **Client** base class.

$TLWM_{Time}$: the time for invoking remotely the team's wrapper method.
Enc_{Time}: the time for encrypting the message in hand.
$Send_{original}$: the time of actual message transmission to the server.
Dec_{Time}: the time needed by the decryptor-team to intercept the server's "processMsg" method and decrypts the received message.
Ack_{Time}: the time to send message acknowledgment to the client.

Table 7.5: Runtime values of Dynamic Adapting the Client-Server Messaging Application

Nr.	**Delay 100ms**	**Delay (10ms)**	**Delay 1ms**
1	305	307	278
2	34	11	9
3	14	9	20
4	15	13	10
5	17	13	10
6	17	16	10
7	11	12	9
8	13	12	10
9	10	12	10
10	16	13	13
:	:	:	:
1000	10	9	6
Avg.	**12.6**	**10.9**	**7.04**

Note from Table 7.5 and Figure 7.6 that runtime values decrease even though the rate of message sending has increased. In case of high rate sending (i.e. with delay of 1ms), the Java-RMI system reuses the objects generated implicitly during remote method invocations like `RemoteObjectInvocationHandler` before they have been garbage collected.

Experiments of this case study could be compared to the experiments established by Horio *et. al.* [51]. In their proposed language, called DandyJ, dynamic aspect weaving mechanism is used. However, application developers need to explicitly deploy the *dynamic aspect* instances of the generic dynamic aspect of DandyJ (called DAspect). In addition, more than 5 aspects are used to implement the encryption/decryption adaptation. Regarding performance, the last established experiment (i.e. with time delay of 1ms) is equivalent to the experiment performed in [51]. The average runtime measured in their experiment is 49.6 ms (with network latency of 27ms between client and server nodes which locate 700km away from each other). Though, the RRP technique registers better performance (avg. of 7.04ms) over a LAN of two hosts.

With respect to expressivity, DandyJ reuses the AspectJ's *aspect* and *joinpoint* (JP) model. Thus, application developers need to be familiar with AspectJ and its JPM. In DOT/J framework, on the other hand, the expressive *playedBy* relationship and the Role-based design of the OT model have been used. All OT/J applications are readable and understandable without being familiar with the OT model thanks to *playedBy* relationship.

In this case study, application developers can access the DOTM via its APIs in their programs to inquiry the status of remote team instances from any involved node. This is considered as *flexibility* of DOT/J; since reasoning about the deployment status of dynamic aspects in DandyJ is achieved by "asking" aspects themselves. Finally, DandyJ – like AWED – uses explicit deployment designators in pointcuts declarations; namely the `host` and `hosts` pointcuts to respectively select joinpoints on remote host or a group of hosts. This reduces the model transparency. In DOT/J, allocation of roles is performed totally in transparent manner, which allows employing new roles dynamically.

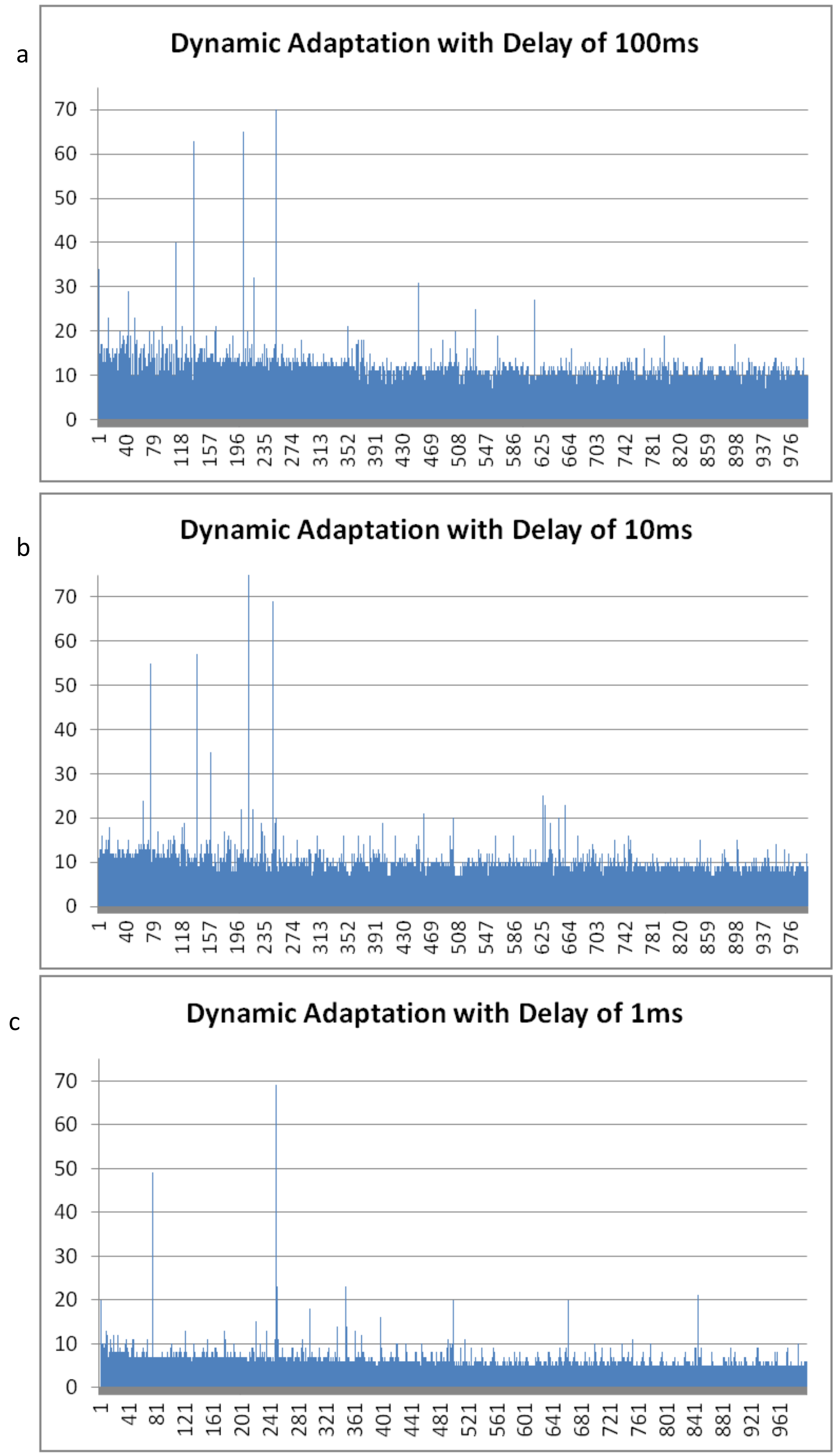

Figure 7.6. Applying Dynamic Adaptation of Client-Server Messaging Application in different delay times: in (a) 100ms, in (b) 10ms, and in (c) 1ms.

7.3 Supporting Distributed Aspects [2]

This section develops a distributed application that defines a simple distributed aspect. The application will be developed with AWED and DOT/J to establish a comparative discussion between both of them as distributed AOP approaches. The experiment in this section aims at emphasizing the capability of DOT/J to leverage the modularity of distributed aspects and supporting them at runtime by providing controllable contextual modules (i.e. remote teams). Then, the performance of each approach will be discussed.

7.3.1 Case Study: A Simple Chatting Application using Distributed Aspects

Consider a simple chatting application that enables clients to send text messages to each others. The basic chatting application is depicted in the class **Messenger** shown in Figure 7.7; the implementation of this class could be found in Appendix-C.2.The conception for *broadcasting* messages to all clients of the distributed application is to intercept the method `processMessage` of class **Messenger**, and send a copy of that text-message to every client except the sender.

In AWED, this application might be developed as follows (see Figure 7.7):

1. Define an aspect module with a *hook* construct. The hook construct should declare a pointcut and an advice to intercept the target method. The implementation of aspect class **MessengerAsp** that was developed by AWED's founders in their tutorial [85] will be used. The implementation of this aspect and its hook, however, could be found in Appendix-C.2.1.
2. Define a *connector* module to connect the target method of class **Messenger** with the pointcut defined in the hook of **MessengerAsp**. Again, the connector module that has been implemented originally in AWED tutorial is used here. The implementation could be found in Appendix-C.2.1.

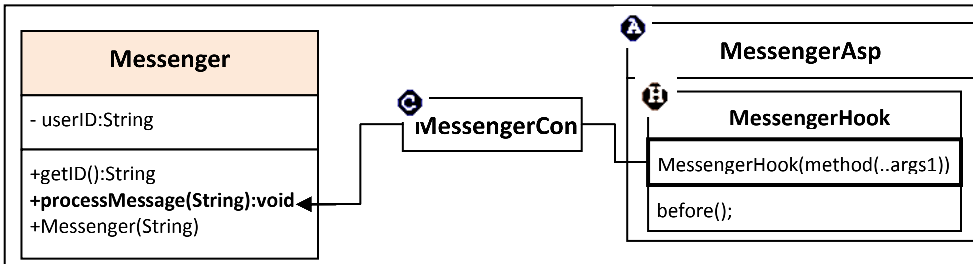

Figure 7.7. A class diagram for a simple chatting application in AWED.

Developers can develop an "identical" application using DOT/J as follows (see Figure 7.8):

1. Define the remote team class **ChattingTeam**, which declares the role class **Chatter** that will be played by class **Messenger**.
2. The role class **Chatter** (the aspect) defines the callin method `beforeProcessMessage` (the advice) which will be invoked *before* the method `processMessage` of the base class. The interception is evaluated according to the following remote CIMB (the pointcut):

```
beforeProcessMessage ← before processMessage;
```
(7.3)

The implementation of team class **ChattingTeam** and its role class **Chatter** could be found in Appendix-C.2.2.

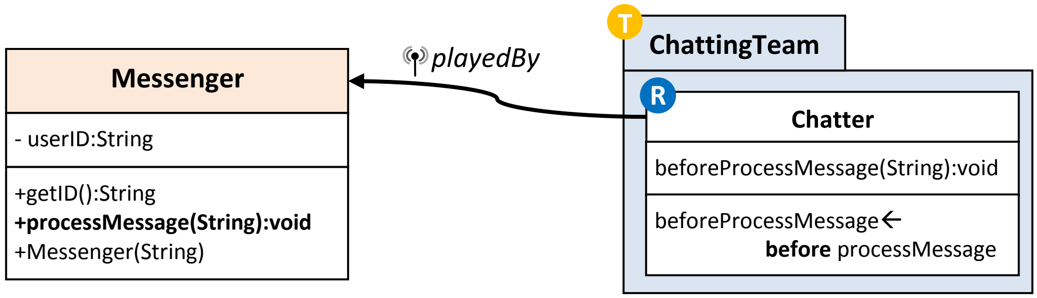

Figure 7.8. A class diagram for simple chatting application in DOT/J.

7.3.2 Deployment of Application

The chatting application will be executed on three nodes to simulate a chatting session between three clients. To employ the aspect **MesengerAsp** of AWED, a copy of that aspect must be deployed on every node as shown in Figure 7.9 (a). Conversely, to employ the remote role-playing of DOT/J, a single remote team instance of type **ChattingTeam** class need to be deployed on a new (or existing) node as shown in Figure 7.9 (b).

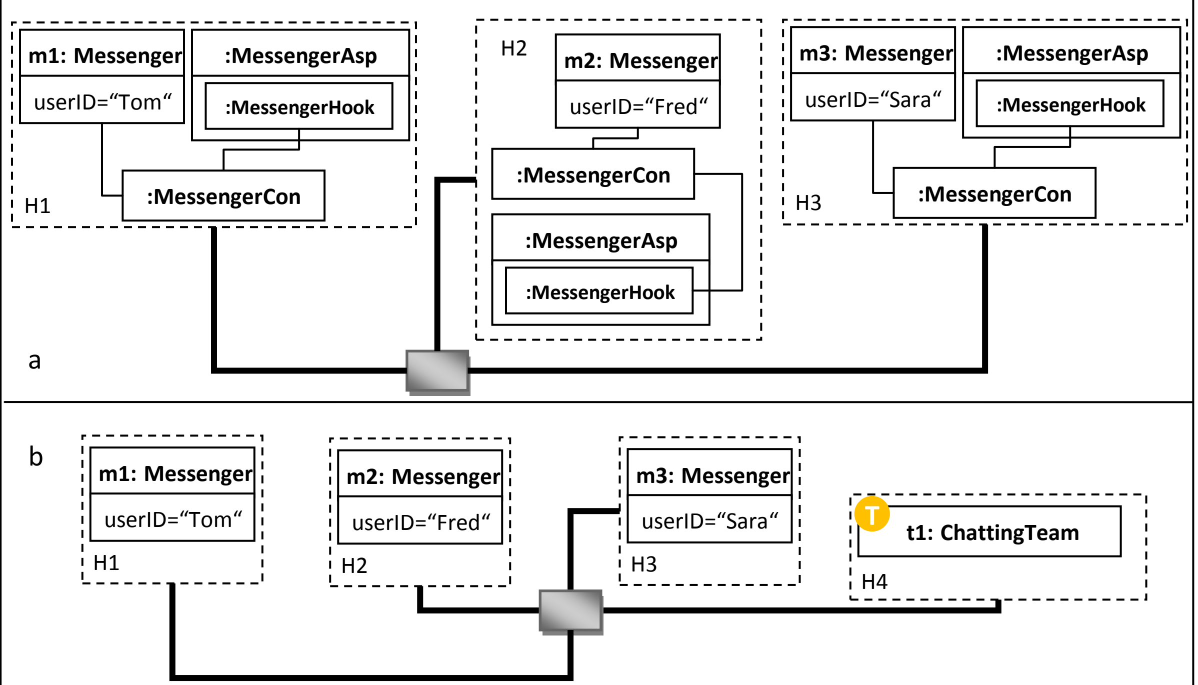

Figure 7.9. A deployment scenario for the chatting application: (a) in AWED, and (b) in DOT/J.

7.3.3 Execution Scenario

At each of H1, H2 and H3 nodes the user is requested to type a specific user-ID when the application starts. Then, he/she can immediately send messages by typing a text message and then press the "Enter" key. In case of the DOT/J version of application, a remote team instance at H4 has been created and immediately activated before the applications at H1, H2 and H3 have started.

In case of using AWED, the user “Tom” at node H1 sends 10 successive messages. Thus, each time “Tom” presses the “Enter” key, the method `processMessage` is intercepted and the hook’s advice `before` in executed at all other nodes (i.e. H2 and H3). As a result, each message sent by “Tom” will be replicated and printed at every node. The same procedure is performed by “Fred” and “Sara”; each one of them sends 10 successive messages.

In case of using DOT/J (Figure 7.9 (b)), the same procedure is performed. Each client sends 10 successive messages. When sending a message, the remote CIMB intercepts the call of method `processMessage`, and executes the roles’ method `beforeProcessMessage` *before* the invocation of method `processMessage` is resumed. To ensure a precise comparison, the same set of text-messages that were used in AWED’s case are used here.

Note that the role’s advice `beforeProcessMessage` will be executed at the team’s side and not on the base objects sides. Therefore, *all* the intercepted messages will be printed at the role’s consol. However, to simulate the behavior of AWED application, developers can implement the role’s callin (advice) `beforeProcessMessage` to *broadcast* every intercepted base message to all clients on hosts other than the host where the message has been sent originally. For example, when “Tom” sends a message, then that message will be disseminated to the hosts of “Sara” and “Fred”. To achieve this, the role **Chatter** defines the following remote COMB:

```
void broadCast(String msg) → void print(String msg);
```
(7.4)

Inside the role’s method `beforeProcessMessage`, an invocation to the method `broadCast` will be issued at each role instance other that the current role instance (which executes the advice).

The DOT/J will be compared against AWED in two modes: (1) without remote COMBs, and (2) with remote COMBs. The first mode compares mainly the performance of DOT/J against AWED with respect to the *Remote Joinpoint Interception.* In other words, both models are compared in case of a complete interception cycle, which includes the total runtime for intercepting the base object method, evaluate the interception at the aspect (role), execute the associated advice, and return the control flow back to the intercepted base object. The second mode, on the other hand, simulates the behavior of the application as developed in AWED, i.e. DOT/J simulates remote advice execution on all nodes by calling out the base method `print` as declared in the expression (7.4).

7.3.4 Discussion and Performance Analysis

The execution environment with the specifications listed in Table 7.3 has been used to perform all experiments. In case of AWED, the 10-messages of “Tom” appear on nodes H2 and H3 as expected. Likewise, messages of “Fred” and “Sara” have been successfully replicated. The runtime values required for a complete interception to each message for every user have been recorded. Table 7.6 illustrates these values (in milliseconds).

Note that the first interception at each node consumes more execution time than the rest because AWED Runtime System performs some preparations for replicating the joinpoint object at every node. This includes in addition the communication between the connector repositories deployed on each node, and the synchronization of states of aspects instances.

Table 7.6. Runtime to intercept the method `processMessage` for 10 times per user (AWED).

No.	**H1 - Tom**	**H2 - Fred**	**H3 - Sara**
1	196	54	66
2	8	5	6
3	7	8	6
4	3	6	8
5	6	6	5
6	5	5	7
7	7	6	6
8	6	5	8
9	17	7	6
10	5	6	5
Avg.	**26**	**10.8**	**12.3**

The recorded results in case of DOT/J implementation are shown in Table 7.7 below. Once again, the first round of remote CIMB interception costs more than the others. However, it is clear that DOT/J performs better than AWED despite that small difference between the runtime values at each of them. The right part of the table shows the runtime of the remote CIMB interceptions including the invocation time of the remote COMB declared in expression (7.5). Note that at each row in table's branches, the runtime is increased in case of remote COMB invocation; which is rational. The exception occurs in the runtime values of the first row of the right branch. This is because generating the proxy objects for the remote base objects is handled before the method `processMethod` is intercepted; because in this mode (i.e. with remote COMB) **Messenger**'s objects are forced to register in the **ChatterTeam** team before they start sending messages. According to the average values shown in Tables 7.6 and 7.7, DOT/J is expected to perform better than AWED in an intensive interception mode.

Table 7.7: Runtime to intercept the method `processMessage` for 10 times per user (DOT/J).

	Without **COMBs** – without registration			With **COMB** – with registration		
No.	**H1 - Tom**	**H2 - Fred**	**H3 - Sara**	**H1 - Tom**	**H2 - Fred**	**H3 - Sara**
1	100	78	53	25	22	20
2	5	2	8	4	9	8
3	5	5	5	12	12	9
4	5	2	5	9	9	9
5	5	7	5	10	9	10
6	2	15	5	10	11	18
7	5	3	5	15	14	8
8	2	10	5	11	19	7
9	15	5	12	16	13	12
10	5	5	5	8	9	15
Avg.	**14.9**	**13.2**	**10.8**	**12**	**12.7**	**11.6**

To assert this expectation, executing the application in both models has been repeated in intensive mode as follows: "At each node, the user sends a standard text message for 1000 times without any delay." The results of this experiment are shown in Table 7.8. As expected, DOT/J consumes less time than AWED in both interception modes; without and with COMBs. This actually refers to: (1) The DOT/J Runtime System caches the remote stubs of the remote base objects, and (2) DOT/J establishes remote communication between the remote base objects of **Messenger** and the remote team instance of **ChattingTeam** at only once, i.e. there is no need to contact the DOTM each the base method `processMessage` is intercepted as AWED does. In AWED, the intercepted object contacts the *connector repository* each time a joinpoint to the method `processMessage` is matched.

Table 7.8. The average runtime values (in milliseconds) for intensive interception to the method `processMessage` for 1000 times per user on H1, H2 and H3.

	AWED	**DOT/J** (without COMB)	**Reduction%**	**DOT/J** (with COMB)	**Reduction%**
H1 - Tom	3.583	1.012	71.8%	2.231	37.7%
H2 - Fred	2.907	0.793	72.7%	1.449	50.2%
H3 - Sara	2.899	0.84	71.0%	1.22	57.9%
Avg.	**3.13**	**0.882**	**71.8%**	**1.633**	**47.8%**

The reduction percentage (shown in the table) points out to the percentage in which DOT/J preserve execution time at each application nodes relevant to AWED's time.

7.3.5 Programming Capabilities of DOT/J and AWED

In all AWED execution scenarios for this application, "Sara", "Tom" and "Fred" cannot know who sends the messages each one of them has received from the others. In practice, this is because AWED does not provide any contextual information about the object at which the remote joinpoint has been matched. Thus, in the first experiment, for example, "Fred" and "Sara" will never know that the first 10 text-messages are sent by "Tom" (unless "Tom" appends his name with each message!). This shortcoming could be generalized as follows: "In AWED, the remote hosts on which a specific advice has been executed (due to a joinpoint matching on other specific remote host) do not obtain any contextual information about the remote object triggering the joinpoint."

In DOT/J, however, this shortcoming could be resolved since role instances are able to issue remote COMBs *during the execution of remote advices*. Furthermore, due to the role confinement rule, role instances are able to exchange the state of their remote players inside the contextual boundaries of team instance. For example, the role instances of **Chatter** can *callout* the base method `getID()` via the following remote method-COMB to fetch the user–ID who sends the current intercepted message:

```
String getUserID() → String getID();                    (7.5)
```

7.4 Multiple Remote Role-Playing and Dynamic Team Attachment

So far, the evaluation of single remote role-playing has been discussed through implementing simple distributed applications. This section will develop a case study for multiple remote role-playing situations. The study also will verify that the precedence of role-playing and the remote teams' activation priorities are preserved.

7.4.1 Case Study: A Multiple Remote Role-Playing Environment

Consider a DOT/J application that consists of two remote teams and two remote base classes as the one shown in Figure 7.10. The diagram illustrates the relationships between the remote base classes and the roles of remote teams. Without repeating the description of *playedBy* relationships, note that the base class **BaseY** plays the remote role **ARole2** in the remote team **RemoteTeamA** and simultaneously the local role **BRole2** in the remote team **RemoteTeamB**. The goal of mixing local and remote role playing is to verify that the DOT/J framework could be integrated precisely with the OT/J infrastructure. That is, a base class can play both local and remote roles without negative interfering or violating to its original functionality.

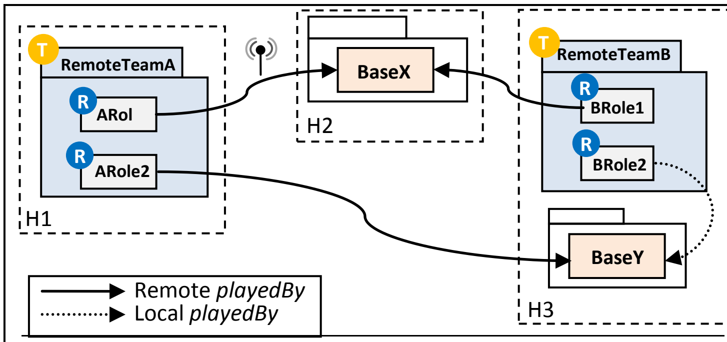

Figure 7.10. An arbitrary distributed application depicted Multiple Remote Role-Playing.

In this case study, the following questions will be answered:

1. Can the base classes **BaseX** and **BaseY** play their roles in different team activation orders?
2. Can the remote base class **BaseY** play the local role **BRole2** and the remote role **ARole2** simultaneously?
3. Does the playing of local roles have higher precedence than playing remote roles?
4. Does the activation status of remote teams of class **RemoteTeamB** *affect* the remote and the local *playedBy* relationships?
5. Can the remote base objects of **BaseX** and **BaseY** exhibit their original functionalities in case of *remote teams' unavailability*?

7.4.2 Implementation

For the purpose of simplicity, the base classes **BaseX** and **BaseY** have been implemented so that each class has declared one method. The `baseMethod` method returns a string value as shown in line 5 of the first code snippet (A). Then, each role has declared a replacement CIMB (as shown at line 4 in the description of team class **RemoteTeamA** in code snippet (B)). The CIMB expression binds the base method with a callin method named `baseMethod`[1]. The complete implementation of could be found in Appendix-C.3.

[1] The callin method of role must be the same name of the bounded base method if that callin issues a "base call" to that base method [26].

A

```
public class BaseX {
  private String baseID = "BaseX";
  private int[] intArray =
       new int[] {10,20,30,40,50};
  public String baseMethod()
   { return this.baseID;}
}
```

B

```
public team class RemoteTeamA{
protected class ARole1 playedBy BaseX
   base when(base.intArray[2]>9){
   baseMethod <- replace baseMethod;
     ..
 }
protected class ARole2 playedBy BaseY
 {..}
}
```

```
public team class RemoteTeamB{
 protected class BRole1 playedBy BaseX {..}
 protected class BRole2 playedBy BaseY { // Local Role Playing
   private String roleID = "BRole2_LOCAL";
     :
 } }
```

C

7.4.3 Application Deployment and Execution

A remote team instance of type **RemoteTeamA** has been deployed on the host H1 (see Figure 7.11), and then that instance has been activated globally. At the application node H2, an instance of remote base class **BaseX** is created, and immediately it starts calling the method `baseMethod` 100 times with a delay of 2 seconds between each two successive calls (to allow monitoring the moment of role-playing). Finally, at H3 an instance from the remote team class **RemoteTeamB** and a base object from the base class **BaseY** have been created. The remote team instance of H3 is immediately activated upon creation, and the base object of **BaseY** starts calling the method `baseMethod` 100 times with a delay time of 2 seconds.

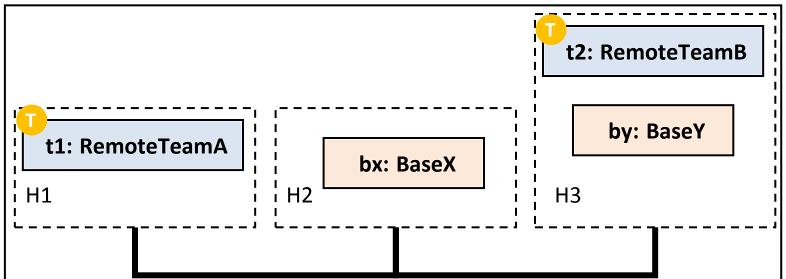

Figure 7.11. A Deployment plan for the application of Figure 6.6.

To answer the aforementioned questions, the application needs to be executed in different remote team *activation order* scenarios. In practice, at the labeling phase, the remote team **RemoteTeamA** has been given an activation priority value greater than **RemoteTeamB** team as shown in the XML snippet below. In this way, it is possible to evaluate whether the base instance **by** of base class **BaseY** is able to play the local role first or not.

```
<RemoteTeam class="de.evaluation.c3.teams.RemoteTeamA" activationPriority="33">
  :
</RemoteTeam>
<RemoteTeam class="de.evaluation.c3.teams.RemoteTeamB" activationPriority="11">
  :
</RemoteTeam>
```

For this purpose, two execution scenarios have been performed as follows:

Scenario-1: **H1** (RemoteTeamA) → **H2** (BaseX) → **H3** (RemoteTeamB)

This scenario executes first the application at H1, and then the application at H2 (on which the remote base object **bx** resides). In this case, **bx** starts playing the **ARole1** in team **t1**. Figure 7.12 (a) shows an output snippet for the application running on H2. Note that the base object method `baseMethod` has been *replaced* by the role callin method.

Afterward, the application at H3 is triggered. The created remote team instance **t2** comprises two roles: a remote role for base object **bx** and a local role for the new generated base object **by** of base class type **BaseY**. At this point of execution, and because **t1** has a higher priority than **t2**, the remote base object **bx** (at H2) *continues* playing role **ARole1**. Concerning **by**, it plays the *local* role **BRole2** as expected; because the base method `baseMethod` has been *replaced* by the local role's callin method as shown in the first three lines of part (b) of Figure 7.12. This actually answers question number 3.

(a)
```
1:ARole1:BaseX
2:ARole1:BaseX
3:ARole1:BaseX
:
74:ARole1:BaseX
75:ARole1:BaseX
76:BaseX
77:BaseX
:
92:BaseX
93:BaseX
94:BRole1:BaseX
95:BRole1:BaseX
96:BRole1:BaseX
97:BRole1:BaseX
98:BRole1:BaseX
99:BRole1:BaseX
100:BRole1:BaseX
```

(b)
```
1:BRole2_LOCAL:BaseY
2:BRole2_LOCAL:BaseY
3:BRole2_LOCAL:BaseY
:
48:BRole2_LOCAL:BaseY
49:BRole2_LOCAL:BaseY
50:ARole2:BaseY
51:ARole2:BaseY
:
70:ARole2:BaseY
71:ARole2:BaseY
72:BaseY
73:BaseY
:
88:BaseY
89:BaseY
90:BRole2_LOCAL:BaseY
91:BRole2_LOCAL:BaseY
```

(c)
```
1:BRole1:BaseX
2:BRole1:BaseX
3:BRole1:BaseX
:
28:BRole1:BaseX
29:BRole1:BaseX
30:ARole1:BaseX
31:ARole1:BaseX
:
62:ARole1:BaseX
63:ARole1:BaseX
64:BaseX
65:BaseX
:
87:BaseX
88:BaseX
89:BRole1:BaseX
90:BRole1:BaseX
```

(d)
```
1:BRole2_LOCAL:BaseY
2:BRole2_LOCAL:BaseY
3:BRole2_LOCAL:BaseY
:
48:BRole2_LOCAL:BaseY
49:BRole2_LOCAL:BaseY
50:ARole2:BaseY
51:ARole2:BaseY
:
63:ARole2:BaseY
64:ARole2:BaseY
65:BaseY
66:BaseY
:
88:BaseY
89:BaseY
90:BRole2_LOCAL:BaseY
91:BRole2_LOCAL:BaseY
```

(e)
```
1:ARole1:BaseX
2:ARole1:BaseX
3:ARole1:BaseX
4:ARole1:BaseX
5:ARole1:BaseX
6:ARole1:BaseX
7:ARole1:BaseX
8:ARole1:BaseX
9:ARole1:BaseX
10:BaseX
11:BaseX
12:BaseX
13:BaseX
14:BaseX
:
```

Figure 7.12. The output of executing the application of Figure 6.6 according to Scenario-1 at (a) and (b), and according to Scenario-2 at (c) and (d). Part (e) lists the results of dynamic remote BGP evaluation.

To make sure that the base object **by** can play the remote role **ARole2** properly, the application at H3 has been programmed to deactivate **t2** automatically after the 50^{th} call to **by**'s method. As Figure 7.12 (b) shows at line 50, **by** starts playing the role **ARole2** because calls of its method have been intercepted by the remote CIMB declared in the role **ARole2**. This actually answers questions 2 and 4 together.

When the `baseMethod` method of **bx** has been invoked at the 75th call, application execution at H1 is stopped to monitor the remote base objects behaviors in case of remote teams' *unavailability*. As expected, both **bx** and **by** base objects have reclaimed the original behavior as illustrated at line number 76 of Figure 7.12 (a) and line number 72 of part (b)[1], which answers question number 5. To complete the scenario, the remote team instance **t2** is reactivated after round number 90 automatically in order to verify that **bx** and **by** can resume playing the roles **BRole1** and **BRole2**, respectively. As illustrated in Figure 7.12 (a) line 94 and (b) line 90, both base objects resume playing their roles after **t2** has been reactivated, which answers question number 4.

Scenario-2: **H3** (RemoteTeamB) → **H2** (BaseX) → **H1** (RemoteTeamA)

This scenario starts executing the application of H3, and then H2. This means that the base object **by** starts playing the local role **BRole2**, and the remote base object **bx** (at H2) starts playing the remote role **BRole1** at host H3. Figure 7.12 (c) and (d) illustrates the output of applications at hosts H2 and H3, respectively. Note at the first few lines that the base objects' functionality has been intercepted precisely. After a while, the application of H1 is executed. In this case, **bx** is expected to start playing the role **ARole1** in **t1** team; because **t1** has activation priority higher than **t2**. The results came as expected as shown in Figure 7.12 (c) at line 30. At the same time, the base object **by** *continues* playing the *local* role **BRole2**. This asserts the answers mentioned in *scenario-1*to the questions in Section 7.4.1.

However, the application at H3 has been programmed to deactivate the remote team instance **t2** automatically before the method `baseMethod` of base object **by** is called for the 50th time. Consequently, **by** will immediately start playing the remote role **ARole2** (see Figure 7.12 (d) line 50). At the same time, **bx** continues playing the remote role **ARole1**. After a while, the remote team instance **t1** at H1 is deactivated. Thus, both base objects **bx** and **by** should reclaim their original behaviors as would be expected (see Figure 7.12 (c) line 64 and (d) line 65, which is the moment of **t1**'s deactivation detection at hosts H2 and H3).

To make sure that the remote base objects **bx** and **by** can respond accurately to the teams' re-activation, the remote team instance **t2** at host H3 has been re-activated. The effect of this reactivation is shown in Figure 7.12 (c) and (d), respectively at lines 89 and 90, where both base objects resume playing the roles of team **t2**; **bx** plays the remote role **BRole2** and **by** plays the local role **BRole2**.

In both scenarios, the remote base object **bx** was able to play the remote role **ARole1** because the remote Base Guard Predicate (BGP) expression [`base when(base.intArray[2]>9)`] (see line 3 in the description of team **RemoteTeamA** in code snippet (B)) is evaluate to "`true`" each time **bx**'s method has been intercepted. However, one more execution scenario should be performed to verify the dynamic evaluation of remote BGPs. For this purpose, the application at host H1 has been executed first, and then the application of H2. As expected, the method `baseMethod` of remote base object **bx** has been intercepted by the remote CIMB defined in the role **ARole1** (see Figure 7.12 (e)). The application at host H2 was programmed to change the value of the 3rd element of the array `intArray` (i.e. the element at index 2) of base object **bx** to be "4" just before the 10th

[1] Line numbers are not identical because the application has not been executed simultaneously at H1, H2 and H3 hosts.

call to method `baseMethod`. As expected, the remote base object **bx** *stops* playing the role **ARole1** because the remote BGP evaluates to "`false`" each time after this change (see line 10 at part(e) of Figure 7.12).

7.5 A White-box Approach for Supporting Transparent Collaboration in Legacy Applications

A Distributed Collaborative Application (DCA) could be defined as a group of separated programs which execute on several network nodes to achieve a shared goal. The demand for easy to design and implement DCAs is highly increased as the complexity of the development of these applications has increased too. The main complexity in the development of DCAs is the decomposition of system functionalities in separate components [53]. For this purpose, several approaches have been developed like Tako [67] and AO-CVE [54]. These approaches employ the AOP concepts to improve the modularity of the DCAs by separating the collaborative functionalities of components from the application core functionality.

The OT model simplifies the separation of *Collaborative Functionalities* of application classes (which could be legacy applications) and capturing them in roles within teams. In fact, the OT model not only helps improving the modularity of collaborative applications, but also improves their understandability through the expressive *playedBy* relationship. The DOT/J framework makes use of these and other features of OT model to improve the modularity of the DCAs. This section will describe a simple *Distributed Collaborative Painting* application, where several "painters" collaborate remotely to draw a shared painting. It will demonstrate how the RRP could be employed as a white-box approach for achieving this goal.

7.5.1 Case Study: Distributed Collaborative Painting Application

Consider the painting application depicted in Figure 7.13 as a Model-View-Controller (MVC) design. The painter (the Controller) can create, prepare, and draw a set of shapes (the Model) on a graphical interface (View). The following is part of the implementation of class **Painter** in Java:

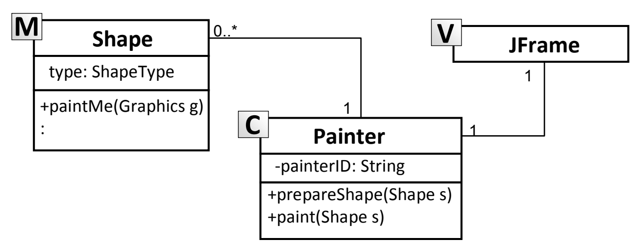

Figure 7.13. Class diagram of painting application.

```
class Painter
 implements MouseListener, ..
{
    JFrame window = ..;
    List<Shape> shapesList = ..;
    :
    public Shape createShape(ShapeType s, Point start, Point end...){..}
    public void paint(Shape s) {..}
}
```

The user of application deals with a GUI that enables her to draw ovals, rectangles, circles, and squares on a specific painting area. First, the user logs on to the application with a specific “painterID” of her choice. Then, the user can choose the type of the intended shape to draw and the border and fill color to use. By using the mouse, the user then draws that shape. The method **createShape** in **Painter** class is used to create a shape object for every drawn shape on the GUI by the user. The actual painting functionality is taking place via the method **paint** which paints the supplied shape object on the “view” component.

7.5.2 Building a Distributed Collaborative Painting Application Transparently

Now, to construct a collaborative painting application from this *legacy* single-user painting application *transparently*, the *Collaborative Functionalities (CFs)* which application classes can have in the collaboration should be determined first. In this regard, a collaborative painting process is needed. Thus, the main CF to be considered is the shapes painting process; which is represented by the **Painter** class. The main demanded collaborating activity is to enable a user to paint a specific shape on her and on other participants’ drawing areas “at the same time”.

Actually, this conception of painting collaboration could be easily modularized in OT model as shown in Figure 7.14, where **Painter** class plays the **CoPainter** role inside the **CollPainting** team. To realize the collaborating activity just mentioned above, the role **CoPainter** simply intercepts the painting CF of a **Painter** base object by the CIMB shown in the figure, and then *broadcasts* the drawn shape to every other **Painter** base objects. One way to perform this is by establishing a COMB that inducts these objects to draw that shape on their own painting areas via the method **paint**. Unfortunately, in OT/J this will cause an *infinite interception cycle*; since the COMB calls **paint** method which is, in turn, intercepted by the CIMB once again. Later, this section will discuss how it possible to overcome this problem efficiently in DOT/J framework without violating application integrity. Anyway, the following code snippet shows part of CollPainting team implementation:

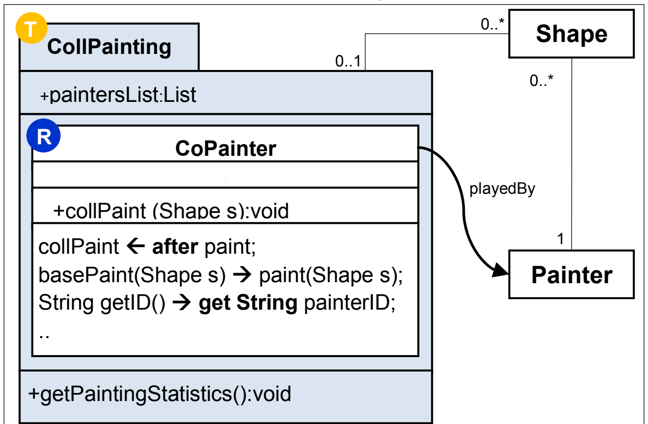

Figure 7.14. Modularizing the CBD of “painting collaboration” in OT.

```
public team class CollPainting{
     List paintersList =…;
     protected class CoPainter playedBy Painter {
             ..
      public void collPaint(Shape s)
      {
        System.out.println(“Painter:”+getID());

               /* paint the shape s at all
         other participants painters */
              for(CoPainter coP : paintersList)
               if(coP != this)
                 coP.basePaint(s);
      }

      collPaint <- after paint;                // CIMB
```

```
        basePaint(Shape s) -> paint(Shape s);    // COMB
        String getID() -> get String painterID; // COMB
       }
       public void getPaintingStatistics() {…}
   }
```

7.5.3 Deployment Plan and Execution Scenario

To construct a distributed painting collaborating, let us deploy three painter-applications on three different nodes and one team-application on a separate node as shown in Figure 7.15. Now, “Sara”, “Fred”, and “Tom” are three painters having three roles inside the remote team instance **coll**. To verify that the collaborating painting is work as desired, each painter has been allowed to paint 10 random shapes on his own GUI as partially shown in Figure 7.16 (a).

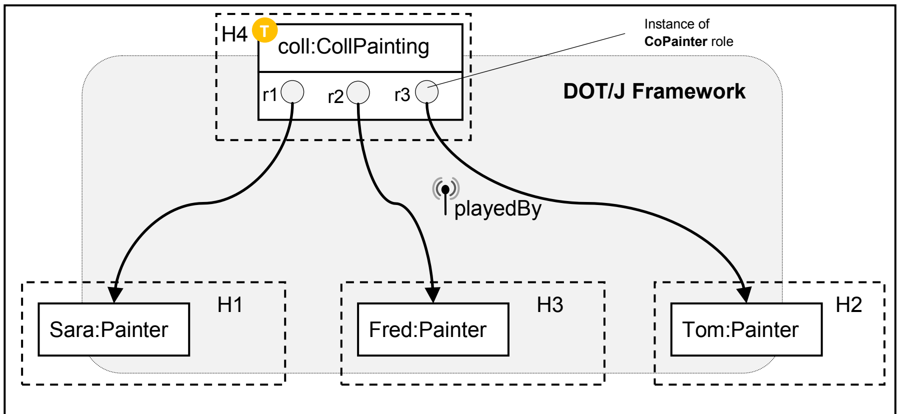

Figure 7.15. Deploying “Painting collaboration” on a distributed Environment.

7.5.4 Results and Performance Analysis

As expected, all the 30 drawn shapes have been successfully replicated and painted on the three GUIs of painter-applications besides the GUI of the team-application as shown in part (b) of Figure 7.16. The runtime elapsed for intercepting the painting CF for each painter is shown in Table 7.9. The time required for executing a complete individual CIMB interception includes:

1. The time needed by a **Painter** base object to dispatch the control flow into the Methods Dispatcher (MsD) in order to look for any applied CIMBs.
2. The time required by the MsD to prepare the Activated Remote Teams List (ARTL) when method **paint** is invoked.
3. Invokes the TLWM on remote team instance **coll** to execute the role’s callin method **collPaint**.
4. The time needed by the corresponding role instance to replicate the drawn shape on all hosts except the host of the intercepted remote base object. Note that this time actually includes the runtime needed to issue the remote COMB [`basePaint(Shape s) -> paint(Shape s);`].

As the results show, DOT/J framework could be applied with more users enrolled in the collaboration but with more overhead lies on the team instance **coll**; because all the remote CIMBs matched on all nodes will be dispatched to **coll** in order to invoke the role's callin method **collPaint**. The execution scenario might need to be repeated with more user participation to verify the DOT/J framework robustness and availability.

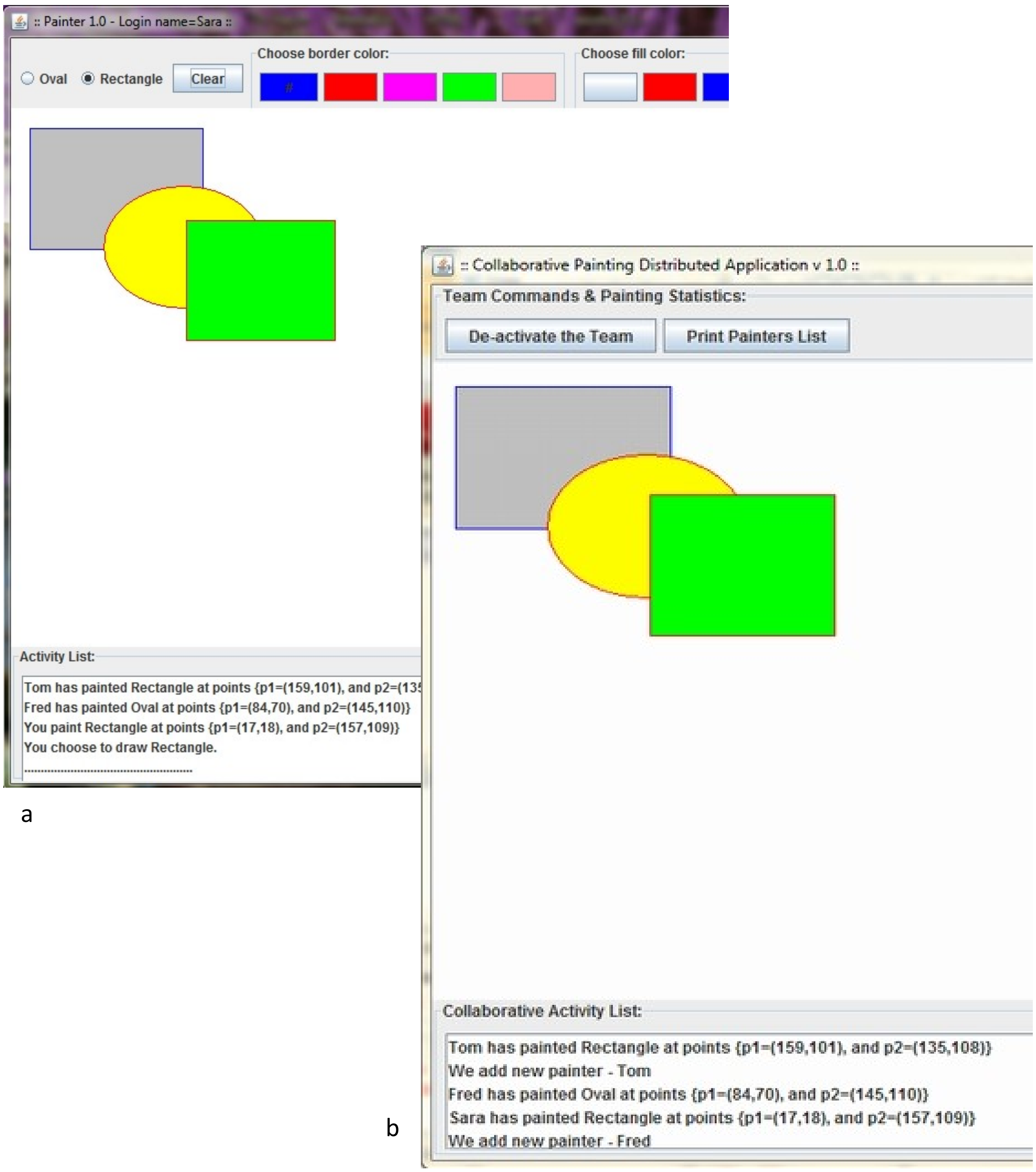

Figure 7.16. The Distributed Painting Collaboration Application in action: (a) the GUI of painter-application of "Sara", and (b) the GUI of the team-application after each painter has drawn his first shape.

Table 7.9. The runtime values (in milliseconds) for intercepting to the method `paint` in painter-applications deployed on hosts H1, H2 and H3.

No.	Sara (H1)	Fred (H3)	Tom (H2)	Avg.
1	50	30	20	25
2	16	7	14	10
3	10	7	13	8
4	12	6	6	7
5	14	6	13	10
6	17	10	16	12
7	8	7	15	9
8	18	10	13	12
9	11	11	9	10
10	17	6	13	12
Avg.	**17.3**	**10**	**13.2**	**11.5**

7.5.5 Avoid the Infinite Callin Cycles

According to the implementation of the collaborative application described earlier, when the behavior of remote **Painter** object is intercepted by the remote CIMB of **CoPainter** role, the role instance bounded to that object will execute the remote method-COMB [`basePaint(Shape s) -> paint(Shape s);`] on all other **Painter** objects (to replicate the drawn shape). This will call the base method **paint** on each of them, which is intercepted by the remote CIMB as well, and so on. In DOT/J framework, it has been observed that the purpose of binding a base method in a CIMB, and the purpose of binding the same method in a COMB are different especially in case of collaborative applications. That is, the base method **paint** is intercepted by the CIMB [`collPaint <- after paint;`] in order to realize the CF of **Painter** objects in the collaboration. But role instances invoke the same method via the COMB in order to *complete* the collaborating cycle.

When the DOT/J Transformation Library (DTL) transforms the base class **Painter**, it reformulates and creates a *shadow* method for each method in the base class. For example, for method **paint**, the DTL rename the method into "`paint$orig`", and create a new method with the name "`paint`", which traps the control flow when **paint** is called on **Painter** objects and dispatches it into the MsD (as discussed in Chapter 5). Now, it is possible to direct the role instances of **CoPainter** to intercept the "`paint`" method via CIMB and callout the "`paint$orig`" method; thus no cyclical interceptions could occur and, at the same time, the goal of collaboration is accomplished without violating application integrity. This idea has been consolidated in the DOT/J framework by telling the DTL to transform the role class **CoPainter** in such a way its instances will callout the "`paint$orig`" method when the **collPaint** method is invoked. To perform this, the attribute "`noCallinCycle`" is used and should be declared along the declaration of **CoPainter** role in the XML labeling file. To be set, it should given the value "`true`" (the default is that cyclic interception is enabled) as follows:

```
<RemoteTeam class="jo.abdullah.teams.CollPainting">
    <RemoteRole name="CoPainter" noCallinCycle="true"/>
</RemoteTeam>
```

7.6 Summary

This chapter has presented the implementation of various case studies to illustrate the using of the DOT/J framework in practice. It has verified the different features of the framework like remote CIMB, remote COMB, and remote BGPs evaluations. The chapter has demonstrated how to employ the DOT/J and the RRP to support dynamic adaptable distributed applications through adapting a simple client-server messaging application. Furthermore, the chapter has compared DOT/J to AWED (a distributed AOP language) and DandyJ (a dynamic aspect weaving approach). The results of execution assert the advantage of DOT/J over AWED especially in case of invasive interceptions. The chapter has introduced the employment of DOT/J framework in developing transparent distributed collaborative applications, and how to enable the legacy non-collaborative applications to enroll in distributed collaborations in simple, expressive, and clear fashion.

Chapter 8

Related Works

The DOT/J approach could be related to several state-of-the-art approaches in different fields. In specific, this chapter relates DOT/J to approaches in distributed aspect-oriented programming, the dynamic behavior adaptation of distributed application objects, and in the field of support for developing distributed collaborative applications in expressive and modular fashion. In addition, it reviews other approaches at which DOT/J has similar capabilities like supporting distributed components and service composition.

8.1 Introduction

The OT model is considered complex because it comprises several programming techniques. In specific, it implements the collaboration-based (role-based) design, and employs the concepts of the AOP in the object-oriented model. Therefore, the OT model could be related to several approaches in different programming fields [73]. By mapping the fundamentals of the OT model to distributed environments, DOT/J improves the modularity of distributed collaboration-based applications. The Remote Role-Playing (RRP); as a new promising programming technique for composing dynamic adaptable distributed applications, empowers DOT/J to support expressive distributed aspects modularization via remote roles. In the scope of distributed AOP, DOT/J could be related to approaches like AWED [47], DyMAC [11], JAC [70] and DJcutter [55]. Furthermore, it is possible to relate DOT/J to other AOP-based approaches that are developed mainly to support the dynamic adaptation of applications like the Two-Phased Aspect-Oriented (TPAO) [84], Transparent Reflective Aspect Programming for Java (TRAP/J) [81], and Dynamic Reconfiguration Systems (DyReS) [21]. The following sections discuss this in detail.

8.2 Distributed Aspect-Oriented Programming Models

Many distributed-AOP approaches have been developed during the last years. Some of them like AWED and DJcutter extend the aspect module and Joinpoint Models (JPMs) of local AOP approaches; for example, AWED extends JAsCo and DJcutter extends AspectJ. Mainly, these approaches augment the JPM of the extended models with remote pointcuts expressions, remote joinpoint matching, and remote advice execution mechanism. In addition, a distributed layer on which aspects could be deployed for weaving in applications is established. The following paragraphs will discuss some distributed AOP models, and compare them to DOT/J framework. Their capabilities to provide an expressive module that can capture the different crosscutting concerns (from a perspective of dynamic adaptation, the new functionalities distributed application objects will acquire when they adapt) will be discussed. This section discusses how they support remote joinpoint matching during application execution, and the mechanism they used to execute the associated remote advice. In addition, it will discuss the capability of these approaches to support the dynamic weaving of new aspects in distributed applications, and emphasizes the advantage of the dynamic RRP over this technique especially with respect to aspect weaving consistency and atomicity.

8.2.1 Aspects With Explicit Distribution (AWED) [47]

The AWED model has already been introduced in the introduction of this dissertation. However, to facilitate aspects deployment, AWED and DOT/J use a dedicated *repository* that must be installed at each node of a distributed application. Specifically, DOT/J maintains a DRS layer, which instantiates the central DOTM component at each host. In AWED, a dedicated *Connector Registry* is deployed likewise. The common feature between both repositories is that they store information about aspects and the declared pointcuts (remote CIMBs). A connector registry is responsible for handling the matching of local joinpoints and the local aspects that are deployed at the same node. In addition, a communication channel is connecting the separated connector registries. This channel broadcasts the joinpoint matching encountered at specific node to all other nodes; thus, the advice associated with the pointcut (i.e. which expresses that joinpoint) is *locally* executed at all nodes.

To support dynamic aspects weaving, all the methods of a target class must be *trapped* to dispatch to the local connector registry in order to execute advice(s) of the matched joinpoint(s). A DOTM component, on the other hand, is responsible for registering the remote team instances of application. In addition, it stores the anchors of base classes for all remote base objects that reside at the DOTM's node. The remote CIMBs (pointcuts) declared by remote roles are encapsulated in the Remote Teams List (RTL), which is reliably replicated at all DOTMs; the DRS uses a reliable multicast communication to enable DOTMs to communicate with each other in order to deliver the RTL to every affected base object at the precise time of execution.

In AWED, when a remote joinpoint is matched at runtime, the connector registry broadcasts the information of this joinpoint to the host (or hosts) mentioned *explicitly* in the pointcut definition. Hence the associated advice(s) are executed. This imposes the deployment of all aspects at every host. The local connector registry then dispatches the control flow to all intended hooks. In DOT/J, roles cannot leave the boundaries of team instances. Thus, they are not deployed on application nodes like AWED's aspects. Therefore, the remote execution of roles' callin methods (i.e. advices) is performed as remote method invocations. Specifically, the *intercepted* base object calls the Team Level Wrapper Method (TLWM) of the remote team instance in which the role declaring the callin method is enclosed. Hence, DOT/J provides a true remote advice execution, while AWED simulates that by executing local advices on local aspect copies.

The "`proceed`" statement in the remote advice "`around`" in AWED has local semantics [11], which means that the method of a local base object is executed instead of the method of the object in which the joinpoint has been actually matched. In DOT/J, the counterpart of "`proceed`" statement is the "base call" mentioned in replacement callin methods (i.e. the role's callin methods which bound with `replace` CIMBs). When a "base call" is encountered, role instance will dispatch the control flow to the bounded remote base object in which the remote CIMB has been *matched*. Nevertheless, DOT/J does not support *multiple* advice execution like in AWED. That is, advice execution is performed in the form of one-to-one in DOT/J; because for each base object a role instance is created. Also, role instances are not deployed on application nodes like in AWED. Thus, the execution of a remote callin method is handled at the node of role instance.

Regarding the dynamic loading and unloading of aspects, AWED reuses the facility of JAsCo, which allows application developers to define new connectors, and employ them automatically in application once they are *dropped* at the application class-path. Therefore, new aspects could be added dynamically at runtime. The success of AWED to enable new advices to be added at runtime resulting from that advices are not *hard coded* in application classes. Rather, aspects and application components are separated and connected only via connectors. Conversely, in OT/J, roles are woven *into* the bytecode of base classes at load-time, which results in a coherent team-role-base "trinity." Practically, DOT/J framework overlaps the obstacle of this cohesion by replacing the local *playedBy* relationship with the RRP relationship. Thus, new remote team instances could be dynamically attached to applications. Consequently, the target remote base objects can play the new roles without reloading their classes.

When a remote CIMB is matched in a specific remote base object, then that base dispatches transparently the control flow to the remote team instances included in the current Active Remote Teams List (ARTL). The ARTL is generated dynamically at the "trap-time" when a base method is called and then dispatched to the Methods Dispatcher (MsD). Therefore, the set of current *playable* roles is implicitly known, which allows a transparent remote advice execution. This involves a transparent "aspect advising" as well. In AWED, application developers *must* specify explicitly the scope of hosts – on which remote advices will be executed – using the keyword "`on`." This, in fact, destroys the separation of aspects' composition and deployment [11], and decreases the transparency level of the model.

Furthermore, the remote pointcuts of AWED can intercept only the visible methods of target classes (i.e. *public* methods). From a perspective of dynamic adaptation, this considered as a limitation in the model; because application objects must be able to adapt by changing *any* of their methods. In OT/J (hence DOT/J) this limitation does not exist; because the OT/J compiler allows application developers to declare CIMBs that bind roles' callin methods to *private* base methods.

Finally, DOT/J allows the role instance in which remote advice is executed to *access* its remote base object in which the remote CIMB of that advice has been matched. In specific, application developers can declare remote COMBs to invoke a specific base method, get a base field, or set a base field. This capability is not supported in AWED (neither all other distributed-AOP approaches); because advices are executed at local copies of aspects.

8.2.2 Dynamic Middleware for Aspectual Components (DyMAC) [11]

DyMAC is a middleware architecture that offers true and transparent distributed composition of aspect components. It uses aspect-oriented composition to connect the application logic to the middleware services. In few words, DyMAC attempted to leverage the object-based approach of aspects to the level of distributed object-based component models. DyMAC considers joinpoints as dynamic runtime conditions, which are typically specified with pointcut designators that describe the type and context of joinpoints. Contextual properties of a joinpoint may include caller information like its host address, and any other information that can constraint the condition. In DOT/J, application developers can declare remote base guard predicate (BGP) at the pointcut level (i.e. along the remote CIMB declaration), which could be used to provide contextual information about the bounded remote base object *explicitly*. Anyway, DyMAC stipulates that to express distributed joinpoints (or the runtime conditions) and their composition in distributed applications, a support is required with three key features: *remote pointcuts* that can evaluate on the invocation of remote methods, *remote advice* that could be transparently executed in a remote environment (while respecting the semantics of advice types like *around*), and *component semantics* of aspects in order to support third party composition and interaction with other aspects.

DyMAC has primarily targeted the component-based models, and DOT/J maps the OT programming model to distributed environments. Though, both present a high degree of transparency with respect to remote advices execution. Furthermore, the evaluation of joinpoints (or remote CIMBs) in DOT/J is carried out at remote base objects. At the same time, the constraints on joinpoints (i.e. remote BGPs) are evaluated remotely at the aspect side (precisely by remote team instances). At the composition level, DOT/J does not require explicit binding between role instances and remote base objects, which is a fundamental feature to support transparent dynamic adaptation. Contrarily in DyMAC, each component in the application needs to provide explicitly two interfaces: one to indicate how to create an instance from that component, and the other to declare component methods. Thus, application developers need to write extra code fragments to specify aspects deployment. In DOT/J, application developers only need to *label* remote classes once.

In addition, all application components in DyMAC must *inherit* the model-specific *ComponentInstance* class in order to bind a component instance to the DyMAC framework. In DOT/J, the remote base objects of applications need to provide only the GRBI, which facilitate the RRP activities without breaking the inheritance chain of their classes. In fact, this issue is important when composing aspects in legacy distributed applications; DyMAC cannot be used to adapt legacy applications because application hierarchy might be deformed.

Finally, in DyMAC there is no mention to synchronous or asynchronous support of advice execution [22]. In DOT/J, however, all advice execution operations are handled synchronously in order to preserve objects and application consistency especially when a specific remote advice invokes a remote field-COMB setter.

8.2.3 **Java Aspect Components (JAC)** [70, 71]

JAC is a dynamic AOP-framework that offers an aspect model to advice objects locally. In contrast to other AOP approaches, it does not introduce a dedicated aspect language; rather, it describes aspects in terms of regular objects. For this purpose, it has proposed the JAC objects. A JAC object could contain different groups of methods, namely: wrapping methods, role methods, and exception handlers. Wrapping methods provide the capability to execute *after*, *before*, and *around* advice code. Role methods, on the other hand, introduce new features in application objects. Andrew and Steve [10] have mentioned that current implementation of JAC supports only *around* advices.

However, JAC uses aspect weavers to deploy aspect objects onto application objects at runtime. A *wrapper controller* must be implemented by the application programmer in order to organize aspects composition at the weave-time. The JAC framework has been extended by notion of a distributed pointcut definition [47]. In fact, JAC simulates the semantics of remote advice execution by executing local advices on local copies of aspects (with this it resembles AWED), which imposes replicating all aspect objects at each host. In this way, JAC encounters the same problem of "`proceed`" statement mentioned earlier, i.e. the control flow is dispatched to local base object and not to the remote object at which the joinpoint has been originally matched.

Both DOT/J and JAC require components repository. JAC replicates its Aspect-Component manager at each host in order to manage the distributed deployment of aspect objects. In addition, JAC introduces a mechanism to preserve the state of similar deployed aspects consistent, which requires state synchronization at each state change. The expensive charges of this mechanism are averted in DOT/J; because role instances are enclosed inside their team boundaries and no need to state synchronization among role instances.

The separation between the weaver classes and wrapper controllers (which specify which wrapper methods should be invoked and in what precedence) of JAC enables the dynamic attachment of new aspects at runtime. The same strategy is supported in DOT/J but with different implementation. That is, the DOT/J framework relaxes the tight coupling between roles and base classes resulted from weaving these roles into base classes. Thus, application developers can define new remote teams, and then employ them dynamically at runtime thanks to the late-binding strategy adopted in DOT/J. In this regard, DOTM component takes care of the deployment of these remote teams, in addition to the pointcuts declared by their roles. In JAC, the weaving and unweaving of advices is handled at runtime, which add more overhead if the application is highly dynamic. In addition, the weaver classes implemented for composing aspects at runtime must take care of aspects interference, and prevent their consistency to be violated.

Finally, DOT/J applications are those written in OT/J, which are highly readable and understandable without being familiar with the infrastructure of any of its subsystems like the OTRE. In JAC, application developers need to have good knowledge of four type of programs; aspect programs, weaver programs, wrapper programs, and their base application.

8.2.4 Distributed Java Cutter (DJcutter) [55]

DJcutter extends a subset of AspectJ language constructs to make them behave as remote pointcuts. DJcutter extends the pointcuts of AspectJ with remote pointcuts to identify execution points (joinpoints) on a remote host. For this purpose, the pointcut `hosts()` is used. In addition, it supports declaring remote inter-type declarations, which enable developers to declare new fields and methods in a base class on remote hosts. Like DOT/J, the application classes targeted in aspect composition in DJcutter must be loaded by a specific class-loader in order to prepare them for aspect weaving on the fly. At runtime, all compiled aspects in DJcutter must be *manually* registered in the *Aspects Server.* The aspects server collects joinpoint information of remote pointcuts definitions, and executes the associated advices *local* to the aspect-server. That is, if a joinpoint has been matched on base object at host A, the associated advice will be executed on the aspect server.

In DOT/J, each remote team instance acts as an aspect server; because it contains the remote advices declared by its roles, and the execution of these advices is carried out on team instances. The advantage of using aspect-server in DJcutter is the capability to share and exchange data between aspects instead of synchronize them among multiple application nodes like in AWED and JAC. On the other hand, using a central aspect-server constitutes a bottleneck in large scale distributed applications, and forms a single point of failure.

8.2.5 Advantage of DOT/J

An approach for supporting distributed aspects has been proposed in [36]. The authors have discussed the problems found in current distributed AOP models like AWED. These problems include variations of aspects versions, the unrestricted access to system functionality (because aspects themselves are uncontrollable), etc. The presented approach adopts an actor-based architecture for supporting robust distributed aspects.

Over all these distributed AOP approaches, the DOT/J approach has the advantage of the remote team module which *encapsulates* the context of aspects' employment, i.e. the scope in which aspects (roles) are applied. Actually, none of the discussed approaches offer a counterpart module. Having this contextual modularization, application developers can manage aspects by the means of their enclosing teams. Therefore, remote roles in DOT/J (through mapping the OT model to distributed environments) give expressive meaning to aspect modularization, and at the same time they are controllable; e.g. via remote teams activation and deactivation mechanisms. The dynamic activation/deactivation of teams (thus dynamic playing of roles) is much efficient than the expensive weaving and unweaving of aspects at runtime especially in case of distributed-AOP approaches that do not support the activation/deactivation of aspects. Through activation priority values adopted in DOT/J, developers can control (even change) the order of role playing according to their needs without stopping application execution [3].

8.3 The Dynamic Adaptation of Distributed Applications

8.3.1 Preface

Let us quote a definition mentioned in [84] for the dynamic adaptability, says: "*Software is considered to be dynamically adaptive if conditions in the executing environment cause new code to be introduced at runtime to achieve new behavior not previously possible with the original code.*" From a perspective of the AOP, the realization of applications dynamic adaptability according to this definition should provide a modular mechanism to specify where, when and how application objects can adapt.

8.3.2 The Non-AOP Approaches

Most of the non-AOP approaches come with the conclusion that a *resource monitoring mechanism* should be established to detect the occurrence of changes in application executing environment. To facilitate this feature and to reduce the complexity of building adaptive distributed applications, Fransisco *et. al.* [23] have recommended that in addition to the implementation of distributed application functional-behavior some other underlying issues should be considered like specify the environment elements to be monitored, the mechanism to detect environmental changes, and which software adaptations should be handled and when?

Anyway, in the field of dynamic adaptation of applications, several non-AOP models and frameworks could be enumerated like Mol`eNE [41], HADAS [33], FORMAware [72], DA with COWS [38], etc. Basically, most of these approaches modify the architecture of the application in order to support adaptation. That is, they allow adding new components, removing existing components, and replacing application components at runtime. Mol`eNE is an object-oriented framework aims at enabling the dynamically adaptable mobile applications. It proposes a dynamic adaptation framework, which is divided into two main sub-frameworks: the detection/notification framework, and the reactive framework. The detection/notification framework monitors application resources and notifies components of changes, while the reactive framework is embedded in each adaptive component and is responsible for change the component's behavior as a response to the new execution conditions. In DOT/J framework, the DOTM component works similar to the detection/notification framework of Mol`eNE. More precisely, it facilitates the attachment of new functionalities through allowing new remote team instances to register dynamically, and then it notifies remote base objects about this registration. The MsD works as the reactive framework in that it coordinates the execution of roles' callin methods according to CIMB modifiers.

HADAS [33 and 35] is another non-AOP approach that allows adding, modifying, and removing code segments of components at runtime. It proposes a component model for encapsulating distributed services, and helps composing new applications from these services. A HADAS component includes two sections; called as fixed and extensible. The extensible section comprises the *mutable* parts of that component. Thus, it can change at runtime. The component performs changes on itself through the *meta-methods* it inherits from the parent of all HADAS components called "*Component.*" These meta-methods are responsible for structural and behavioral changes. The embedding of new methods in components, however, is not easy and error-prone because ambiguities might occur, which can impact application consistency. Furthermore, the cost of removing the woven methods might be expensive since several considerations must be taken into account like component inter-dependencies and functionality integrity. In addition, the architectural hierarchy of legacy applications could be broken; because each adaptable component *must* extend the parent "Component" of HADAS. In DOT/J, remote base objects can have new functionalities in terms of RRP, which enables the control over these functionalities, stopping/resuming them dynamically via team activation/deactivation mechanisms. Recall that a role in its essence is a "personality" of the base object inside team. Thus, with dynamic RRP, new personalities could be "created" for specific base object without changing the architecture of that base object.

8.3.3 The AOP Approaches

The following subsections discuss some AOP-based approaches that support the adaptation of distributed applications and the dynamic reconfiguration of systems.

8.3.3.1 *A Two-Phased Aspect Oriented Solution* [84]

In [84], a Two-Phased Aspect-Oriented (TPAO) solution to dynamic adaptation has been presented. The first phase (which takes place at the development time) identifies the points of adaptation in the application code. The second phase encompasses activities concerning the actual adaptation, include: check of adaptation conditions and *adding* or *removing* the code of adaptation. This phase takes place at runtime. It was shown that one of the major difficulties for achieving an AOP solution for dynamic adaptation is how to make the existing application *adapt-ready*, i.e. how to extend the application so that new functionalities could be loaded and unloaded at runtime dynamically.

The approach intercepts the core components of program by advices of aspects that act as *traps* in order to provide infrastructure support like the ability to insert/remove new functionalities. These functionalities, in turn, could be used to perform adaptation-specific processing. The DOT/J framework presents a similar technique. In specific, the MsD traps the invocation of *all* methods of a specific base class (as a first phase). Then, it checks at runtime for any (already gathered or newly deployed) remote CIMBs (pointcuts), and executes the associated roles' callin methods (as a second phase). In fact, DOT/J provides a generic layer on which the remote role-playing activities take place by using the GRTI and GRBI remote interfaces. This enables new adaptation code to be easily employed in and withdrawn from the application. By this technique (in addition to remote team activation/deactivation mechanisms), the problem of overlapping of *old* and *new* components (aspects) during adaptation [43] in TPAO approach and alike, which may lead to unpredictable and/or undesirable objects behavior, is avoided in DOT/J; because the whole RRP process is governed by the order and priority of teams' activation.

The TPAO approach, and similar approaches like in [14], have not yet support dynamic adaptation of distributed applications. In the survey introduced by Jorge and Siobhán [40] to assess the capability of dynamic adaptation approaches, one of the criteria used to compare approaches is their capability to cope with the *unanticipated* adaptation. However, the survey has studied the approaches of dynamic adaptation for centralized applications. Though, DOT/J is a promising approach that can support dynamic adaptation for unanticipated adaptation; because it allows new roles to be added during application execution without interrupt the execution of base applications.

8.3.3.2 ***Transparent Adaptation for Java Programs (TRAP/J)*** [81]

TRAP/J is a software tool for enabling developers to add new adaptable behaviors to existing Java applications transparently without the need of their source code. Similar to TPAO [84], TRAP/J uses a two-phase approach. In the first phase, TRAP uses aspects to provide the necessary *hooks* to realize runtime re-composition of application and produce *adapt-ready* program. At the second phase, new behaviors could be introduced via interfaces to the adaptable classes, which are *wrapped* versions of the original application classes.

Once again, the problem in TRAP/J is that the unanticipated adaptation of applications is not supported. That is, to enable the adaptation of application by changing functionalities that are not considered previously, application execution must be interrupted, and the first phase must be repeated. Moreover, TRAP/J support adaptation of single applications and not distributed applications. Thus, the encryption/decryption case study discussed in Chapter 6 cannot be implemented with TRAP/J; because there is no mechanism to synchronize the deployment of aspects. In addition, application consistency might be violated in TRAP/J; since no mechanism for resolving aspect weaving precedence is mentioned.

8.3.3.3 ***Dynamic Reconfiguration Systems (DyReS)*** [21]

DyReS is a Java-based framework for distributed dynamic AOP that offers coordination support for distributed adaptation in aspect-oriented middleware. DyReS observes that coordinating the weaving and unweaving of multiple inter-dependent aspects is verbose and error-prone task because structural integrity and global state consistency need to be ensured. However, DyReS does not support remote pointcuts, and the use of XML – to describe how to control aspects weaving – reduces the expressive power of the framework and decreases the degree of transparency of the dynamic adaptation. It was mentioned in [51] that an XML description to control the dynamic weaving of aspects in DyReS cannot be installed during runtime; rather, it must be installed statically on all nodes before application execution. In DOT/J, new remote team instances can enroll dynamically during application execution. Thus, DOT/J supports a dynamic weaving of aspects (roles) without stopping or reloading the application. In addition, the DOTM component supports the dynamic and transparent deployment of remote team instances.

With respect to application state consistency during the dynamic weaving and unweaving of aspects (roles), the base objects of application (which are affected by this weaving/unweaving of roles in DOT/J) are guaranteed to start playing the new woven roles in consistent manner thanks to the lazy-RTL update strategy, which enables base objects to reach a consistent state before they play the new roles. In addition, precedence of role-playing is resolved at the registration-time of remote teams by the DOTM components.

8.4 Supporting Distributed Collaboration-based Applications

The DOT/J framework maps the OT model to distributed applications. Thus, the DOT/J framework could be introduced as a technique for composing distributed collaboration-based applications. Several approaches in the scope of transparent collaborations between separated applications that are not implemented with this facility have been developed like TaKo [67] and Flexible JAMM [37]. TaKo is considered as an environment-specific approach as it targets the AWT- and Swing-based Java applications. DOT/J shares several features with TaKo; both approaches address the transparent collaboration between legacy applications. However, TaKo proposes a blackbox approach for supporting collaboration transparency. The DOT/J approach, on the other hand, requires application developers to implement the collaborative functionalities in teams explicitly. Thus, DOT/J tends to be a whitebox approach to achieve transparent collaboration. The main problem of blackbox approaches is that the collaborating applications *must* be identical. Otherwise, the collaborating will fail. This means that different versions of the same application might not be collaborate. Conversely, DOT/J allows developers to *connect* heterogeneous applications together in a specific team via enabling their objects to play different roles in that team. Using DOT/J as a tool to realize a whitebox transparent collaboration has been already introduced in Chapter 6 via a simple case study. However, developing a complete whitebox framework will be left as a future work.

The Collaborative Virtual Environments (CVEs) is another research field where a secure and integrated *shared environment* that provides to the geographically dispersed users the awareness they need to communicate and collaborate as if they were co-located in a real place. In this regard, AO-CVE [54] presents a proposal for employing the AOP concepts in the development of CVEs, which stands on intercepting the functionalities of application components, and dispatching the control flow to a dedicated middleware layer. This layer, then, interconnects components and aspects dynamically. The approach requires all aspects to be previously registered in this layer in order to ensure accurate interconnections. This requirement, however, is similar to registering remote team instances at runtime. But in DOT/J new remote team instances are allowed to be attached to the application. In AO-CVE, there is no guarantee that application consistency is preserved if new aspects need to be injected. Furthermore, the AO-CVE (as an AOP-based approach) lacks the support for clear collaboration modularity and expressive relationship between collaborative components and the declared aspects. DOT/J shows a high expressivity degree thanks to the expressive “playedBy” relationship.

Several approaches have been developed in the scope of collaborative software design and modeling environments like CoDesign [42] and GroupUML [61]. These and alike approaches cope mainly with problems like conflict detection in designed models, shared state and time synchronization, etc. In this regard, DOT/J enables application developers to control collaborations activities, and to manipulate the collaborative functionalities inside teams, which simplifies resolve any conflicts in the exchanged data between participants.

8.5 Distributed Components

Remote team instances could be introduced in distributed programming as distributed components. A remote team class can define both bounded and unbounded role classes; thus, it integrates service composition, collaboration-based design and application dynamic adaptation. The DOT/J framework supports two types of access to remote team instances: transparent access by remote base objects when they play their roles, and explicit access at the high abstraction level of programs via the remote façades and the GRTI remote interface.

The mechanism used by the DOT/J framework to look remote team instances up at runtime via remote façades is similar to the mechanisms used in CORBA [89] and EJB [46]; in specific, through the Java Naming and Directory Interface (JNDI) [92]. This mechanism stands on binding name-based identifiers to components, and then enables client programs to look them up later using these identifiers. In this regard, the DOTM uses the full-qualified names of remote team classes. Application developer has the choice to look up either a specific activated or un-activated remote team instance via calling the DOTM’s APIs. If a specific remote team does not provide remote façade, then application developers are able to access that team explicitly via the GRTI remote interface. For this purpose, the GRTI declares an API which allows invokes explicitly any *public* method of a remote team.

Chapter 9

Conclusions and Future works

This chapter documents the dissertation conclusions, and verifies the achievement of the goals mentioned in the introduction. Then, it discusses the obstacles that have been encountered during the implementation of the DOT/J framework, and suggests practical solutions to overcome them besides listing some future works.

9.1 Conclusions

The dissertation comes to the following conclusions:

1. The carrying out of Dynamic Adaptation of Distributed Application Objects (DADAO) requires application objects to be able to acquire new functionalities at runtime without reloading them or interrupting application execution.
2. Several approaches have been developed to employ the AOP concepts in distributed applications. They enable developers to declare aspects that can replace application objects' methods dynamically with new functionalities (advices); thus, these objects said to adapt and change their behavior according to the new functionality. These approaches suffer from several problems: primarily, the lack to modules for supporting explicitly the context of aspects' employment. In addition, application developers do not have the proper mechanisms to control the effects of aspects on application objects dynamically, which reduces their capability to specify how distributed applications can adapt and when?
3. Not all distributed-AOP approaches support the dynamic weaving/unweaving of aspects at runtime, which is a primary requirement to accomplish the DADAO. Approaches that support the dynamic weaving of aspects, on the other hand, suffer from several problems in consistency insurance of aspects and the target base objects. Beside their lack to perform atomic weaving/unweaving of aspects in/from multiple distributed objects in consistent manner.
4. The Object Teams (OT) model presents the "team" module as a context of role-playing. A team class binds roles to application objects via the expressive "playedBy" relationship. This dissertation has explained that the shortcomings mentioned above could be resolved through mapping the fundamentals of the OT model to distributed environments. For this purpose, the DOT/J framework has been presented.
5. The dynamic adaptation of applications reveals that application objects must be able to adapt to *unanticipated* changes in their executing environments. To achieve this in

DOT/J, base objects of distributed applications must be able to play new roles dynamically to acquire new functionalities. This entails relaxing the cohesion between base and role classes used in OT.

6. To accomplish this relaxation, the team and base classes, which involved in the Remote Role-Playing (RRP) relationship, should be transformed into remote team and remote base classes equipped with a *generic remote role-playing* capability.
7. The separation between the remote base objects and role instances of distributed applications imposes the implementation of a layer which can connect them at runtime in order to perform the RRP activities precisely.

9.2 Verifying the Dissertation's Goals

This section verifies the fulfillment of goals which have been mentioned in the introduction.

9.2.1 Introduce the notions of Remote Team, Remote Role, and Remote Base.

The dissertation has introduced a formal definition for the remote team, remote role and remote base concepts in Chapter 3. The modules that represent these notions in application development are the counterpart modules presented by the OT/J programming language. Thus, the semantics of these notions in the OT model are preserved. At the application design level, the *antenna symbol* ( ) has been invented to mark the remote classes involved in RRP relationships. In addition, remote team instances have been introduced as distributed components. The DOT/J framework simplifies the access of the services of these components through *Remote Façades*; a notion of the interfaces these components provided.

9.2.2 Enable the application objects and roles of "playedBy" relationships to reside on distinct hosts/processes.

The location of remote base objects and remote team instances (either the physical or the logical location) is not a requirement of the RRP relationship. This location independency of remote objects allows playing roles transparently from "*anywhere*." Thus, remote base objects, for example, can move from one application node to another without interrupting role-playing process. This capability strengthens the claim that DOT/J could improve the modularity of ubiquitous and mobile systems.

9.2.3 Achieve the DADAO through enabling remote base objects of applications to play dynamically different roles in different teams.

The adaptation of distributed application objects at the behavioral level means that an adapted object must preserve its original structure and modifies only its behavior. In DOT/J, application developers can define and attach new remote team instances in a running application without worrying about changes in the structure of player objects. In fact, role-playing does not affect structures of base players; rather, their behavior. In addition, role playing precedence could be easily governed via the activation priority values. The dynamic employment of roles involves a dynamic and transparent deployment of remote CIMBs at all application nodes through the reliable replication mechanism used by the DOTM component.

9.2.4 Allow the dynamic deployment of teams at runtime and playing roles without losing the original functionality of base objects.

It is easy to define and instantiate new remote teams in DOT/J. The DOT/J Runtime System (DRS) enables application developers to create remote team instances at any application node, and then put them to service in a plug-and-play fashion. The DRS is responsible for deploying remote team instances (their remote stubs) and the list of remote CIMBs declared in roles on all application nodes at runtime without interrupting application execution. After then, the affected base objects will receive a new *RRP-map* from the DOTM component, which is properly handed in. The role-playing process can take place then without these objects losing their original functionality in case that remote team instances have no longer available (due nodes crash).

9.2.5 Achieve a Secure Remote Role-playing and Safe Dynamic Adaptation.

A secure RRP points out to affect base objects' functionality only by remote CIMBs which have been defined in remote roles. The bounded role instances, on the other hand, should access their base objects only through the declared remote COMBs (which includes remote method- and field-COMBs), besides remote base guard predicates. At runtime, the DRS ensures that all COMBs will be handled by the accurate bounded remote base object; because the linking between roles and base objects is carried out by the mediation of the DRS itself.

Furthermore, DOTM is the *only* component that connects transparently remote base objects with remote team instances. Thus, base objects' functionality will be adapted only by the remote CIMBs encapsulated in the Remote Teams List (RTL) delivered to these objects by the DOTM as well. The operations of registration of remote team instances and the RTL replication are performed automatically in a transparent and reliable manner in DOT/J framework. Therefore, application programmers, for example, cannot register new remote team instances in the DOTM component manually. Moreover, the Java-RMI system supports the creation of objects' remote stubs from customized socket factories. Thus, customized socket factories that are augmented with the Socket Security Layer (SSL) could be used in order to enable the DRS to generate remote object stubs that are able to communicate over the SSL.

In addition, the DOT/J framework ensures a safe and consistent adaptation of application objects; because base objects use a *lazy-RTL update* strategy, which enables these objects to reach a consistency state before they play new remote roles.

9.3 Contributions

The dissertation contributions could be summarized as follows:

1. The concepts and implementation details of the Distributed Object Teams for Java (DOT/J) framework have been presented.
2. The Remote Role-Playing (RRP) has been introduced as a promising distributed programming technique which aims at enabling distributed application objects to play the roles of teams remotely in the semantics of *playedBy* relationship.
3. The RRP technique has been evaluated in several software design and development fields:
    a. In dynamic adaptable distributed applications.

   b. In the support of collaboration transparency between non-collaborative heterogeneous legacy applications.
   c. In the support of a modular distributed applications evolution.
   d. In distributed aspects modularization, composition, and deployment.
   e. In distributed components composition and deployment.
4. The dissertation has presented how the Dynamic Adaptation of Distributed Application Objects (DADAO) could be accomplished in a modular and expressive way using the RRP concept.

9.4 Obstacles and Future Work

In the current implementation of DOT/J framework, the remote base and team objects of distributed applications are represented as remote objects according to the Java-RMI disciplines. This strategy could cost expensive “execution charge” for two main reasons:

1. All methods of a remote base class (the declared and the inherited methods[1]) are *trapped* and *dispatched* to the Methods Dispatcher (MsD). When the MsD is invoked, the Activated Remote Teams List (ARTL) pertaining to the called base method should be prepared first, and then the manipulation of remote CIMBs involved in that ARTL is taking place. This involves the execution of callin methods of remote roles, which is performed *synchronously* as a Remote Method Invocation (RMI), i.e. remote base object is blocked until the execution of callin method is finished. Thus, remote base objects might wait for a long time before they reclaim the control flow. In addition, at every callin method execution, the base method’s arguments must be transferred over the “connection media” to every remote callin method involved in the ARTL. This could cause overload on the network if these arguments are of large sizes (e.g. images or document files).

2. The declared remote BGPs are actually evaluated by remote team instances each time a callin method has been executed. This means that remote base objects *must* wait for the result of evaluation even if it always “*false*.” This will consume part of the network bandwidth and could increase runtime overhead. The situation might become even worse if the base method, which causes the execution of that callin method, has passed a large number of arguments (which could be of large sizes as well). Then, these arguments need to be transmitted into the remote teams’ Team-Level Wrapper Method (TLWM), corresponded to the callin method, each time base method is called.

To reduce the overload on remote team instances and the network traffic (as a future work), the transmitted arguments of remote base methods could be *cached* by team instances, and a mechanism to preserve these arguments consistent could be implemented. In addition, the DRS must allow the *asynchronous* remote callin execution for remote CIMBs of “`after`” and “`before`” modifiers. This specification exempts remote base objects to wait until the control flow returns from the remote roles. However, this issue is not simple and

[1] See Chapter 5 – Section 5.4.3 for exceptions.

need more discussion; because it can violate application consistency if, for example, a callin method of an "`after`" CIMB invokes a remote field-setter that changes the base object state.

9.4.1 RRP at the Source-Code Level

DOT/J concepts like *remote team* and *remote playedBy* could be represented at the source code level of applications by extending the OTDT of the OT/J language. This may include the support of using model-specific *annotations* at program source code to work as compiler directives. For example, an annotation like `[@RemoteBase(name = "MyBase")]` might be declared in a team class to inform the compiler that the role class associated with the base class `MyBase` should be compiled as a remote role, and the enclosing team class as a remote team class. In addition, a keyword like "`remotelyPlayedBy`" can express the relationship between remote base classes and remote role classes rather than labeling them in XML files. This, of course, should not include the case of legacy applications in which the source code is not available.

9.4.2 The Limitation of adopting Java-RMI System

The DOT/J framework represents remote base objects and remote team instances according to Java-RMI disciplines. At the construction time, remote base objects and remote team instances *expose* their remote stubs "in the backstage" using the static API function `exportObject (object, port)` of the Java-RMI model-specific class `UnicastRemoteObject`. They pass the following parameters:

- `Object`: the current remote base or team object ("`this`" keyword is provided).
- `Port`: the port number at which the exported remote object will listen to remote requests (the value 0 is used in order to enable Java-RMI System to choose dynamically a free port).

Compulsorily, DOT/J uses this API since Java-RMI system only supports the dynamic creation of remote stubs (i.e. proxy objects) for those remote objects exported via this API. Therefore, the use of this API *prevents* base objects and team instances to expose further remote stubs with different remote interfaces (in fact, a "`RemoteException`" exception will be raised if the same remote object has exported another remote interface). In practice, this means that application developers cannot export remote objects at the source code via this API, which limits developing (Java-RMI)-based distributed applications coexistence with the RRP. From the opposite direction, legacy applications which already used this API could not be composed in the DOT/J framework either. As a future work, another middleware that allows representing remote base and team objects and dissolves this limitation should be used in DOT/J framework instead of Java-RMI.

9.4.3 The Problem of Object Serialization

One of the *indirect* problems in DOT/J is objects serialization. More specifically, object type arguments which might be provided to a specific remote base method must be serialized before they are dispatched to team instance side. The serialization process is required by Java-RMI system in order to marshal these arguments by the sender and un-marshal them by the receiver, and vice versa. In Java-RMI, a specific object is automatically serialized if the class, from which that object has been created, implements the marker interface `java.io.Serializable`.

Currently, DOT/J requests application developers to manually serialize the classes of object type parameters at the source code. As a future work, a mechanism for serializing parameters transparently needs be developed by the DRS, for example via bytecode transferring. This will add more flexibility to the framework.

9.4.4 Extensibility of DOT/J

The current implementation of DOT/J framework is based on Java-RMI middleware to provide the necessary communication for remote objects during the RRP. This results in a middleware-dependent framework. This restricts using DOT/J framework, for example, in "cloud computing"; because internet communication protocols need to be used. As a future work, more generic technique to implement that communication using approaches like Java-RMI over the Internet Inter-Orb (RMI over IIOP) [93, 6] might be used. Actually, RMI over IIOP integrates the best features of Java-RMI with the best features of CORBA [89].

9.4.5 Considerations of Type-Safety and Type-Checking Systems

Replacing the local references of base objects with the Generic Remote Base Interface (GRBI) stubs in remote role instances at runtime could cause type-checking side-effects. For example, instructions like [`if (`**`base instanceof`**` MyBase)`] will always return "`false`."; because the keyword **`base`** (which is a placeholder of the bounded base object) will be replaced by a remote stub. Thus, developers of OT/J applications must take into account this issue. Otherwise application integrity might be violated. The same integrity problem could be raised if remote roles are using the Java Reflection APIs on base objects at runtime. For example, profiling of bound base objects will be applied on the dynamic proxy objects and not on the actual base objects. In practice, this problem could arise in case of mapping legacy applications to the DOT/J framework.

9.4.6 DOT/J Evaluation as a Tool

In chapter 7, several DOT/J case studies had been evaluated. The validity of DOT/J had been verified for these case studies, but it is still needed to verify the validity of the DOT/J framework in general at the abstract level (i.e. for any case study that fits the scope of DOT/J).

9.4.7 Use a formal language to verify and validate RRP

In this dissertation, DOT/J framework has been verified through establishing a set of case studies. In these case studies, the performance of DOT/J runtime system and the accuracy of RRP application are the main the goals. As a future work, it is better to verify and validate DOT/J framework and the RRP concepts through formal languages.

APPENDICES

Appendix - A

The DOT/J APIs

This appendix presents the complete structure of DOT/J framework and gives a detailed description of each part by inspecting the main APIs defined by its modules. We use the UML Class diagrams to represent the main components compose the DOT/J framework, namely: the DOT/J Transformation Library (DTL), and the DOT/J Runtime System (DRS).

A.1 The DOT/J Transformation Library (DTL)

The DTL is shown as a UML Class Diagram in Figure A.1, which comprises two nested Java packages. The DTL is extending the OTRE transformers list; thus, its packages have been named according to the name of the OT/J' primary package. The DTL includes the following classes:

1. `DOTJTeamsPPTransformer`: This class implements the interface `org.cs3.jmangler.tau.CodeTransformerComponent`, which is dedicated to perform all code modifications at bytecode of the loaded classes. This class is the DOT/J *Transformer Composite* that transforms remote team classes (including remote role classes) of distributed OT/J applications according to the algorithm in Chapter 5. This class creates an instance of the class `DOTJTeamTransformer` which handles the actual bytecode transformations. The main method implemented by this class is `transformCode` which is the central JMangler method for transforming classes' code.

2. `DOTJBasesPPTransformer`: this class implements the JMangler's Code Transformation Interface (see Appendix-B.3) that performs all the necessary modifications to the bytecode of remote base classes. This class implements the method `transformCode` of JMangler interface `org.cs3.jmangler.tau.CodeTransformerComponent`. Once called by JMangler's Class-Loader, this method dispatches the transformation process to an instance of the class `DOTJBaseTransformer`, which applies the algorithm in Chapter 5 on remote base classes of distributed OT/J applications.

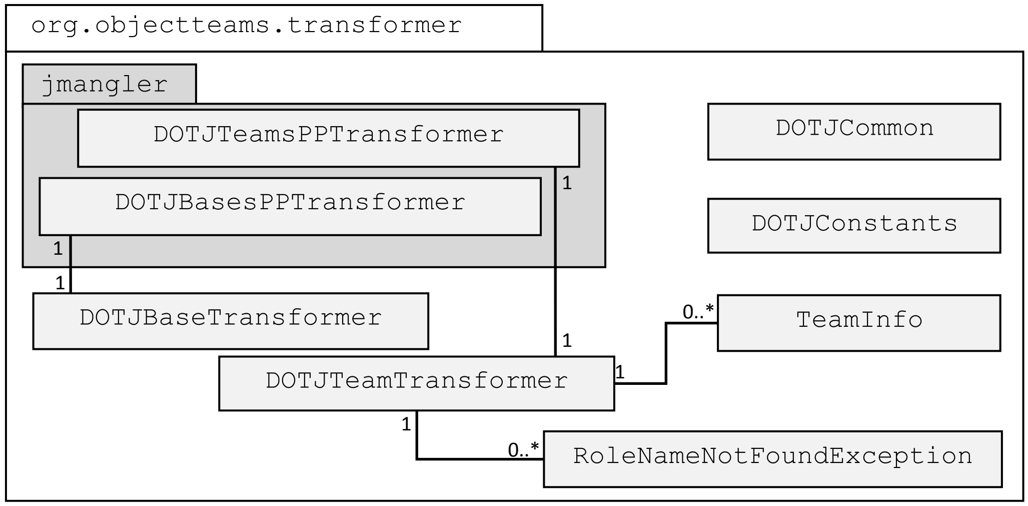

Figure A.1: The DOT/J Transformation Library Class Diagram.

3. `DOTJCommon`: This model-specific class declares the common methods that are used internally by DOT/J transformers. It includes the necessary methods for preparing bytecode segments or injecting common new methods in the team and base classes under transformation. Also, the DOT/J transformers might inject into classes' bytecode some method invocations to methods of this class like method `dotjFlagIsSet`, which is invoked at the application execution (e.g. from the `main` method) to check whether the "`-dotjFlag`" flag has been set or not.

4. `DOTJConstants`: We declare all the model-specific constants and attributes in this interface. These constants and attributes are used by DTL during the transformation process.

5. `TeamInfo`: This model-specific class is used by the Remote Team Bytecode Transformer (RTBT) during the XML labeling file parsing to build the list of team classes that must enter the transformation cycle by the RTBT.

6. `RoleNameNotFoundException`: if the application developer declares in the XML file a remote role name that is not exist or its name has been written with typo, then this exception will be raised.

A.2 The DOT/J Runtime System (DRS)

The DRS is the actual distributed layer on which the Remote Role-Playing (RRP) activities are carried out. The DRS is implemented in the DOT/J framework as a separate Java JAR library, which comprises mainly the GRIs of remote team and base classes, the DOTM component, and other framework-specific classes like the RTT and RBT constructs. The class diagram of the DRS is shown in Figure A.2, which has been organized in two nested packages.

A.2.1. The root package – `org.dotj.distribution`

This package contains the classes and interfaces which facilitate the RRP activities. Following is a list of these classes and interfaces:

- The `DistributedObjectsTeamsManager` class (the DOTM)

This class is the heart of the DRS. It declares three primary fields: the first is `theMainTeamsTable` of type `TeamsTable`, which organizes and stores the registered remote team instances. The second field is `theMainBasesTable` of type `BasesTable`, which likewise stores and organizes the anchors of the remote base objects resided in the JVM where the current local DOTM has been created. The third field is named as `peerCommChannel`. This field is of class type `JChannel`; one of the JGroups classes used to handle the communication between the distributed DOTMs.

This class declares the following APIs[1]:

<table>
<tr><td>invisible</td><td><code>+ String _OT$registerRemoteTeam
(String fullQTName, Object theTeamRemoteStub,
HashMap<String,ArrayList<TeamCallinRecord>> RCP,
int activationPriority)</code></td></tr>
<tr><td colspan="2">This API will be invoked transparently by the remote team instances of application during their creation to register in the local DOTM. The API requires the following parameters:
- <code>fullQTName</code>: The full-qualified name of the remote team class.
- <code>theTeamRemoteStub</code>: the remote stub exported by the registering team instance. This stub generated automatically either from the teams' GRI or the implemented remote façade (if any).
- <code>remoteTeamCallinFrames</code>: the list of all remote CIMBs (the remote pointcuts) declared by the remote roles of the registering team instance.
- <code>activationPriority</code>: the associated Remote Team Activation Priority (RTAP) value, or 0 if no value has been assigned.
This API returns a string value which represents the Remote Team Unique Identifier (RT-UID) of the registering team instance.</td></tr>
</table>

[1] We classify the APIs into *Visible* and *Invisible* to point out to the possibility to invoke them from client programs or not. APIs marked as "visible" can be invoked by client code.

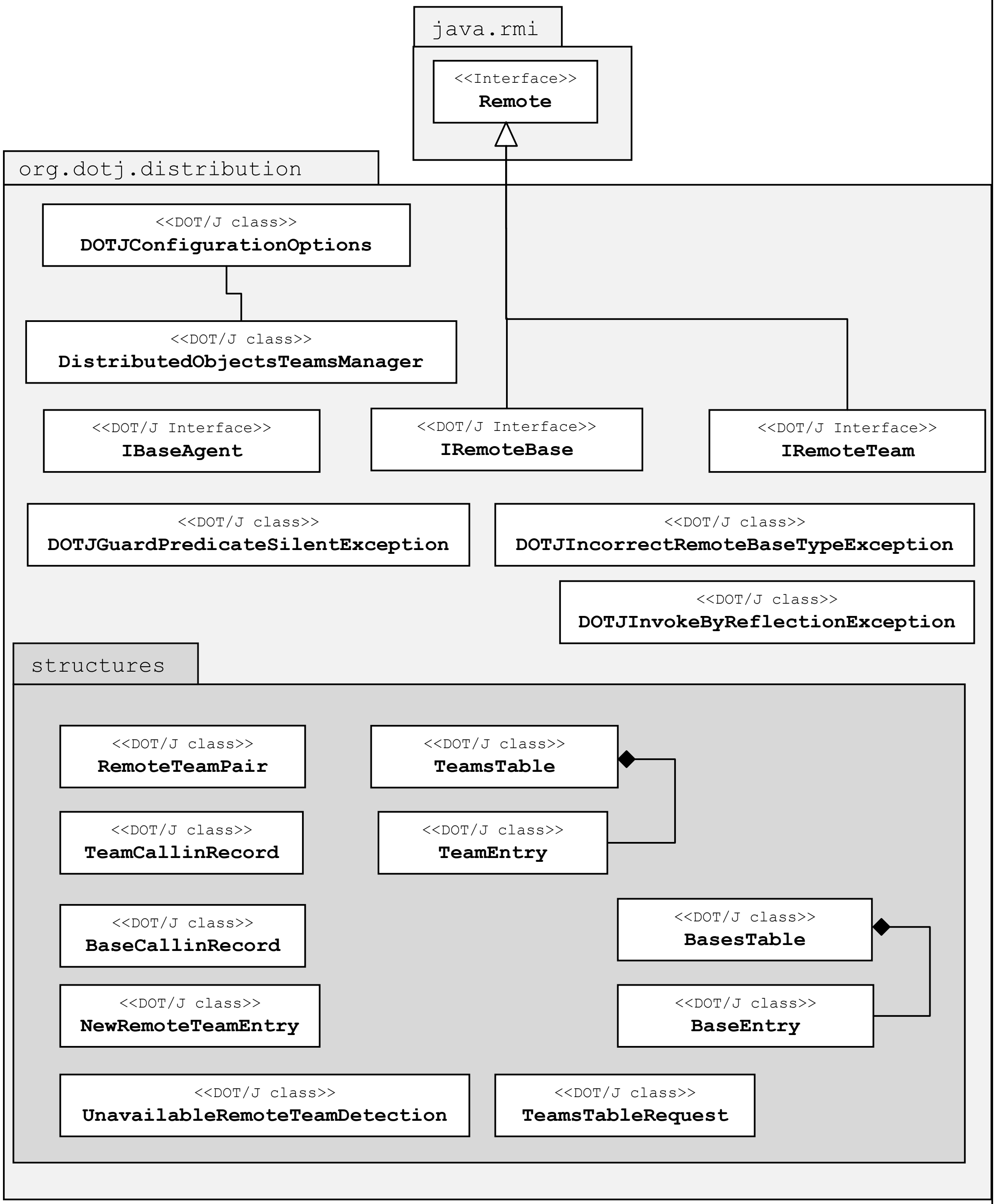

Figure A.2: UML Class Diagram of the DOT/J Runtime System (DRS).

invisible	**+ void** _OT$registerRemoteBase (**String** fullQualifiedBaseClassName,IBaseAgent theBaseLocalAgent)
This API will be invoked transparently by remote base objects in order to register their *anchors* in the local DOTM. When called, base objects should pass the full-qualified name of their classes and the anchor interface they implement.	

Visible	**+ static** DistributedObjectsTeamsManager getCurrentDistributedObjectsTeamsManager()
This API is used internally by remote base objects and remote team instances to grip a handle to the local DOTM when they register. This API could be invoked at the client code catch the local DOTM, for example, to invoke its different visible APIs like the inquiring for activated team instance of specific team type (see the next APIs).	

Visible	**+ Object** getRemoteTeam(**String** remoteTeamClassName) **+ Object** getActivatedRemoteTeam(**String** remoteTeamClassName) **+ Object** getRemoteTeam(**String** remoteClass, **String** hostOrIP)
These APIs could be invoked by application developers to fetch the remote stub of a specific remote team instance. The first API will grip the stub of any available remote team instance of that designated team class name. The second API will try to fetch a stub for a currently activated remote team instance. Since all the DOTMs in a distributed application have the same copy of the RTT, application developers can obtain remote stubs for remote team instances from remote JVMs. Therefore, the third API enables application developers to obtain the remote stub of the remote team instance resides in a specific host (i.e. a specific JVM). Actually this is important when two (or more) remote team instances of the same team class have been instantiated at different nodes.	

Invisible	**+** LinkedHashMap<RemoteTeamRecord, **ArrayList**<TeamCallinRecord>> _OT$getRemoteTeamsList(**String** baseName)
The remote base objects of applications invoke this API transparently to request the latest copy of the RTL from the DOTM. Initially, a remote base object invokes this API for the first time when any of its methods has called for the first time. Then, it invokes this API when it has been notified by the DOTM due to RTL updates.	

Invisible	**+ void** _OT$UnavailableRemoteTeamDetection (**String** unavailableTeamUID, **String** detectorBaseName)
When a specific remote base object detects the unavailability of a remote team instance during any remote CIMB execution, it invokes this API to inform the DOTM of this detection. The detector base object supplies the RT-UID of the detected team instance and the name of its own base class. The DOTM uses the detector class name to exclude the base objects of that class from being notified with the team unavailability.	

Invisible	**+ boolean** `_OT$isReady()`
When an application starts executing on top of the DOT/J framework, a local DOTM instance is created automatically at the host of that application. If that DOTM instance is not the first to create, then the completion of DOTM instance creation is assured when it obtains the RTT (which represents the RTL) from the DOTJ group's coordinator. This API is called transparently by the DRS to indicate that the created DOTM is consistent.	

Invisible	**- void** `send(`**Address** `receiver,` **Address** `sender,` **Serializable** `msg)`
This private method is used by DOTMs to realize the communication between their instances. Parameter `msg` holds the actual message to be sent, and `receiver` and `sender` parameters represent the final destination node and source node, respectively. If `receiver` is `null`, then the message is broadcasted to every member in the group. This method, in fact, wraps the counterpart method of `JChannel` class in JGroups system.	

Invisible	**- void** `_OT$performAvailabilityRound()`
The coordinator DOTM performs a periodic remote teams' availability check cycle. For this purpose, a thread is created to invoke this private API to check the availability of each remote team instance registered in the RTT. After the check cycle is completed, the DOTM updates the RTL for each remote base class in its RBT if required.	

- The `DOTJConfigurationOptions` class

This special-purpose class implements a set of APIs that are used to configure the runtime system of the DOT/J framework (DRS). It declares a set of global fields that hold information about framework's executing environment like DOTJ group name, the time of the periodic availability check cycle performed by DOTMs, etc.

visible	**+ static void** `setRMIsecurityProperties()`
This API first customize the Java-RMI system execution according to the properties fit well with the DOT/J requirements (see Appendix-B.1), and then it creates a local DOTM instance. This API is invoked statically from the application's main method *iff* the DOT/J flag is set (see Appendix-C.4).	

- The `IBaseAgent` interface

This interface enables each remote base object to provide an anchor to register in the DOTM. This interface declares the following methods:

invisible	**+ void** `_OT$nudge()`
When an RTL update is performed by the DOTM (due to new remote team instance registration or unavailability detection of a registered one), then it notifies all the affected remote base objects by this update via invoking this API on all remote base objects (per base class) locating at the same JVM. When called, it sets RTL-flag into "`true`", which means that remote base objects *must* download the new RTL before the invocation of the next method.	

invisible	**+ boolean** `_OT$isAlive()`
The DOTM invokes this API during the notification of remote base objects with new RTL updates to make sure at least one base object from the base class to be notified is remaining alive.	

- The `IRemoteBase` interface

This interface realizes the Generic Remote Bases Interface (GRBI) through which roles can interact remotely with their remote base objects. As shown in Figure A.2, this interface extends the specific Java-RMI `Remote` interface. However, this interface declares the following methods:

invisible	**+ Object** `_OT$invokeThisMethodInBase` (**String** `methodOrField`, **Object**`[] args`, **int** `flag`, **String** `methodSignature`)
We have called this method as the Generic Invocation by Reflection Method (GIRM). It is invoked transparently by role instances on behalf of remote COMBs and remote BGPs evaluation. This method requires the following parameters: - `methodOrField`: The name of method to invoke or field to set/get. - `args`: the arguments required by the method being invoked or the value of the field being set on base object packaged in an array of objects. - `flag`: indicates the type of invocation (1 – for method COMB, 2 – for field setter, and 3 – for field getter COMB invocation). - `methodSignature`: the signature of the base method being invoked, or the signature of field to be set. The method returns a value of type `object` (which may be a boxed value that needs to be un-boxed at the caller role instance), or `null` in case that the remote base method is of type `void`, or when a field setter is issued.	

invisible	**+ boolean** `_OT$checkType(`**String** `baseClass,`**long** `classSerializableID)`
This API is invoked transparently by remote team instances prior the binding process between the current remote base object (its remote stub) and a role instance. A remote team instance invokes this API remotely on the remote base object to make sure it has been generated from the same base class mentioned in the right-hand-side of the *playedBy* relationship declared by the desired role class. The team instance supplies the full-qualified base class name and the CSID of that base class. If the invocation returns “`true`”, the binding is proceed, else that (i.e. “`false`” is returned or a remote exception has occurred) the binding process stopped.	

- The `IRemoteTeam` interface

This interface realizes the GRTI provided by remote teams. It declares the following methods:

invisible	**+ Object** _OT$invokeThisMethodInTeam (**String** methodName, **Object** source, **Object**[] args, **String** methSignature)
This is another GIRM method that enables remote base objects invoking the TLWMs associated to the remote CIMBs declared by remote roles. When invoked, the following parameters must be supplied: - `methodName`: The name of the TLWM to invoke. - `source`: The exported remote stub of the remote base object sparking the call. - `args`: the arguments required by the TLWM packaged in an array of objects. If no arguments is required then `null` will be passed instead. - `methSignature`: the signature of the desired TLWM, which is important for arguments un-boxing. When a replacement remote CIMB is encountered, the return value of this method must be handled properly by both the called team instance and the caller remote base object.	

invisible	**+ boolean** _OT$isAlive()
This API is invoked to make sure a remote team instance is still alive. This method is invoked transparently by the DOTM when it performs the periodic availability check, and by the MsD when preparing the ARTL due to a base method call.	

visible	**+ boolean** remoteActivate() **+ boolean** remoteDeactivate()
These two APIs could be invoked by application developers to, respectively, activate and deactivate remote team instances. When invoked, these APIs activate/deactivate remote team instance *globally* for all their local application threads.	

invisible	**+ boolean** _OT$isActiveT()
This API is invoked by remote base objects, and just prior the execution of any remote CIMB on specific remote team instance, in order to check the activation status of that team instance.	

visible	**+ Object** remoteCall (**String** methodName, **Object**[] arguments)
When application developer catches a handle of a specific remote team instance, he can issue remote method invocations to any *visible* method on that team instance. The developer needs to pass the name of the desired method, and the arguments required by that method packed in an array of objects. As this method is a generic reflective invocator, the expected result must be properly un-boxed or casted by the developer.	

- The `DOTJGuardPredicateSilentException` class

Remote base objects of an application are imperceptive to the remote BGP evaluation process. Thus, when a remote CIMB is matched, the remote base object invokes the corresponding TLWM on a remote team instance. Then, any remote BGPs must be evaluated at this time. If the remote BGP has been evaluated to “false” then the process of remote

CIMB execution *must* be *silently* ended; because evaluating BGPs is a matter of team instances and not remote base objects. To achieve this, the invoked TLWM returns a new instance of this exception class. Once it arrives to the caller remote base object, it drops the remote CIMB and traverse the next one.

- The `DOTJIncorrectRemoteBaseTypeException` class

This exception is raised if a remote base object found invalid when being bounded to a role instance (see the description of method `_OT$checkType`).

- The `DOTJInvokeByReflectionException` class

This exception is raised whenever a problem occurs during the execution of either a remote TLWM via the GIRM of remote teams, or a remote COMB via the GIRM of remote base objects.

- The `DOTJDOTMNotFoundException` class

This exception is raised when fetching an invalid DOTM instance, or when the application has executed before a local DOTM is installed (this occurs when remote base and team classes have been transformed by the DTL, but the "`dotj`" flag has not been set – See Appendix-C.5).

A.2.2. Package of internal structures – `org.dotj.distribution.structures`

This package comprises the "supportive" staff that completes the structure of DOT/J Runtime System. It mainly comprises those structures that are responsible of storing and organizing remote objects and their stubs, namely; the Remote Teams Table (RTT) and the Remote Bases Table (RBT). Anyway, in this section we will browse the classes of these structures and describe each of them briefly.

- The `TeamsTable` class

This class represents the repository at which the DOTM stores registered remote team instances. In fact, the whole registration process of remote team instances is carried out by an instance of class preserved in each DOTM; specifically, the DOTM dispatches all registration requests along with the necessary information of remote team instances toward this instance. Thus, it is responsible for receiving registration request, creates and organizes remote team registration records, keeps track of these records, and answers DOTM inquiries when a specific remote team instance is to be looked up.

In case of remote team instance registration, the DOTM invokes the following method of `TeamsTable` instance. The method returns the registered record as an instance of class type `TeamEntry` so that caller DOTM can broadcast this new team entry to all group members.

```
+  TeamEntry addTeamEntry
(String fullQTeamName, Object remoteTeamGRIStub,
HashMap<String, ArrayList<TeamCallinRecord>> teamCallinFrames,int remoteTAP)
```

This method requires the following parameters; which are (as we mentioned in Chapters 5 and 6) prepared automatically by the DOT/J transformers:

- `fullQTeamName`: The full-qualified remote team class name.
- `remoteTeamGRIStub`: The remote team GRI stub generated by registering team instance or remote façade that instance implements.
- `teamCallinFrames`: The complete CIMB expressions list that registering team instance confines. This list is prepared by DOT/J transformers according to the disciplines discussed in Chapter 5, and organized as follows: "for each remote base class, group all declared remote CIMBs in spite of the bounded base-level method".
- `remoteTAP`: the value of RTAP associated for registering remote team instance; the default is `0`.

Additionally, this class declares other methods to remote team's entry, look up remote stub of specific remote team instance, etc. Recall that all `TeamsTable` instances of all DOTMs must hold the same set of team entries.

- The `BasesTable` class

This class represents the Remote Base Table (RBT) that acts as a local repository of all remote base objects instantiated at the JVM where each DOTM is instantiated. Thus, each of these DOTM preserves an instance of this class type. Fundamentally, this class holds the anchors of remote base objects once they register in the DOTM, organizes the registered remote base objects records, and delivers all update notifications that DOTM issues due to RTL updates to all affected remote base objects.
The main methods this class declares are:

```
+  void addBaseEntry(String baseClassFQName, IBaseAgent baseAgent)
```

Whenever an object is instantiated from specific remote base class, it registers in the local DOTM that has been created in the same JVM. The DOTM forwards the registration order to the instance of `BasesTable` it preserves by invoking this method. The only parameters required to fulfill registration process are:

- `baseClassFQName`: The remote base object's full-qualified class name.
- `baseAgent`: The anchor interface that registering base object implements.

Internally, the RBT instance organizes remote base records by first storing information within records of type `BaseRecord`. Each `BaseRecord` instance represents the group of similar remote base object records; i.e. for each remote base class, the anchors of registered base objects of that class type are listed together in single group. For example, the RBT shown in Figure 6.3 (Chapter 6) at Host1/Process1 is organized as follows:

```
                      BaseRecord = ("org.package.MyBase"), List= { Anchor_b1
                                                                   Anchor_b2
                                                                   Anchor_b3
                                                                   Anchor_b4 }
BasesTable rbt = {
                      BaseRecord = ("org.services.Vertex"), List= { Anchor_v1
                                                                     Anchor_v2
                                                                     Anchor_v3 }
                 }
```

Figure A.3. A "snapshot" of the Remote Base Table at the H1/P1 shown in Chapter 6 - Figure 6.3.

```
+  void updateBasesList(TeamEntry teamEntry)
+  void updateBasesList(TeamsTable rtt)
+  void updateBasesList(TeamsTable rtt, String excludeThisBase)
```

We explain that the DOTM notifies all remote base objects with any changes that occur on the RTL they interested with. In this regard, the DOTM dispatches the performance of this process to the RBT instance it holds. The later is meant to take care of how the notification must be done, and which of remote base objects are in target? Let us first recall that notifying remote base objects with changes of RTL is triggered either when new remote team instance is registered, when DOTM receives URTD message (that indicates the unavailability of specific remote team instance) from one of DOTJ group members or when an unavailable remote team instance is detected during the periodic cycle of unavailability check, or when a specific remote base object detects the unavailability of specific remote team instance.

For each situation of these notifications, this class declares a corresponding method to carry out that type of notification, so only that set of remote base objects affected by RTL updates are notified. The first method is invoked whenever a new remote team instance is registered. In this case, the RBT searches among its records for remote base classes that have roles to be played in this new remote team instance. Then, it uses the first founded anchor for each remote base class to nudge that class. Nudging a remote base class will force a valid instance of that base class type to download the new RTL from the DOTM; so that all instances of that base class type will have this new RTL.

The second method is used to notify all remote base objects registered in the RBT with changes of RTL that result from unavailable remote team instance detection. Note that the absence of specific remote team instance may seriously change the map of remote role-playing. Thus, a global notification process is performed. The same notification process is carried out by the third method when a specific remote base object "snitches" the unavailability of remote team instance; except that this remote base object (i.e. its base class) is excluded from this notification as it already reformulate its copy of RTL.

The classes remaining in this package (`RemoteTeamPair`, `TeamCallinRecord`, and `BaseCallinRecord`) are used internally for further organization and facilitation of remote role-playing support.

Appendix – B

The Technologies used in DOT/J Implementation

The DOT/J Runtime System (DRS) operates on top of two off-the-shelf technologies, namely; the Java-RMI Middleware, and the JGroups Communication System. The first technology is used to facilitate and support the communication channel in RRP between remote base objects and remote team instances. More precisely, the DRS relies on Java-RMI to execute remote CIMBs and remote COMBs, and to evaluate remote BGP expressions. The second technology provides a reliable communication between DOTMs components their selves to preserve the consistency of RTLs. Additionally, the DOT/J Transformation Library uses JMangler (along with BCEL) to handle the bytecode transformation of remote team, remote role, and remote base classes. This Appendix presents these techniques according to the requirements and disciplines of DOT/J framework.

B.1 Java-RMI System

Java-Remote Method Invocation (Java-RMI) is a distributed object model for Java programming language that uses Remote Method Invocation (RMI) to enable the programmer to create distributed Java-based to Java-Based applications [94]. Basically, Java-RMI allows a remote Java object locating in specific JVM to invoke the methods of another remote Java object that locates in different JVM.

In this section, we will discuss only those Java-RMI specifications signify the DOT/J framework. The detailed specifications of Java-RMI could be found at [83, 88, 94, and 6]. Thus, we will emphasize the mechanism provided by Java-RMI to implement remote objects. Then, we discuss, within the allowed scope of Java-RMI customization, which of Java-RMI properties have been customized in DOT/J.

B.1.1 Create Remote Java Objects

Java-RMI runtime system uses the *Proxy*-based technology to implement remote objects [10]; which stipulates that remote objects need to export specific proxy instances at the JVMs where their methods will be remotely invoked. When a method is remotely invoked, the dedicated proxy instance dispatches that invocation properly to the actual object, and returns any expected result as well.

Anyway, to implement remote Java objects in Java-RMI, these objects should implement a specific *remote interface* that declares those methods that will be invoked remotely by clients. To create a remote interface, application developers simply need to add the special interface `java.rmi.Remote` to its “`extends`” clause. Additionally, all declared remote methods must *throw* the remote exception `java.rmi.RemoteException` or one of its subclasses.

Afterward, objects could be exported into distributed environment either by extending the model-specific class `java.rmi.server.RemoteObject` or its subclass `java.rmi.server.UnicastRemoteObject`; which defines a singleton remote object, and then transparently exports remote objects. In this case, class hierarchy of legacy applications might be deformed. The second option is to use the APIs of the class `UnicastRemoteObject` explicitly. This way, the hierarchy of classes is preserved. Unfortunately, this option restricts remote objects to export only one remote stub (the first option, actually, allows any number of remote interfaces to be implemented and exported).

In case of explicit remote objects exporting, application programmer use one of three different formats for the `exportObject` method of class `UnicastRemoteObject` to export a remote stub of a specific remote object, which are:

```
Remote exportObject(java.rmi.Remote obj, int port)                  B.1
RemoteStub exportObject(java.rmi.Remote obj)                        B.2
Remote exportObject(Remote obj, int port,
                    RMIClientSocketFactory csf,
                    RMIServerSocketFactory ssf)                     B.3
```

In all cases, a remote stub is required. In this regard, Java-RMI system offers two options for generating the remote stubs of remote objects. The first one is explicit, where application programmers need to *compile* remote classes (after they compile them with `javac`) using the RMI compile utility `rmic`. Then, the generated remote stub classes will be *downloaded* at the client side once a remote object is being accessed. After downloading the required remote stub, a new proxy instance is created which implements those methods that are declared in the remote interface.

While this mechanism compels application developer to recompile and prepare for binding and downloading remote stub classes, it might become more verbose in case of exporting large number of various distributed applications objects. Since Java 1.5, however, Java-RMI system supports the process of generating remote stubs and deploying them dynamically at runtime, i.e. there is no need to recompile classes with `rmic`. Appreciably, the implicit generating of remote stubs (via proxy objects) improves and simplifies distributed application development. In DOT/J, we export remote bases’ and remote teams’ stub respectively from the GRBI and GRTI interfaces. Also, we use the automatic generation of proxies to add more transparently and flexibility.

To access a remote object and invoke its remote methods, client programs need to *look* the remote stub of that remote object *up* at the Java-RMI *Registry*. For this purpose, Java-RMI implements the interface `java.rmi.registry.Registry` to provide a bootstrap service for registering and retrieving remote objects by using simple names [83]. This mechanism requires application developer to *bind* the remote stub of a specific remote object in the registry of the server side with a specific name (ID). Then, clients *look* that remote stub *up* by inquiring the registry passing its ID. In fact, this is one of two methods used to deploy remote objects for remote access. The second *convenient* method for accessing remote objects is by passing their remote stubs to the client side as parameters. In DOT/J framework, we adopt this mechanism to pass the remote stubs of remote base objects (generated from the GRBI) to remote team instances as parameters of the TLWM.

In the format (B.3) of `exportObject` method, Java-RMI allows the use of customized version of *Socket Factories*. In practice, Java-RMI uses two *default* socket factories to hide the verbose details of the low-level creation and manipulation of sockets: the first one is used to create *server* sockets, which are used by remote objects to respond to method invocations. The other socket factory is used to create *client* sockets that client programs use to invoke remote object's methods. However, developers can create their own customized socket factories via implementing the interfaces of these factories [6]. Thus, security utilities, encryption/decryption mechanisms, object compression, etc, could be incorporated in these customized factories. For example, we can implement client and server socket factories that can use the SSL/TSL security layer to secure the communication between remote base objects and remote team instances in the RRP.

B.1.2 Remote Method Invocation

Java-RMI aims at enabling the invocation of remote object methods the same way their local methods are invoked. When a client issues a call to one of a remote object's methods, Java-RMI system handles the execution as follows:

1. All arguments other than remote objects are *passed-by-copy*. For arguments of primitive types like `int`, `char`, etc, their values simply copied to the destination (the called remote object). If the passed argument is of object type, then the content of that object is copied using the *object serialization* of the Java platform. This implies that classes of object types *must* support the serialization process; either by implementing the `java.io.Serialzable` interface or by implementing manually the serialization mechanism.
2. Passing remote objects is carried out by *pass-by-reference.* Thus, whenever a remote object is passed, its remote stub is dispatched instead. This fact remains true even if remote objects are passed as part of state of an ordinary object. At the destination side, the passed remote stub is used to generate a new proxy instance which represents the remote object. However, if a remote object is passed twice in the same method invocation, then only one remote stub is generated but two references are used at the destination JVM.
3. The control flow is transferred into the called remote object while the caller is blocked. The caller resumes execution once the result of invocation is returned, or when a remote exception occurs.

B.1.3 Customize the Properties of Java-RMI System

Java-RMI enables developers to customize, to some extent, the behavior and model-specific functionality of RMI by change specific properties. In this section we explain those properties that are used in DOT/J and the values they have been assigned:

1. *The security policy*. Java-RMI system requires a security manager to be installed before any of remote invocation activities is taking place. For this purpose, Java-RMI advises the creation of new `RMISecurityManager` instance to control the communication between nodes over its system. The installation of this security manager accomplished as follows: a security policy of Java platform must be set first, which organizes the permissions of resource allocation and usage. The security policy could be set as follow:

   ```
   System.setProperty("java.security.policy", "dotj.Policy")
   ```

   The argument "`dotj.Policy`" is the name of the file where permissions are listed. In DOT/J, the DTL transformers automatically generate and assign this file at fly, which adds more security and safety.
2. *A transparent and automatic generation of proxy objects*. To enable the automatic generation of proxies from the remote stubs of remote objects, Java-RMI requires the following property to be set to "`true`" (the default is "`false`"):

   ```
   System.setProperty("java.rmi.server.ignoreStubClasses", "true")
   ```

B.1.4 Using Java-RMI in DOT/J

Basically, we use Java-RMI middleware to realize the communication between remote base objects and remote team instances in RRP in order to facilitate executing remote CIMBs and remote COMBs, and evaluating the remote BGP expressions. Practically, we employ Java-RMI in DOT/J framework as follows:

1. *Embedding Java-RMI in DOT/J.* In DOT/J, we set the properties of Java-RMI at an early stage of application execution and before any remote team or base objects have been created.

   For this purpose, the DOT/J transformers inject at the method "`main`" of the application starter class a call to the static method `setRMIsecurityProperties` of class `DOTJConfigurationOptions` (see Appendix-A). This method mainly installs a new security manager, and creates the required security policy file at fly. This method is also set the RMI property required to enable the automatic generation of proxies, and creates a DOTM instance at the local node.
2. *Generate the remote stubs of remote teams and remote base objects.* We adopt in DOT/J framework exporting remote stubs of remote base and team objects through invoking the static method `exportObject` of class `UnicastRemoteObject` of format (B.1). We pass the value `0` to the parameter `port` which lets Java-RMI to assign dynamically a suitable port number at runtime[1].

B.1.5 Advantages and Disadvantages of Java-RMI

The Java-RMI model as a middleware to develop distributed object-oriented programming has several advantages. Here, we mention some of them:

[1] See for examples: Listing 5.3 line 5 and Listing 5.15 line 4 in Chapter 5.

1. *Compatibility*. Java-RMI is totally implemented for Java using Java itself. Thus, application developers will not get worry about distributed applications compatibility to Java. Consequently, we insure the compatibility between the Java-RMI part used in the DOT/J framework and OT/J.
2. *Transparency*. Java-RMI hides the cumbersome code needed for creating the sockets and the low-level communication between stubs. In addition, it presents a convenient invocation of remote methods that looks very similar to local invocations, and preserves the same call semantics. Moreover, it supports the dynamic generation of proxy objects without using the Java-RMI registry.
3. *Platform independent*. Since Java is platform independent language, then distributed applications developed by Java-RMI work properly on top of any platform.
4. *Simple and clear to implement*. This leads to more robust, maintainable and flexible applications.
5. *Maintain a safe environment for applications on Java platform by installing security managers and using specific class loaders.* As we discussed earlier, the security manger is one of Java-RMI requirements to enable communication between remote objects and clients.
6. *Provided with a dedicated Distributed GC*. Java-RMI Garbage Collector is responsible for collecting the dead remote references and proxy objects of remote objects.

Java-RMI, on the other hand, is a Java-based technology. Thus, non-Java applications cannot make use of Java-RMI. This shortage comes from the fact that remote objects implemented with Java-RMI export only Java-based remote interfaces. Besides, the followings could be considered as disadvantages:

1. *Non-Extensibility*. Java-RMI does not allow using, for example, new communication protocols between remote objects.
2. *Thread-Unsafe*. Java-RMI cannot guarantee thread-safety when several clients invoke the same remote method concurrently. In this regard, DOT/J uses synchronized methods, which prevent, for example, two remote base objects to access the same team method simultaneously.
3. *It is "Hackable" if the ordinary code deployment is used.* Generating remote stubs of remote objects using "`rmic`", and then stores them in a specific server (e.g. HTTP server) does not prevent replacing these stubs with "infected" stubs, which can cause serious deformations to application's functionality. In DOT/J framework, we do not adopt this mechanism; thus, the safety of application code against intrusion is ensured.

B.2 JGroups: A Reliable Multicast Communication System

JGroups [86] is a toolkit for reliable group communication that uses the terms *member* and *group*; the first expresses the communicating nodes while the second represents cluster of nodes. Therefore, developers can create groups of processes whose members can send messages to each other. The one significant feature here is the *reliability*. JGroups supports a reliable unicast and multicast message transmissions, i.e. lost messages are retransmitted. Also, the system keeps track of all members in each group, and notifies group members when a new member joins or an existing member leaves or crashes. In DOT/J, we create the group “DOTJ” which all DOTM members must join.

In JGroups model, the centric component in its architecture is the “`JChannel`” class. For a process to join a group, it has to create a channel and connect to it using the name of that group. The created channel is then the handle to of that group through which a member can send messages to a single member or to group, and receive messages from group. Channels of JGroups are operating on top of protocol-stack that implements the properties specified for a given channel. It contains a number of protocol layers in a bidirectional list where all messages sent and received over the channel have to pass through all protocols. However, members of a group are communicating asynchronously when using channels. To support synchronous communication, JGroups presents Building Blocks to provide more sophisticated APIs on top of a channel.

B.2.1 Creating `JChannel` instances and Joining the “DOTJ” Group

The process of creating channels and joining specific group is very simple in JGroups. Application developers need only to create a new `JChannel` instance passing the desired configuration of protocol stack as a parameter, then connect that channel to a specific group through channel’s API `connect` passing group name as a parameter. We use in DOT/J the UDP as transport protocol; which uses multicast or multiple unicasts to send and receive messages. Therefore, we create a channel at the construction of DOTM as follows:

```
1 public DistributedObjectsTeamsManager()
2 { .
    .
3  channel = new JChannel("udp.xml");
4  channel.setOpt(channel.LOCAL, new Boolean(false));
5  channel.setReceiver(new ReceiverAdapter()
6  {
      :
7  }
8  channel.connect(DOTJConfigurationOptions.getDeafultGroupName());
   :
9 }
```

At line 3 we create a new channel using the configuration XML file provided with JGroups, and in line 4 we prevent the channel to process its own sent messages. Line 8 is the actual joining process where we connect all DOTMs channels to one specific group. To enable channel instance receiving messages from group members, it has to create a `ReceiverAdapter` instance which receives every message from group members and allows any further manipulations.

B.2.2 Sending Messages

Developers can use channel’s APIs to send data in the form of messages to the whole group or to a specific member in that group. For this purpose, `JChannel` class implements the `send()` methods. This method has several formats, but we use in DOT/J the most familiar one:

```
public void send(Address dst, Address src, Object obj)
```

Where `dst` is the destination node that will receive the message. If null, then the message will be sent to all members of group. Parameter `src` represents the address of the sender, and application developers can pass null instead so that the transport protocol (e.g. UDP) will replace it with the accurate address. Finally, the payload is passed in parameter `obj` that has to be serialized. Internally, `JChannel` creates an instance of type `org.jgroups.Message` from these arguments, and then sends that instance properly.

The DOTMs send messages between each other mainly to preserve the consistency of the RTL. More specifically, DOTM uses its channel to send messages in order to achieve the following tasks:

1. Obtain the Remote Teams Table (RTT) when a new DOTM is created. At the construction time, the created DOTM joins the group of "DOTJ", and if it is not the first member to join, it sends `TeamsTableRequest` message to the DOTJ group coordinator. When the coordinator receives this message, it immediately sends back the current RTT wrapped in a message to the designated DOTM.
2. Broadcast the registration of new remote team instances. When a remote team instance register in the local DOTM, then that DOTM sends a `NewRemoteTeamEntry` message (including information bundle about that team instance) to the entire group. Thus, all members (except the sender) will receive this message and add the new remote team entry to their RTTs accordingly.
3. Announce remote team unavailability. During the execution of remote CIMBs, if the remote base object detects a specific remote team instance unavailable, then it contacts the local DOTM to inform it with this detection. Accordingly, if that DOTM makes sure of the unavailability of that remote team instance, then it sends `UnavailableRemoteTeamDetection` message to the DOTJ group including the RT-UID of the unavailable team instance. Therefore, all DOTMs can reformulate the RTL they hold and preserve its consistency.

B.3 JMangler: A Framework for Java class files load-time transformation

JMangler [29] allows the Load-Time interception and transformation of application classes. It provides a modified version of the system class `ClassLoader` to integrate with JVM so that it can load all application-specific classes before they arrive to JVM for execution. As Figure B.1 illustrates, JMangler framework is incorporating with JVM class loading system by providing the Transformation Manager interface which interacts with the Composition Algorithm that activates and coordinates the action of one or more Transformer Components (or simple Transformers). In turn, Transformers analyze the loaded classes and specify the modifications to be done.

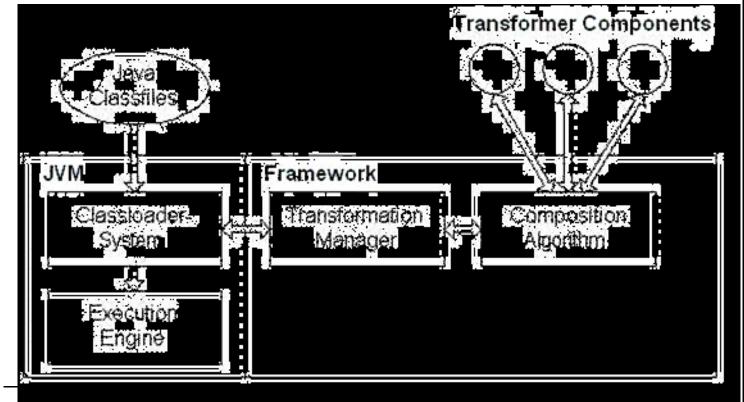

Figure B.1: The global architecture of JMangler framework.

However, JMangler enables developers to develop their own Transformers to extend/replace the capabilities of JMangler transformers. In this regard, developers need to organize their transformers, regarding the order of transformation, and modify the Document Type Description (DTD) that JMangler provides. Then, the user-defined transformer (i.e. other than JMangler original transformer) is a Java class that implements `org.cs3.jmangler.tau.InterfaceTransformerComponent` interface for interface transformation and/or `org.cs3.jmangler.tau.CodeTransformerComponent` for code transformation. The first one enables developers to add a class, method, or field to the class under transformation, in addition to class hierarchy modification and increase class, method, or field visibility. The second interface enables changes to the JVM code of methods, i.e. the instructions that JVM executes. In this regard, JMangler uses BCEL APIs to handle these changes.

In Chapter 5, we have illustrated that DOT/J extends the bytecode transformers list of OT/J by appending extra two transformers; one is dedicated to transform remote team classes and the other to transform remote base classes. While OT/J is currently using JPLIS APIs instead of JMangler, we leave the implementation of DOT/J transformers on top of JPLIS as a future work. Thus, all examples and case studies in this dissertation are using JMangler.

Appendix - C

Implementation of Case Studies of Chapter 7

This appendix contains the implementation of case studies discussed in Chapter 7. The code of DOT/J applications is written entirely in OT/J with ODTD v 1.3.3 using Eclipse SDK Version: 3.5.2 - Build id: M20100211-1343. The code of AWED application is written in JAsCoDT — A JAsCo Plug-In for the Eclipse IDE v 1.4.10 (Copyright © 2002–2005 JAsCo Development Team).

C.1. A Simple Distributed OT/J Application

Class: **MyBase** (full-qualified-Name: `de.evaluation.c1.bases.MyBase`):

```
package de.evaluation.c1.bases;
public class MyBase {

  private String myID;
  private int[] myVals;

  public MyBase(String baseId)
  {  this.myID = baseId; }

  public void setVals(int[] newValsSet)
  {  this.myVals = newValsSet; }

  public String whoAmI()
  {
       return "instance of MyBase class";
  }

public static void main(String[] args)
{
  System.out.println("Case Study 1 - Node : H2\n #Press ENTER to know who am I?");
  MyBase b2 = new MyBase("base_b2");
  b2.setVals(new int[]{6,12,9,7});

  String s;
  //============================================
  // Press ENTER to know who am I.
  while(true)
  {
   try {
```

```
      (new BufferedReader(new InputStreamReader(System.in))).readLine();
      s = b2.whoAmI();
      System.out.println("I am " + s);
      } catch (IOException e) {  e.printStackTrace();}
  }
 }
} // EOC - End of Class
```

Team: **MyTeam** (full-qualified-Name: `de.evaluation.c1.teams.MyTeam`):

```
package de.evaluation.c1.teams;
public team class MyTeam {

      private int THS = 34;

      public int sum(int[] arr)
      {
            int t = 0;
            for(int i=0; i<arr.length; t+=arr[i++]);
            return t;
      }

  public static void main(String[] args)
  {
      System.out.println("Case Study 1 - Node : H3");
      MyTeam t1 = new MyTeam();
      //======================================================
      System.out.print(" #To toggle between activation status" +
                        " of t1 press ENTER:");
      //    Initially, t1 is NOT active
      while(true)
      {
       try
       {
        (new BufferedReader(new InputStreamReader(System.in))).readLine();
         if(t1.isActive(ALL_THREADS))
                  t1.deactivate(ALL_THREADS);
         else
                  t1.activate(ALL_THREADS);
         System.out.print("-Activated? -" + t1.isActive(ALL_THREADS));
        } catch (IOException e) {e.printStackTrace();}
       }
    }
}// EOT - End of Team
```

Role: **MyRole** (full-qualified-Name: `de.evaluation.c1.teams.MyTeam$MyRole`):

```
protected class RoleA playedBy MyBase
  base when(sum(base.myVals)>= THS) //remote BGP
{
  callin String whoAmI()
  {
      String postFix = RoleA.class + " in team " + MyTeam.class;
      return base.whoAmI()+ ", ID= "+getBaseId()+" playing role"+postFix;
   }

  whoAmI <- replace whoAmI;                // remote CIMB
  String getBaseId() -> get String myID;   // remote COMB (field-getter)
```

```
}
```

C.2. A Comparative Case Study: a Simple Chatting Application using Distributed Aspects

Class: **Messenger** (full-qualified-Name: `de.evaluation.c2.bases.Messenger`):

```
public class Messenger {

	boolean active = false;
	private String id = "";

	BufferedReader in =
		new BufferedReader(new InputStreamReader(System.in));

	public Messenger(String _id) {
	super();
		this.id = _id;
	}

	public void StatusActive(){
		active = true;
	}

	public void StatusInactive(){
		active = false;
	}

	public String getID()
	{
		return this.id;
	}
	public String input(){
		String resp ="";
		try {
			resp = in.readLine();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return resp;
	}

	public void print(String s){
		System.out.println(s);
	}

	public void processMessage(String s){

	}

	public void startConsole(){
		while(true){
			String resp = input();
			if(resp.equals("active")) StatusActive();
			if(resp.equals("inactive")) StatusInactive();
			if(resp.equals("Exit")){
```

```
                        try {
                              in.close();
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                        System.exit(0);
                  }
                  long a,b;
                  a = System.currentTimeMillis();
                  processMessage(resp);
                  b = System.currentTimeMillis();
                  System.out.println("runtime = " + (b-a));
            }
      }

      /**
       * @param args
       */
      public static void main(String[] args)
      {
      BufferedReader in =
                 new BufferedReader(new InputStreamReader(System.in));
            String userId = "GUEST-" + in.hashCode();
            System.out.print("Enter a user ID:");
            try {
                   userId =  in.readLine();
            } catch (IOException e) {
                   userId = "GUEST-" + in.hashCode();
            }

            Messenger m = new Messenger(userId);
            m.StatusActive();
            m.startConsole();
      }
}
```

C.2.1. The AWED Implementation

Aspect**: MessengerAsp** (full-qualified-Name:`de.evaluation.c2.aspects.MessengerAsp`):

```
class MessengerAsp {
   hook MessengerHook {                             // The Hook construct
   static int c = 0;
   MessengerHook(method(..args1))
   {
      execution(method) && !joinpointhost(localhost); // The pointcut ..
    }
   before()                                              // The Advice ..
   {
     System.out.println("MSG#" + ++c + ": " + (String)
           thisJoinPoint.getArgumentsArray()[0] );
    }
  }
}
```

Connector**: MessengerCon**:

```
static connector MessengerCon {
      all.pkg.MessengerAsp.MessengerHook hook0
        = new all.pkg.MessengerAsp.MessengerHook(void
                              all.pkg.Messenger.processMessage(String));
      hook0.before();
}
```

C.2.2. The DOT/J Implementation

Team**: ChattingTeam** (full-qualified-Name: `de.evaluation.c2.teams.ChattingTeam`):

```
public team class ChattingTeam {
      // to register the Messenger users - remove the comment
      //LinkedList<Chatter> l = new LinkedList<Chatter>();
      static int c = 0;
      public static void main(String[] args) {

            (new ChattingTeam()).activate(ALL_THREADS);
            System.out.println("Team is Ready ..");
      }
}
```

Role**: Chatter** (full-qualified-Name: `de.evaluation.c2.teams. ChattingTeam $Chatter`):

```
protected class Chatter playedBy Messenger {
      public void beforeProcessMessage(String msg)
      {
            /* remove the comment to enable the remote COMB
             for(Chatter ch : l)
             {
               if(!ch.equals(this))
                  ch.printAtAll(msg);
             }
            */
            System.out.println("MSG#" + ++c + ": " + msg;
      }

      public void beforeActive()
      {
            l.add(this);
      }

      // remote CIMB
      beforeProcessMessage <- before processMessage;
      beforeActive <- after StatusActive; // to register the base

      // remote COMB - remove the comment to enable the remote COMB
      //void printAtAll(String s) -> void print(String s);
 }
```

C.3. Multiple Remote Role Playing

Class**: BaseX** (full-qualified-Name: `de.evaluation.c3.bases.BaseX`):

```
public class BaseX {
      private String baseID = "BaseX";
```

```
      private int[] intArray = new int[] {10,20,30,40,50};
      public String baseMethod(){return this.baseID;}
}
```

Class**: BaseY** (full-qualified-Name: `de.evaluation.c3.bases.BaseY`):

```
public class BaseY {
      private String baseID = "BaseY";
      public String baseMethod(){return this.baseID;}
}
```

Team**: RemoteTeamA** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamA`):

```
public team class RemoteTeamA{

}
```

Role**: ARole1** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamA$ARole1`):

```
protected class ARole1 playedBy BaseX base when(base.intArray[2]<9)
{// Remote Role
    private String roleID = "ARole1";
    callin String baseMethod()
    {
       return this.roleID +":"+ base.baseMethod();
    }

   baseMethod <- replace baseMethod; /*remote CIMB */
}
```

Role**: ARole2** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamA$ARole2`):

```
protected class ARole2 playedBy BaseY
{// Remote Role
    private String roleID = "ARole2";
    callin String baseMethod()
    {
       return this.roleID +":"+ base.baseMethod();
    }

   baseMethod <- replace baseMethod; /*remote CIMB */
}
```

Team**: RemoteTeamB** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamB`):

```
public team class RemoteTeamB{

}
```

Role**: BRole1** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamB$BRole1`):

```
protected class BRole1 playedBy BaseX
{// Remote Role
    private String roleID = "BRole1";
    callin String baseMethod()
    {
       return this.roleID +":"+ base.baseMethod();
    }
```

```
   baseMethod <- replace baseMethod; /*remote CIMB */
}
```

Role: **BRole2** (full-qualified-Name: `de.evaluation.c3.teams.RemoteTeamB$BRole2`):

```
protected class BRole2 playedBy BaseY { // Local Role Playing
   private String roleID = "BRole2_LOCAL";
   callin String baseMethod()
    { return this.roleID +":"+ base.baseMethod();}

   baseMethod <- replace baseMethod; // local CIMB
}
```

C.4. Encryption/Decryption of Client-Server Messaging Application

Class: **Client** (full-qualified-Name: `de.evaluation.c4.bases.Client`):

```
public class Client {
 private Socket s;
 private PrintWriter os;
 private BufferedReader br;
 public boolean connect(String host, int port) {
   try {
      this.s = new Socket(host, port);
      os = new PrintWriter(s.getOutputStream(), true);
      br = new BufferedReader(new InputStreamReader(s.getInputStream()));
      return true;
   } catch (UnknownHostException e)
            {…
              return false;
            }
 }
 public void send (String msg)
  {
   String response = "NULL";
   if(s.isConnected())
    {
      os.println( msg);
      os.flush();
      try {
            response = br.readLine();
      } catch (IOException e) {...}
                  System.out.println("RESP:" + response);
     }
     else
       System.out.println("Not Connected!!");
 }
}
```

Class: **Server** (full-qualified-Name: `de.evaluation.c4.bases.Server`):

```
public class Server extends Thread{

private String serverId;
PrintWriter os;
BufferedReader br;
```

```
Socket client;
private String cID = "";
private boolean running;
public Server(String id, Socket cl)
 {    running = true;
      this.serverId = id;
      this.client = cl;
      this.start();
 }
public void run()
{
 while(running)
 {
  try {
  br = new BufferedReader(new InputStreamReader(client.getInputStream()));
  os = new PrintWriter(this.client.getOutputStream(), true);
  String s = br.readLine();
  processMsg(s);
  if(!s.startsWith("X0X"))
      respond("Acknowledge!!");
  else
  {.. }
  } catch (IOException e) {..}
  }
 }

 Public void processMsg(String ms)
 { System.out.println(" -- MSG --: " + ms); }
}
```

Team: **DecryptionTeam** (full-qualified-Name: `de.evaluation.c4.teams.DecryptionTeam`):

```
public team class DecryptionTeam  {
  protected class DecryptedServer playedBy Server {
      callin void processMsg(String s)
         { base.processMsg(decrypt(s));}
      private String decrypt(String msg)
      {           :
                  return ..;
      }
        // remote CIMB ..
        processMsg <- replace processMsg;
      }
}
```

Team: **EncryptionTeam** (full-qualified-Name: `de.evaluation.c4.teams.EncryptionTeam`):

```
public team class EnrcyptionTeam {
IRemoteTeam  dt = null;               // a handler to the decryptor-team
DistributedObjectsTeamsManager dotm;  // a handle to the local DOTM
///// Roles ..
protected class EncryptedClient playedBy Client
  base when(decryptionTeamIsDeployed())
{
            callin void send(String msg)
            {
                   base.send(encrypt(msg));
            }
```

```
            private String encrypt(String msg)
            {      :
                   return ..;
            }

            send <- replace send;
      }

public EncryptedClient()
{
  this.activate(ALL_THREADS);
  try {
      dotm = ..;// get the local DOTM ..
      dt = (IRemoteTeam) dotm.getRemoteTeam("..DecryptionTeam");
      if(dt != null) // the decryptor-team is deployed ..
      {
            dt.remoteActivate(); // activate it remotely
      }
  } catch (Exception e) {..}
}

private boolean decryptionTeamIsDeployed()
{
   if(dt == null) // not yet gripped ..
   {
      dt = (IRemoteTeam) dotm.getRemoteTeam("..DecryptionTeam");
      if(dt != null)
      {
        dt.remoteActivate(); // activate it remotely
        return true;
      }
   }
   else
      return dt.isActive();
 }
}
```

C.5. Executing the DOT/J Applications

To execute the DOT/J applications listed in this Appendix:

1. Add the DOT/J Library (`org.dotj.1.0.0.jar`) to the build path of application.
2. In the "**Run Configuration**" dialogue, specify which class comprises the "`main`" method, and add the "`-dotj`" flag to the **Program arguments** (found under the tab "`(x)=Arguments`") to allow creating new DOTM instance at start.
3. To execute DOT/J applications over a LAN network, enforce Java to use IPv4 by setting the property `java.net.preferIPv4Stack` to `true`; in the "**Run Configuration**" dialogue, under the tab "`(x)=Arguments`", add `-Djava.net.preferIPv4Stack=true` to the **VM arguments**.

List of Figures

Appendices

List of Listings

List of Tables

List of Publications

- A. Al-Zaghameem. *Extending the model of Object Teams/Java programming language to distributed environments*. In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '10). ACM, New York, NY, USA, (2010).

- A. Al-Zaghameem. *Supporting Distributed Aspects by Extending Object Teams Model into Distributed Environments*. The 2nd Conference on Information and Communication Systems - ICICS'11, Irbid, Jordan, (2011).

- A. Al-Zaghameem. *An Expressive Approach to Distributed Applications Dynamic Adaptation*. International Journal of Computer Science and Information (IJCSI), Volume 9, Issue 4, July, (2012).

- A. Al-Zaghameem. *Toward Expressive Dynamic Adaptation of Distributed Applications*. The 8th International Conference in Computer and Networking Technologies (ICCNT'12), Gyeongju, Korea, August 27-29, 2012, (2012). (Accepted paper).

References

Published Papers

[1] A. Al-Zaghameem. *Extending the model of ObjectTeams/Java programming language to distributed environments*. In Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '10). ACM, New York, NY, USA, (2010).

[2] A. Al-Zaghameem. *Supporting Distributed Aspects by Extending Object Teams Model into Distributed Environments*, The 2nd Conference on Information and Communication Systems - ICICS'11, Irbid, Jordan, (2011).

[3] A. Al-Zaghameem. *An Expressive Approach to Distributed Applications Dynamic Adaptation*. International Journal of Computer Science and Information (IJCSI), Volume 9, Issue 4, July, pp. 57 – 66, (2012).

Books

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2 (1995).

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*, 3rd Ed., Sun Microsystems, ISBN 0-321-24678-0. Addison-Wesley. (2005).

[6] W. Grosso, *Java RMI*, O'Reilly, 1st ed., ISBN: 1-56592-452-5, (2001).

Research References

[7] A. Dantas, J. Yoder, P. Borba and R. Johnson, *Using aspects to make adaptive object-models adaptable*. In RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Oslo, Norway (2004).

[8] A. de Moura, C. Ururahy, R. Cerqueira, and N. de La Rocque Rodriguez, *Dynamic Support for Distributed Auto-Adaptive Applications*. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCSW '02). IEEE Computer Society, Washington, DC, USA, pp.451-458, (2002).

[9] A. Popovici, T.Gross, and G. Alonso. *Dynamic weaving for aspect-oriented programming*. In Proceedings of the 1st international conference on Aspect-oriented software development (AOSD '02). ACM, New York, NY, USA, pp.141-147. (2000).

[10] A. Stevenson and S. MacDonald. *Smart proxies in Java RMI with dynamic aspect-oriented programming*. IEEE International Symposium on Parallel and Distributed Processing – IPDPS 2008, pp.1-6, April (2008).

[11] B. Lagaisse and W. Joosen. *True and transparent distributed composition of aspect-components*. In Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware '06), Michi Henning and Maarten van Steen (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 42-61. (2006).

[12] B. Meyer, *Applying "Design by Contract"*, Computer on (IEEE), v.25 n.10, pp.40-51, October, (1992).

[13] B. Schilit, N. Adams, and R. Want. *Context-Aware Computing Applications*. In Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications (WMCSA '94). IEEE Computer Society, Washington, DC, USA, pp. 85 – 90, (1994).

[14] C. Courbis and A. Finkelstein. *Towards aspect weaving applications*. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, pp.69-77, (2005).

[15] C. Escoffier, R. S. Hall, and P. Lalanda. *iPOJO: an extensible service-oriented component framework*. IEEE International Conference on Services Computing, IEEE Computer Society,July, 2007, pp. 474–48, (2007).

[16] C. Sekharaiah K., J. Ram D. *Object Schizophrenia Problem in Modeling Is-Role-of Inheritance*. In: Inheritance Workshop in 16th European Conference on Object-Oriented Programming, University of Málaga, Spain (2002).

[17] D. Suve, W. Vanderperren, and V. Jonckers, *Jasco: an aspect-oriented approach tailored for component based software development*. In Proceedings of AOSD'3. ACM Press, NY. U.S.A., pp.21-29. (2003).

[18] E. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. *Expressive scoping of distributed aspects*. In Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD '09). ACM, New York, NY, USA, pp.27-38. (2009).

[19] E. Tilevich and Y. Smaragdakis, *J-Orchestra: Enhancing Java programs with distribution capabilities*. ACM Trans. Softw. Eng. Methodol. 19, 1, Article 1, August, (2009).

[20] E. Truyen and W. Joosen. *Run-Time and Atomic Weaving of Distributed Aspects*. In Transactions on Aspect-Oriented Software Development II, pp. 147‑181. Springer, (2006).

[21] E. Truyen, N. Janssens, F. Sanen, W. Joosen. *Support for Distributed Adaptations in Aspect-Oriented Middleware*, In the 7th International Conference on Aspect-Oriented Software Development AOSD'08, (2008).

[22] F. Dantas, T. Batista, N. Cacho, and A. Garcia. *Towards Aspect-Oriented Programming for Context-Aware Systems: A Comparative Study*. First International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments – SEPCASE '07, Minneapolis, MN, USA. May (2007).

[23] F. J. S. Silva, M. Endler, and F. Kon. *A Framework for Building Adaptive Distributed Applications*. In The 2nd Workshop on Reflective and Adaptive Middleware, Rio de Janeiro, Brazil, (2003).

[24] F. Kon. *Automatic Configuration of Component-Based Distributed Systems*. PhD thesis, University of Illinois, Urbana, Illinois, (2000).

[25] F. Steimann, *Role = interface: A merger of concepts*, Journal of Object-Oriented Programming, Vol.14, No.4, pp.23–32 (2001).

[26] G. Cabri and *et. al.*, *Collaboration-Driven Role Suggestion for Agents*, Proc. of IEEE Workshop on Distributed Intelligent Systems - Collective Intelligence and Its Applications (DIS'06), Prague, Czech, June 15-16, (2006).

[27] G. Kiczales, E. Hilsdale, J. Hugunin, and *et al*. *An Overview of AspectJ*. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01), Lindskov Knudsen (Ed.). Springer-Verlag, London, UK, pp. 327-353. (2001).

[28] G. Kiczales, J. Lamping, A. Mendhekar, and *et al*. *Aspect-oriented programming*. In Proceedings of the ECOOP'97. Springer-Verlag LNCS 1241, Finland, pp. 220-242. (1997).

[29] G. Kniesel, P. Costanza, and M. Austermann, *JMangler - A Framework for Load-Time Transformation of Java Class Files*. Proceedings of IEEE Workshop on Source Code Analysis and Manipulation (SCAM), IEEE Computer Society Press. (2001).

[30] G. Valetto, G. E. Kaiser, and G. S. Kc. *A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems*. In Proceedings of the 8th European Workshop on Software Process Technology (EWSPT '01), Vincenzo Ambriola (Ed.). Springer-Verlag, London, UK, pp. 102-116. (2001).

[31] H. Zhu, *Some issues of role-based collaboration*, Canadian Conference on Electrical and Computer Engineering. IEEE CCECE 2003. vol.2, pp.687- 690, May (2003).

[32] H. Zhu. *Role mechanisms in collaborative systems*. International Journal of Production Research, Vol. 44, No. 1, pp. 181–193, January (2006).

[33] I. Ben-Shaul, *et al.*, *HADAS: a network centric framework for interoperability programming*, In Proceedings of the Second IFCIS International Conference on Cooperative Information Systems, COOPIS '97. (1997).

[34] I. Ben-Shaul, H. Gazit, O. Holder, B. Lavva: *Dynamic self adaptation in distributed systems*. In: Robertson, P., Shrobe, H.E., Laddaga, R. (eds.) IWSAS 2000. LNCS, Springer, Heidelberg, vol. 1936, pp. 134–142. (2001).

[35] I. Ben-Shaul, O. Holder, and B. Lavva. *Dynamic adaptation and deployment of distributed components in Hadas*. IEEE Transactions on Software Engineering, vol. 27, no. 9, pp. 769 – 787, September (2001).

[36] I. Mejía and M. Südholt. *Towards a robust model for distributed aspects*. In Proceedings of the 1st workshop on Modularity in Systems Software (MISS'11), Porto de Galinhas, Brazil. In ACM, New York, NY, USA, pp.19 – 22, (2011).

[37] J. Begole, M. B. Rosson, and C. A. Shaffer. *Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems*. ACM Transactions on Computer-Human Interaction (TOCHI), vol. 6, No. 2 (June 1999), pp. 95 – 132, (1999).

[38] J. Fox. *A Formal Orchestration Model for Dynamically Adaptable Services with COWS*. In the 3rd International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2011), Rome, Italy, pp. 67 – 72, September (2011).

[39] J. Fox and S. Clarke. *Exploring approaches to dynamic adaptation*. In Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction (MAI '09). ACM, New York, NY, USA, pp.19 – 24. (2009).

[40] J. Fox and S. Clarke. *A Survey of Dynamic Adaptation Technique*. The 3rd International Multi-Conference on Engineering and Technological Innovation: IMETI, Orlando, Florida, USA, (2010).

[41] J. Malenfant, M.T. Segarra, F. Andre, *Dynamic Adaptability: The MolèNE Experiment*, Reflection 2001, LNCS 2192, Springer-Verlag, (2001).

[42] J. Y. Bang and *et al.*, *CoDesign: a highly extensible collaborative software modeling framework*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - (ICSE '10), ACM, New York, NY, USA, vol. 2, pp. 243-246, (2010).

[43] K. Biyani and S. Kulkarni, *Assurance of dynamic adaptation in distributed systems*. Journal of Parallel Distributed Computing, Vol.68, pp.1097‑1112, (2008).

[44] K. Graversen, *Role Collaborations*, In Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT) at AOSD'04. (2004).

[45] K. Kakousis, N. Paspallis and G. A. Papadopoulos. *A survey of software adaptation in mobile and ubiquitous computing.* Enterprise Information Systems, vol. 4, no. 4, pp. 355 – 389, November (2010).

[46] K. Saks, *JSR 318: Enterprise JavaBeans™,v 3.1, EJB Core Contracts and Requirements*, Sun Microsystems, INC., California 95054, U.S.A (2009).

[47] L. Navarro, M. Sudholt, W Vanderperren, and et. al. *Explicitly distributed AOP using AWED*. In Proceedings of AOSD'06. ACM, NewYork, NY, USA, pp. 51-62. (2006).

[48] M. A. Sallem, F. J. da Silva. *The Adapta Framework for Building Self-Adaptive Distributed Applications*, The Third International Conference on Autonomic and Autonomous Systems, 2007. ICAS07, (2007).

[49] M. Baldauf: *A Survey on Context-Aware Systems*, Int. J. Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, (2007).

[50] M. Dahm, *Byte Code Engineering with the BCEL API*, Technical Report B-17-98, Freie Universität Berlin, April, 2003. (2003).

[51] M. Horie, S. Morita, and S. Chiba. *Distributed dynamic weaving is a crosscutting concern*. In Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11). ACM, New York, NY, USA, pp.1353-1360, (2011).

[52] M. Mezini and K Lieberherr. *Adaptive plug-and-play components for evolutionary software development*. In Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98). ACM, New York, NY, USA, pp. 97-116, October (1998).

[53] M. Pinto, L. Fuentes, J.M. Troya. *Towards an aspect-oriented framework in the design of collaborative virtual environments*. In Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS.01), (2001).

[54] M. Pinto, M. Amor, L. Fuentes, and J. M. Troya. *Collaborative virtual environment development: An Aspect-oriented approach*. In Proceedings of DDMA Workshop. Phoenix, Arizona, pp. 97-102, April (2001).

[55] M. Nishizawa, S. Chiba, and M. Tatsubori. *Remote pointcut - a language construct for distributed AOP*. In Proc. of AOSD04. ACM Press, (2004).

[56] M. Salehie and L. Tahvildari. *Self-Adaptive Software: Landscape and Research Challenges*. ACM Transactions on Autonomous and Adaptive Systems, Vol. 4, No. 2, Article 14, Publication date: May - 2009. (2009).

[57] M. Tatsubori and *et. al.*, *A Bytecode Translator for Distributed Execution of "Legacy" Java Software*, ECOOP2001, LNCS 2072, pp. 236–255, (2001).

[58] M. VanHilst and D. Notkin. *Using C++ Templates to Implement Role-Based Designs*. In Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software (ISOTAS '96), Kokichi Futatsugi and Satoshi Matsuoka (Eds.). Springer-Verlag, London, UK, pp.22-37, (1996).

[59] M. VanHilst and D. Notkin. *Using role components in implement collaboration-based designs*. In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96). ACM, New York, NY, USA, vol. 31, no. 10, pp.359-369, October (1996).

[60] M. Weiser. *Ubiquitous computing*. IEEE Computer, vol. 26, pp. 71–72, October, (1993).

[61] N. Boulila. *Group support for distributed collaborative concurrent software modeling*. In Proceedings of the 19th IEEE international conference on Automated Software Engineering (ASE '04). IEEE Computer Society, Washington, DC, USA, pp. 422-425, (2004).

[62] N. Li, J. C. Mitchell, and D. Tong. *Securing Java RMI-Based Distributed Applications*. In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04). IEEE Computer Society, Washington, DC, USA, pp.262-271 (2004).

[63] N. Loughran, N.Parlavantzas, M.Pinto, P. Sánchez, M. Webster, A. Colyer: *Survey of Aspect-Oriented Middleware*. AOSD-Europe Project Deliverable No: AOSD-Europe-ULANC-10. (2005).

[64] O. M. Group. *IDL to Java Language Mapping*, Version 1.3. OMG, Needham, MA 02494, U.S.A., (2008).

[65] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. *Composing Adaptive Software*. Computer, vol.37, Nr.7 (July 2004), pp. 56 – 64, (2004).

[66] P. McGachey, A. L. Hosking and J. Eliot B. Moss, *Class Transformations for Transparent Distribution of Java Applications*, In Journal of Object Technology, vol. 10, 2011, pages 9:1–35. (2011).

[67] R. Mondéjar, P. García-López, E. Fernández-Casado, and C. Pairot. *TaKo: Providing transparent collaboration on single-user applications*. Computer Languages, Systems and Structures, Elsevier Science Publishers, Amsterdam, The Netherlands, vol. 38, no. 1, pp. 108-121, April (2012).

[68] R. Moreira, G. Blair, and E. Carrapatoso. *FORMAware: Framework of Reflective Components for Managing Architecture Adaptation*. Lecture Notes in Computer Science, 2003, Volume 2596/2003, pp. 115 – 129, (2003).

[69] R. N. Taylor, N. Medvidovic, and P. Oreizy, *Architectural styles for runtime software adaptation*. Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009, pp. 171 – 180, September (2009).

[70] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. *JAC: A flexible solution for aspect-oriented programming in Java*. In Proceedings of Reflection'01, volume 2192 of LNCS. Springer-Verlag, Sept. (2001).

[71] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. *JAC: an aspect-oriented distributed dynamic framework*. Software Practice and Experience, 34(12):1119‑1148, (2004).

[72] R. S. Moreira, G. S. Blair, and E. Carrapatoso. *Supporting Adaptable Distributed Systems with FORMAware*. In Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04), IEEE Computer Society Washington, DC, USA. vol.7, pp. 320 – 325, March (2004).

[73] S. Herrmann. *Object teams: Improving modularity for crosscutting collaborations*. In Procs. of Net.ObjectDays. Springer, pp. 248-264. (2002).

[74] S.Herrmann. *Composable Designs with UFA*. AOSD'02 -Workshop on Aspect Oriented Modeling with UML, April 22, (2002).

[75] S. Herrmann. *Programming with Roles in ObjectTeams/Java*, American Association for Artificial Intelligence (AAIA), (2005).

[76] S. Herrmann. *Demystifying object schizophrenia*. In Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance, MASPEGHI '10, ACM. (2010).

[77] S. Herrmann, C. Hundt, and K. Mehner, *Translation Polymorphism in Object Teams,* Technical Report, Technical University Berlin. (2004/05).

[78] S. Herrmann, C. Hundt, and K. Mehner, *Mapping use case level aspects to ObjectTeams/Java*. In Workshop on Early Aspects at OOPSLA'04, (2004).

[79] S. Herrmann, C. Hundt, and M. Mosconi. *OT/J Language Definition V1.3*. Technical Universitäte Berlin. Object Teams home-page: http://www.objectteams.org, (2009).

[80] S. Liang and G. Bracha. *Dynamic class loading in the Java™ virtual machine*. In Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98). ACM, New York, NY, USA, pp. 36 – 44, (1998).

[81] S. Sadjadi, P. McKinley, B. Cheng, and R. Stirewalt. *TRAP/J: Transparent Generation of Adaptable Java Programs*. On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE Lecture Notes in Computer Science, Springer Berlin / Heidelberg (Lecture Notes in Computer Science) vol. 3291/2004, pp.1243 – 1261, (2004).

[82] S. Subotic, and J. Bishop. *Emergent Behaviour of Aspects in High Performance and Distributed Computing*. In Proceedings of SAICSIT'05, pp. 11–19, (2005).

[83] Sun Microsystems. *Java Remote Method Invocation Specification V1.5.0*. Sun Microsystems Inc., Santa Clara, California,U.S.A. (2004).

[84] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. *An aspect-oriented approach to dynamic adaptation*. In Proceedings of the first workshop on Self-healing systems (WOSS '02), David Garlan, Jeff Kramer, and Alexander Wolf (Eds.). ACM, New York, NY, USA, pp. 85-92, (2002).

Website References:

[85] AWED website, http://awed.gforge.inria.fr/doku.php?id=tutorial:home (last visited: Feb. 2012).

[86] B. Ban. *JGroups, a toolkit for reliable multicast communication*. http://www.jgroups.org/, 2002. (Last visited: Nov. 2011).

[87] DJAsCo, http://ssel.vub.ac.be/jasco/documentation_distributed.html (last visited: April, 2012).

[88] Java-Remote Method Invocation – Release Note for JDK™ (Oracle-website), http://download.oracle.com/javase/6/docs/technotes/guides/rmi/relnotes.html , (last visited: Sep. 2011).

[89] Object Management Group (OMG), CORBA v3.1 , Release Date: January-2008, Home-page: http://www.omg.org/spec/CORBA/3.1/, (last visited: Nov. 2011).

[90] Object Teams home, http://www.eclipse.org/objectteams/, (last visited: May, 2012).

[91] Oracle™ – JPLIS: Java Programming Language Instrumentation Services, web-site: http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html, (last visited: Mar. 2012).

[92] Oracle™, *Java Naming and Directory Interface (JNDI)*, http://www.oracle.com/technetwork/java/jndi/index.html (last visited: Mar. 2012).

[93] Oracle™, *Java RMI over IIOP*, http://java.sun.com/products/rmi-iiop/ (last visited: Mar. 2012).

[94] Oracle™, Remote Method Invocation (RMI) Home, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, last visited (Mar. 2012).

[95] The AspectJ™ Programming Guide, website: http://www.eclipse.org/aspectj/doc/released/progguide/index.html. (last visited: Nov. 2011)

[96] T. Lindholm, F. Yellin, *The Java™ Virtual Machine Specification,* 2nd Ed, Website: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, (last visited: Nov. 2011)