Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática
2º Semestre 2008/2009

Analysis of Support for Modularity in Object Teams based on Design Patterns

João Luís Lopes Gomes, nº 26530

Orientador: Prof. Doutor Miguel Pessoa Monteiro

22 de Outubro de 2009

Nº do aluno: 26530
Nome: João Luís Lopes Gomes

Título da dissertação: Design Pattern Implementation in Object Teams

Palavras-Chave:

- Object Teams/Java
- Padrões de Concepção
- Programação Orientada a Aspectos

Keywords:

- Object Teams/Java
- Design Patterns
- Aspect-Oriented Programming

To my grandmother São, who always supported me in everything but could not see me finish my studies.

Acknowledgements

I would like to thank my supervisor, Miguel P. Monteiro, for the support and motivation he gave me while writing this document. He's been the best (yes, I'm expressing my opinion, and with such a strong word like BEST!) supervisor I could ask for, always being available to help me and to provide an incomparable knowledge which was one of the basis for all of this work. More than a supervisor, he has been a friend, always supporting me and my work.

To my parents, grandparents, uncles and cousins for all the support they've been giving me in all the years of my life. There are no words to describe everything they've given be and what I feel for them.

To my girlfriend Vânia, por seres a pessoa mais fixe do mundo! For encouraging me to keep working until the end, and always being present when I needed it.

Last but not least, to all my friends (who will never read my dissertation), the ones who played Risk with me, the ones who put up with me, the ones who would come to meet me while I was working, even if they were not in the best conditions, the ones I've met abroad and might never see again, the ones who came with me to music gigs, then ones who played Magic with me, the ones who watched TV series with me and helped me wasting time when I should be working, the ones who waited for me every time I was late for some meeting, the ones who I've known since forever and always supported me with a beer at Charrua, and to all the others I've forgot to mention (you know I'm always forgetting stuff), to everyone I've met in these 12+5 years of school!

To my friend Paula who actually read this dissertation without any knowledge about the subject, and spotted my writing errors. Thanks!

Also, to João Pedro Martins Rogeiro for not lending me his laptop charger for 5 minutes.

Abstract

The paradigm of Aspect-Oriented Programming is currently being studied and matured. Many aspect-oriented languages have been proposed, including Object Teams for Java (OT/J). However, to date few studies were carried out to assess the contribution of the various languages available and compare their relative advantages and disadvantages. The aim of this dissertation is to contribute to fill this gap.

In the past, implementations of design patterns in Java and AspectJ were successfully used as case studies to derive conclusions on the relative advantages and disadvantages of the language under consideration. This dissertation follows this approach, with the development of a suitable collection of examples based on the well-known Gang-of-Four design patterns.

Two repositories of implementations in OT/J of the complete collection of 23 Gang-of-Four design patterns have been developed, to be used as a basis for subsequent analysis. The scenarios used for the examples are based on Java repositories by independent authors, freely available on the Web.

Based on the repositories developed, an analysis of the modularizations obtained with OT/J is presented and compared with the results obtained using Java and AspectJ.

OT/J provides direct language support for 3 of the patterns. 20 patterns yielded separate modules for the patterns, of which 10 modules proved to be reusable. Only in 1 of the patterns, no significant differences between Java and OT/J were obtained.

Resumo

O paradigma de programação orientada a aspectos ainda se encontra a ser estudado e maturado. Presentemente existem várias linguagens orientadas a aspectos, incluindo a linguagem Object Teams for Java (OT/J). No entanto, até à data, existem poucos estudos sobre a possivel contribuição das várias linguagens existentes, assim como uma comparação das suas vantagens e desvantagens relativas. O objectivo desta dissertação é de contribuir para preencher esta lacuna.

No passado, a implementação de padrões de concepção em Java e AspectJ foi utilizada com sucesso em casos de estudo para tirar conclusões sobre vantagens e desvantagens das linguagens usadas. Esta dissertação segue esta abordagem, com o desenvolvimento de um conjunto de exemplos adequado, baseado nos conhecidos padrões de concepção do Gang-of-Four.

Foram desenvolvidos em OT/J dois repositórios completos do conjunto dos 23 padrões de concepção do Gang-of-Four, que são usadas nas análises subsequentes. Os cenários usados para os exemplos são baseados em repositórios Java elaborados por autores independentes, e encontram-se disponíveis na Web.

Com base nos repositórios desenvolvidos, uma análise às modelarizações obtidas no OT/J é apresentada e comparada com os resultados obtidos em Java e AspectJ.

O OT/J oferece suporte directo da linguagem a 3 dos padrões. 20 padrões produziram módulos separados para os padrões, módulos dos quais 10 são reutilizáveis. Apenas para 1 dos padrões não foram obtidas diferenças significativas entre a versão Java e a em OT/J.

Index

Code Listing Index

List of figures

List of class diagrams

List of tables

1. Introduction

Aspect-Oriented programming (AOP) is a recent paradigm, still subject of research and maturation. Multiple programming languages following the AOP paradigm have been proposed. However, few studies have been taken aimed at comparing the existing AOP languages, in terms of strengths and limitations of its constructs and mechanisms, as well as modularity potential. This document approaches the AOP language Object Teams for Java (OT/J), providing a case study based on the implementation in OT/J of two repositories of the 23 design patterns [9], by the Gang-of-Four (GoF). It aims at further increase the knowledge on modularity capabilities of OT/J, as well as producing a comparative analysis between OT/J and the AOP language AspectJ. Moreover, since OT/J is backwards compatible with Java, some comparisons are also made with this language.

This chapter is structured in the following sections: section 1.1 introduces the motivation for this work; sections 1.2 and 1.3 present the problem this dissertation proposes to solve, as well as the proposed solution, respectively. The contributions of this essay are listed in section 1.4, followed by rest of the document outline in section 1.5.

1.1 Motivation

The emerging of the Object-Oriented programming (OOP) paradigm offered software developers the means to look at systems as groups of entities and interactions between these entities. Although this allowed developers to implement bigger and more complicated systems in an easier way, the systems developed were essentially built under a static model, thus if later modifications to the system were needed it meant several hurdles. This happens due to the difficulty in separating into modules the different concerns involved in a certain system, which means that minor changes in a system module might require several changes in unrelated modules. AOP is introduced as a paradigm to complement the OOP paradigm, providing the means to modularize crosscutting concerns, i.e., concerns which are scattered along several classes. Moreover, it aims at allowing developers to dynamically modify their system, without modifying the original system model, so it can easily grow to meet new requirements.

The introduction of AOP paradigm was responsible to the creation of several aspect-oriented programming languages [2], such as AspectJ and OT/J. AOP languages offer several advantages to the developer, mainly in terms of modularization for reusability. Despite these advantages, when compared to OOP, AOP languages still lack maturation and are subject to research.

In contrast to AspectJ, few studies have been made to assess the OT/J language mechanisms and its support for modularization of cross-cutting concerns. OT/J makes use of some mechanisms that are not present in AspectJ, such as virtual classes and family polymorphism (see section 2.2), however few studies exist to assess the advantages of using these mechanisms in a AOP language. This dissertation aims at providing a study about OT/J capabilities for modularity and reuse, as well as producing a comparative analysis between OT/J and AspectJ.

Design patterns aim at producing modular solutions for recurring programs in software construction. Their implementation provides insights on both strengths and limitations of the languages in which they are implemented. However, to the date, few complete repositories of design pattern implementations have been produced. The production of new repositories of design patterns in different aspect-oriented programming languages would open ways for several comparative studies, for instance between different languages and language features, as well as studies on which language constructs would provide better solutions for certain problems.

Case studies, based on pattern implementation, have successful in bringing insights on the relative advantages potential contributions of aspect-oriented paradigm and some programming languages [10][21]. The work behind this dissertation involves developing case studies based on the implementation of two design pattern repositories, paving the way for an analysis of OT/J support for modularity, assessing its drawbacks and advantages, as well as to produce a comparative analysis with other programming languages, for instance AspectJ and Java.

1.2 Problem description

There are not many studies focused on comparing the relative strengths and limitations of AOP programming languages, as well as their potential for modularity. Even less studies have been done with this aim focused on OT/J. One reason for the lack of publications with this intent is the inexistence of a complete repository of the GoF design patterns implemented in OT/J. Design patterns aim at producing reusable designs, thus their implementation is a good case study which provides insights of languages characteristics, specially those oriented for reusability and modularity. A complete repository of design patterns would provide enough material to produce a mature analysis to the support for modularity offered by OT/J and various comparisons with other AOP languages. Thus, the problem this dissertation aims at solving is the lack of studies about the OT/J language when compared to other AOP languages, as well as the lack of code material to conduct these studies.

1.3 Proposed solution

Design patterns are reusable solutions for recurring problems in software design. Almost all of the 23 GoF design patterns have cross-cutting structures, between the pattern itself and the involved participants in the pattern. Implementations of these patterns prove perfect to assess the modularity support of the language where the implementations are done, providing insights on the language mechanisms used to support the design pattern implementations.

For this dissertation two repositories of the GoF design patterns have been implemented in OT/J[1]. These repositories are from independent authors, Hannemann and Kiczales [11] (which has been

[1] Project eclipse/OTDT with implementations of both repositories can be found at:
http://ctp.di.fct.unl.pt/~mpm/OTJGoF.rar

completely implemented in OT/J) and James Cooper [3] (there are 5 patterns from this repository missing in the OT/J implementation, see section 3.3), and are freely available on the web. Using independent repositories, rather than creating new examples reduce the bias probability. These implementations provide enough material to assess the potential for modularity of OT/J. Moreover, one of the repositories, by HK, is available both in Java and AspectJ. This provides a first source of material to assess the relative advantages and drawbacks of the language, when compared to Java and AspectJ. Also, James Cooper implementations are systematically based on classes from the standard Java *swing* library, which permits analysing the OT/J capability of handling systems with these properties.

It has been chosen to implement two different repositories in order to guarantee that a given pattern is reusable, thus producing a better analysis to the support for modularity offered by OT/J. For this analysis, the implementations have been divided into four distinct groups of reusability:

- Direct language support, grouping patterns for which OT/J mechanism provided a solution inherent to the language;
- Reusable modularizations, for patterns that yielded a reusable module;
- Non-reusable modularizations, for patterns which did not yield a reusable module;
- Same implementation as in Java, which holds patterns that were offered no advantages with its implementation in OT/J.

These four groups allow for an assessment of pattern characteristics that make a pattern reusable.

1.4 Contributions

The major contributions of this dissertation are:

- Implementation of two repositories (from independent authors) of the 23 GoF design patterns in OT/J. These implementations serve as basis for the analysis and comparisons done in this dissertation and pave the way for future studies on OT/J.
- Analysis of the OT/J pattern implementations, focused on OT/J support for modularity and reusability, including design pattern code examples and respective class diagrams.
- Comparative analysis between results obtained with OT/J pattern implementation and the results obtained by HK with AspectJ pattern implementation [11].
- Complete chapter on the language mechanisms and constructs of OT/J, including several code examples from design patterns, which address the language constructs discussed.

1.5 Structure of this document

The rest of this dissertation is as follows: In chapter 2 Object Teams for Java is described as an Aspect-Oriented programming language and its mechanisms are discussed, providing illustrating examples when necessary. Chapter 3 provides relevant background information for the analysis in chapter 4, which analyses the pattern implementations in OT/J in terms of reusability. Moreover, it provides a comparative analysis between OT/J and Java and OT/J and AspectJ. Chapter 5 mentions related work. Chapter 6 concludes this document and tackles the work that may be done in the future with the material provided in this dissertation.

2. Object Teams for Java

This chapter is about the programming language Object Teams for Java, OT/J, which implements role-modelling [24][25][26] from OT/J programming language perspective [12][17]. The use of roles provides the means to a better separation of concerns [22], separating objects definition from their behaviour. Thus, OT/J is introduced as a programming language aiming at providing Java with the means to modularize cross-cutting concerns [17]. It is a programming language whose design is influenced by object collaboration [12]. Moreover, quantification and obliviousness (see section 2.1) make it aspect-oriented according to the definition proposed by Filman and Friedman [8].

The first sections of this chapter introduce some relevant concepts for OT/J. Section 2.1 shortly describes Quantification and Obliviousness, generic properties for a programming language to be considered Aspect-Oriented. In section 2.2, insights are provided on Virtual classes and Family Polymorphism, relevant OT/J mechanisms.

Throughout sections 2.3 to 2.10, the primary constructs and aims of OT/J are described and discussed. Several code fragments are used to illustrate these constructs, enabling a better understanding of the concepts and how to work with OT/J. Except when explicitly stated, code fragments are from an OT/J implementation of a scenario of the Observer Pattern (originally by James Cooper [3]), which uses the reusable *Observer Pattern* module found at the OT/J webpage. In section 3.5, the complete implementation code is discussed. Section 2.11 concludes this chapter.

2.1 Quantification and obliviousness

Aspect-Oriented Programming (AOP) languages provide the programmer the means to separate into different modules the various cross-cutting concerns of a system. As proposed by Filman and Friedman [8], a programming language should have the following characteristic in order to be considered Aspect Oriented: being able to make quantified statements about the behaviour of programs, keeping programs oblivious to aspect behaviour. Quantification allows for the possibility of adding new aspects to a program, while obliviousness is the property of keeping this program unaware of aspects. A given piece of code is said to be oblivious to the aspects if it does not include any references or dependencies to aspects. This means that no changes are needed in existing code in order to add new behaviour to it.

Filman and Friedman separate the types of systems which allow quantifying over them in two types: *Black-box* and *clear-box* systems. Black-box systems quantify over interfaces, wrapping around system members with aspect-specific behaviour [8][12]. Clear-box systems allow quantification over parsed structure of system members, for example, quantifying over all variables which occur in a certain condition inside a cycle [8].

Quantified assertions are actions that should be executed when some circumstances are met. In other words, are *programming statements which state that when some condition C occurs, on an existing*

program P, some action A should be performed. In order to provide the means to quantify over existing programs, some aspects of the language should be considered:

- What kind of C conditions can be specified? (Answer in section 2.6)
- How do A actions to be taken, interact with the existing code in program P and to other actions? (Answer in section 2.4)
- How are actions composed into existing code? (Answer in section 2.6)
- Over what elements of the base program can one quantify? (Answer in section 2.6)
- Is quantification possible at run-time? (Answer in section 2.6)
- How do different actions interact among each other? (Answer in section 2.4)

All these questions are answered in this chapter, in the scope of OT/J.

2.2 Background on language mechanisms

2.2.1 Virtual Classes

In traditional OO programming languages like Java, *inner classes* are classes declared inside other classes, which are predefined at compile-time. This means that, sub-classing the outer class will not allow sub-classing its inner classes. Contrary to these, and analogously to virtual methods [6] (class methods that can be redefined in subclasses), *virtual classes* are inner classes which can be overridden and polymorphically redefined in subclasses of its outer classes, and are subject to dynamic binding [19].

Virtual classes are dependent of and accessible through instances of its enclosing classes. This means that the actual type of a certain virtual class is only known at run-time, since it is dependent of the object used to access it [5]. Virtual classes are the basis for family polymorphism, which makes use of an outer class to define a *family* of collaborating inner classes. Family polymorphism is described in 2.2.2.

2.2.2 Family Polymorphism

Family Polymorphism (FP) is used in several programming languages, such as gbeta [5], CaesarJ [20] and OT/J [12].

Method dispatch is the mechanism which maps a method call to a specific code block. During compile time, compilers only have access to the static structure of programs, so they have no means to check if the run-time type of the object is the same declared in the code or if it is some subclass. In this case, the actual method cannot be statically determined and some kind of run-time binding must take place, in order to determine which code is to run.

Late binding is a feature found in all OO programming languages. Late binding assures that the exact class type of an instance or the implementation of some requested method is correctly selected at run-time. Combining inheritance with overriding and considering the use of late binding to assure correctness of the chosen method implementation, introduces a feature called *polymorphism*.

Polymorphism is the possibility to have members of different class types to be handled using a common interface. In other words, it's the possibility of several distinct classes to have different implementations for the same method. Late binding ensures that the correct method implementation or class type of the class instance calling the method is selected at run-time.

The next example will try to better explain the concept of polymorphism. Consider a class TrainingProgram, which represents some kind of training course program for employees of some company (see Figure 1). Training programs may vary depending on the employee position in the company, thus, TrainingProgram may be sub-classed, for instance by ManagerTraining and SalesmanTraining. TrainingProgram class implements a method startProgram() to start a certain course, which is implemented or inherited by any of its sub-classes. The concept of polymorphism supported by late binding allows for any TrainingProgram instance (for instance trainingInstance) to call startProgram(), assuring that the chosen implementation for startProgram() is the one associated with trainingInstance actual class type. This feature allows for all TrainingProgram instances to call startProgram() and guarantees that the correct method implementation is selected at run-time. Polymorphism enables extensibility of a given section of source code, since a given method call, in a section of code, can be bound to multiple implementations, depending on the class instance which is calling this method.

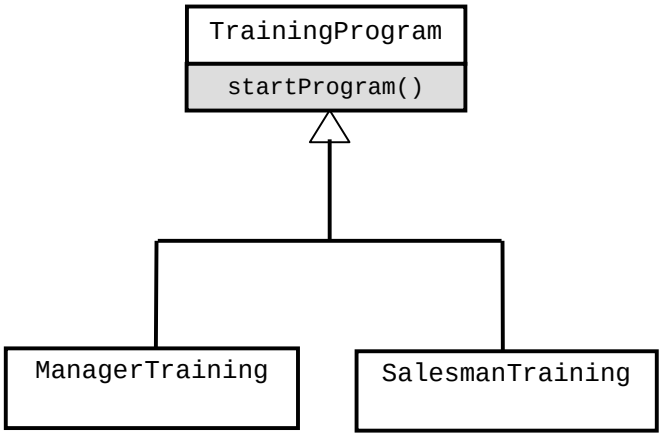


Figure 1 Hierarchy for polymorphism example

However, the client may need to capture relations between multiple objects. For example, consider the existence of a hierarchy of employees, having Employee as super-class, and Manager and Salesman as sub-classes (see Figure 2). Consider also that, in order to start a given training program for some employee, the employee attending the course must be specified. This means that method startProgram() from any trainingProgram instance, must receive as argument an instance of Employee class, becoming startProgram(Employee). Moreover, the client requires that managers receive a manager training, and respectively, the same for salesmen. For instance, startProgram method on TrainingProgram receives an Employee instance, while ManagerTraining must receive an Employee instance of type Manager. With traditional polymorphism it is not possible to guarantee the later requirement, i.e., it is not possible to capture relations between several objects and their methods (see

Figure 2). Requirements like this impose that program must not be seen as a class with some methods, but rather as a *group of relations between classes*, i.e., a collaboration between multiple classes.

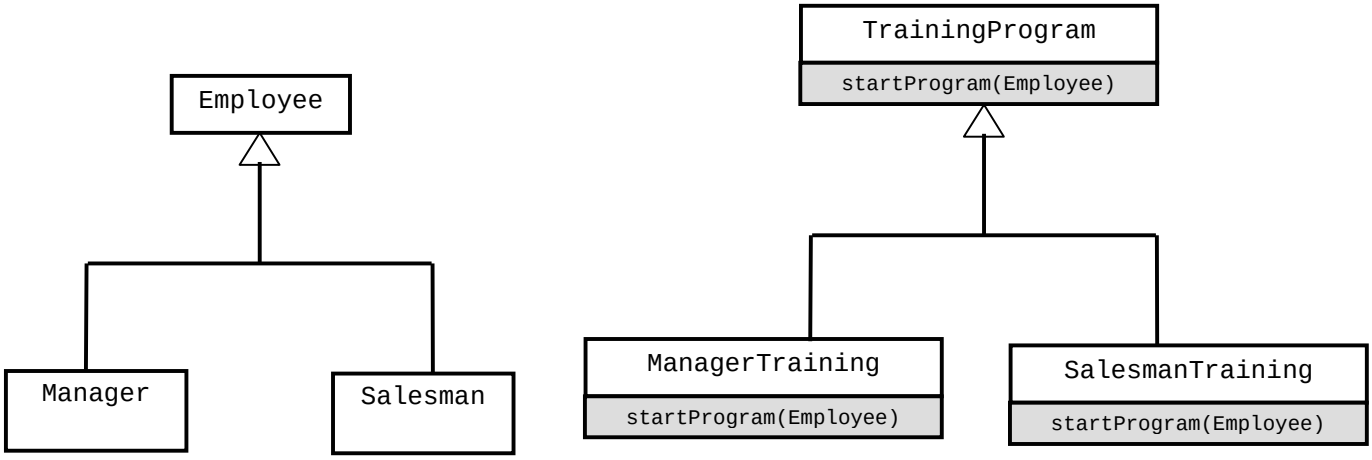


Figure 2 Relations between multiple objects captured by traditional polymorphism

Family polymorphism (FP) is introduced as a mechanism to support a multi-class perspective of polymorphism, allowing one to represent relations between several classes. Collaborating classes are encapsulated in a single object, known as *family class*, representing a family of classes, i.e., a group of distinct classes which collaborate with each other (see Figure 3). As proposed by Erik Ernst [5], one might think of collaborating classes as *attributes* of an enclosing class. Collaborating classes are inner classes uniquely owned by family class *instances*, known as *family objects*. Family objects work as a package for concrete classes. Family objects are instances of some family class variant, but it is not statically known which concrete class family it is. Classes within a family class are virtual classes, thus may be redefined. As referred in 2.2.1, the actual type of virtual classes is only known at run-time, i.e., on the basis of the family object. This enables family classes to be redefined, which means various variants of the family class may co-exist. Moreover, with virtual classes, static knowledge about all the subfamilies of some family class does not need to be propagated through the whole system. FP must statically assure that within class family refinements, collaborative class instances of different families are not mixed inappropriately at run-time, which would produce type errors in the system. Since collaborative class instances are owned by the family object, this must be passed as argument to every method which makes use of class family relations, ensuring consistency among collaborating classes. FP provides flexibility, since it is possible to create subclasses of existing family classes, and safety, since it ensures that classes from different family implementations are not mixed.

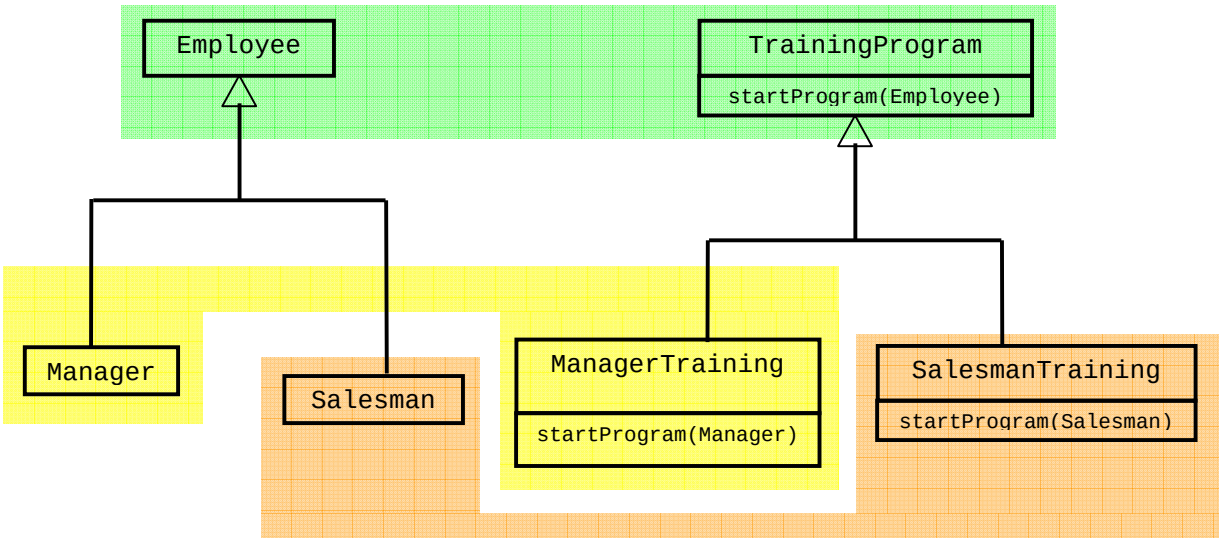


Figure 3 Relations between multiple objects captured by family polymorphism

Continuing with the TrainingProgram example, the client needed to guarantee that each TrainingProgram instance would only collaborate with correct Employee instances, i.e., startProgram method on Manager/SalesmanTraining would receive as argument a respective Manager/Salesman Employee instance (see Figure 3). FP allows for the creation of a family class that represents the relation between TrainingProgram and Employee classes. The family class may be sub-classed redefining this relation, for instance creating a family class for each relation ManagerTraining-Manager and SalesmanTraining-Salesman (see Figure 4). Each instance (family object) of these family classes, maintains consistent collaboration between instances of both TrainingProgram and Employee, making sure that startTraining in each TrainingProgram instance receives the according Employee instance as argument.

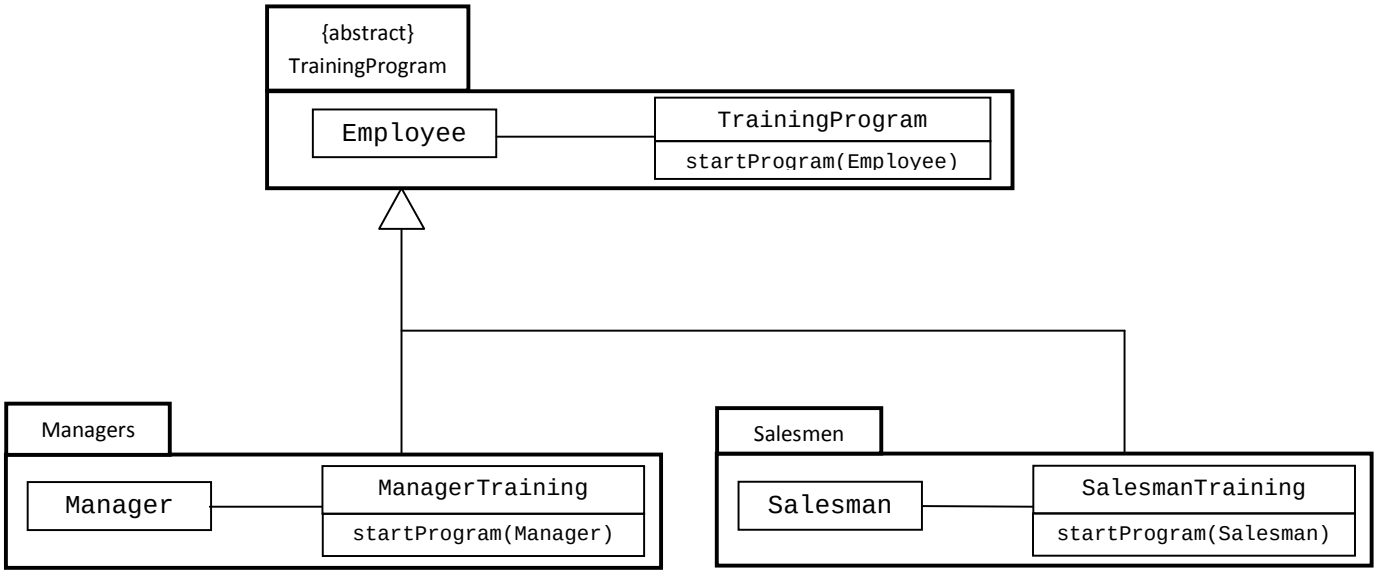


Figure 4 Family classes created by Family polymorphism

2.3 Roles in Object Teams

The term *role* is introduced exclusively in the context of OT/J. Roles work as a way of supporting different actions for the same object in different situations, a mechanism to describe the behaviour of objects within a specific context of interaction. Roles provide the possibility to focus on a particular aspect of a *base object* and to refine its behaviour according to the context on which they are used. Base objects are usually plain Java classes, whose behaviour is meant to be completely context-independent. Contexts are modelled in OT/J with the construct *team*, a new keyword introduced in OT/J (not supported by plain Java), hence the name *Object Teams*. Teams are discussed in subsection 2.4. Further discussion of role modelling is out of the scope of this dissertation.

An example of roles played by an object, are the different functions that can be played by some person, Person, through the course of its lifetime. These functions are comparable to roles, since they are refinements of the behaviour of Person. These functions might change with time (are dynamically composable), just as roles which may be *temporarily* played by some base object. Person can play the role of Student for some time in its life. When Person finishes its studies, he will find a job and start playing the role of Employee. If Person would get sacked, he would no longer play the Employee role, but will still maintain its original behaviour, i.e., will still be a person, since this is the core behaviour of the base object. Similarly in OT/J, a given role is played by a base object, within some context. In OT/J, Person would be represented as a base object, Student and Employee would be roles played by Person and its lifetime would be the context where those roles would be played. For instance, in the Observer implementation (see section 3.4) Watch2LSubject class creates a window, which is a base object for the role Subject that plays in the context of Observer pattern (see Code Listing 1).

Roles can optionally be bound to a base object. Unbound roles work as normal auxiliary classes within the context of the team module. Bound roles are related to base objects via the OT/J reserved word **playedBy**, which states that a role is played by instances of the base class. For example, in the Observer Pattern implementation (see section 3), the base object Watch2LSubject plays the role of Subject (line 2), in the context of WatchSubject:

```
01  public team class WatchSubject extends ObserverPattern{
02    protected class Subject playedBy Watch2LSubject {...}
03  }
```

Code Listing 1 Binding a role to a base class example

Bound roles are attached to the base in a non-symmetric way. Base objects are not supposed to keep track of the roles they are playing, i.e., they are *oblivious* to the roles. Otherwise, this would imply that a base object would have to grow indefinitely to know all the roles which had it as its base. This allows for flexible multiplicity, since it enables any number of roles share the same base object [17]. Base objects can play multiple roles at a given time. Also, base objects can play the same role more

than once, however, in order to maintain roles identifiable, base objects cannot play repeated roles in the same context. In the remainder sections of this report the term role always refers to bound roles.

Role classes may declare fields and methods the same way as plain Java classes. For purposes of *a-posteriori* integration into existing systems, the scopes of roles and its bases are non-overlapping, i.e., fields or methods with the same name may exist in both objects. Fields declared by roles are managed by the role instances declaring them, in order to avoid the need for base objects to grow dynamically.

The scope of a role is always context-dependent. Roles are meant to be enclosed in their own module, along with all context-specific state and behaviour (the constructs of OT/J were designed to achieve this aim [17]).

2.4 Teams as context for Roles

A context for a set of collaborations is modelled in OT/J with the use of special classes called *teams*, one of the key concepts of Object Teams [17]. Teams provide the capability of grouping roles that collaborate with each-other within a given context [12].

Teams unify properties of classes and packages. Teams, in contrast to Java classes, may be *active* or *inactive*. However, teams share some properties of plain Java classes, i.e., they have methods and fields, can be extended through inheritance and may be explicitly instantiable with the **new** keyword.

Also Teams can be approached as Java packages, since they contain sets of classes (role classes). Any inner class of a team is a role class. Teams may exist within other teams, which also makes them roles (in addition to being teams). Figure 5 illustrates the possible relations between team, role and base instances.

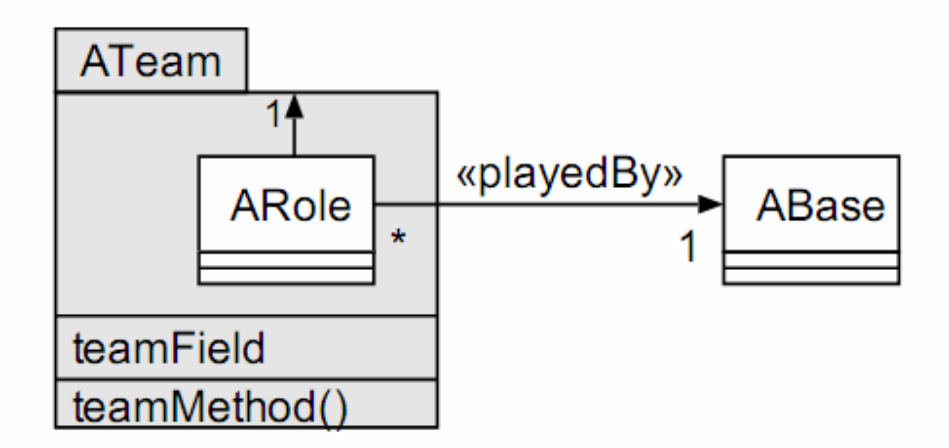


Figure 5 Basic structure of a team containing a role played by a base

As an example of team creation, a team WatchSubject, which extends the abstract team ObserverPattern (see section 3.5), is declared using the following syntax:

```
public team class WatchSubject extends ObserverPattern{...}
```

Code Listing 2 Definition of a Team example

Roles are confined to the team where they are described, imposing a strict discipline of encapsulation. Since roles are specific to a context, they are not supposed to be referred to outside a team. Also, one role is in the same team of other roles with which it interacts, giving rise to a kind of a *family* of roles. This is where OT/J makes use of the family polymorphism feature [17], [12]. Team classes are the family class, while team instances are family objects.

Given that teams work as normal classes, some form of inheritance of teams should also be provided. Teams' inner classes are virtual classes. Since a team is a class that encapsulates a set of classes, family polymorphism (aided by virtual class properties) is used to guarantee type safe subclassing of teams [12]. Teams may inherit from other teams, implicitly inheriting all the roles the super team implements. Role collaboration is supported by family polymorphism and made explicit in team methods. Family polymorphism also guarantees role instance encapsulation in team instances [13].

2.5 Translation Polymorphism

At run-time, retrieving the base instance of a role or retrieving a certain role played by a base instance (navigating between role and base instances) is possible. This navigation is done in such a way that team methods do not mention any base classes, i.e., team context is fully implemented in terms of its roles. Also, outside a team context roles are not referred to, i.e., role classes need not to be mentioned in base packages [12]. OT/J allows for base instances to be provided in cases where role instances are expected, and to return role instances where base instances are expected. The mechanisms that support these substitutions between role and base instances are described next.

Lowering

When there is need to pass a role instance outside the context of its containing team, this instance is *lowered* to, and passed outside team boundaries as its corresponding base object. Lowering a role (navigating from a role instance to its base) is a matter of following its playedBy link, since every bound role is attached to a base instance. Lowering resembles an up-cast in sub-type polymorphism, where a class instance type is "moved up" in its hierarchy, yielding a super-class. In this case the base object of the role is returned, as if it were its super-class. As up-casts, lowering is always type-safe, since if a role instance needs to leave the context of a team it is guaranteed to be bound to a base instance (except for Externalized Roles, see section 2.9), which will be returned by lowering the role. In order to allow a role class to be explicitly lowered, it must implement the interface ILowerable (line 1 of Code Listing 3), and use the lower() method to return its corresponding base instance (line 3 of Code Listing 3):

```
01  public class Role implements ILowerable playedBy Base{
02    public Base testLowering(){
03      return (Base) this.lower();
04    }
05  }
```

Code Listing 3 Explicit role lowering example

In the previous example testLowering() method (line 3) will return the base instance of type Base which plays the role Role. Again, explicit lowering is type-safe since it must be implemented by a bound role class.

Lifting

Navigating in the opposite direction, base to role instance is called *lifting*. However, since a base object may play many roles and it is oblivious of the roles it plays, this is not as trivial as lowering, from the type-checking perspective.

Lifting takes place when a role instance from some context must be retrieved from a base instance. This happens when a base instance is to enter the context of a team; it is *lifted* to the role instance it plays in this context. In comparison with sub-type polymorphism, lifting would resemble a down-cast, where a cast is performed from super-class to sub-class, in our case from base to role instance. But contrary to a down-cast, lifting is always type-safe, since at the time of the lifting translation, if such role instance does not yet exist, one will be automatically created. To ensure that a base object is always lifted to the same role instance within some team, each team instance maintains a mapping from base to role objects [12], which uses a base instance and a role type as a key to return as attribute a role instance. Lifting can be seen as a function: *teamInstance X baseInstance X roleType = roleInstance*, i.e., a given role instance is identified by its type, the team context instance where it is played and its base instance.

Lifting can be done either explicitly or implicitly (in the callin binding case). Explicit lifting implies the need to use the *as* keyword. Continuing with the Observer Pattern example (section 3), one base object may be added as an Observer of a certain Subject. The team method addObserver (line 2 of Code Listing 4) is to be used outside the team context, so it accepts base instances as arguments, which are lifted to its corresponding role instances in the team context, for instance Watch2LSubject base instance is lifted to its Subject role. Inside the method, since it is within the team context, it is now possible to treat base instances as role instances:

```
01  public <AnyBase base Observer>
02  void addObserver(AnyBase as Observer obs, Watch2LSubject as Subject sub){...}
```

Code Listing 4 Explicit base lifting example

The syntax in line 1 of Code Listing 4 asserts that AnyBase refers to any base object which plays the Observer role in the context of this team.

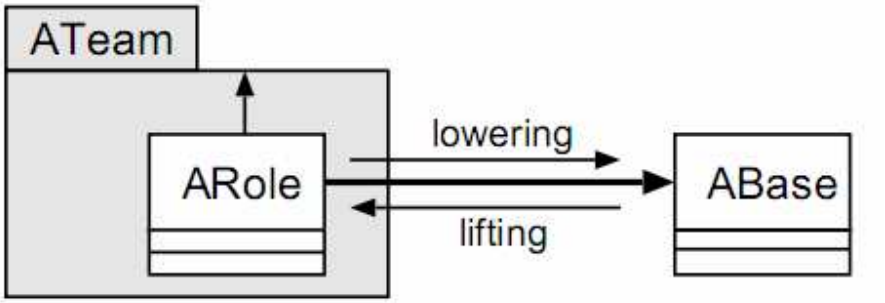


Figure 6 Possible directions of the translations between role and base

Figure 6 illustrates lifting and lowering translation, from base to role instances and vice-versa.

The rules that state what substitutions between role and base instances are allowed are called *translation polymorphism* [14], which makes use of the lifting and lowering mechanisms. The main goal of translation polymorphism is to produce a language that provides navigation from role to base and vice-versa, as easy as changing views in sub-type polymorphism. Mapping mismatching structures (bases and its roles) provides ease when integrating individually developed components. Translation polymorphism combines static and dynamic sharing, i.e., inheritance and dispatching between instances (see section 2.6), respectively. Therefore, one must remember that the classes involved in these translations, especially in the lifting translation, may be part of inheritance hierarchies. This means that care must be taken, when translating a base to a role instance; base and role classes may be related by a playedBy relation, but base and role instances may be some sub-classes of these classes. For instance, consider Figure 7. There is an abstract role AbstRole which has a sub-class role ConcreteRole inheriting from it. This abstract role is bound to an abstract base-class AbstBase sub-classed by ConcreteBase. If there is a method that needs lifting from the base to the role, one must pay attention to the dynamic type of the base instance, for instance ConcreteBase, so that lifting works out by choosing the right role type. Just looking at the static type of the object would lift to the abstract role AbstRole class, which would not work. Looking at the dynamic type is called *smart-lifting*. Smart-lifting is the mechanism that handles divergences between different inheritance hierarchies of base and role classes. It considers the dynamic type of base instance in order to dynamically return a role instance of the most appropriate role type during the lifting translation [14]. Smart-lifting provides type safety when translating from base to role instances, considering the existence of inheritance hierarchies. The algorithm used for smart lifting is out of the scope of this report.

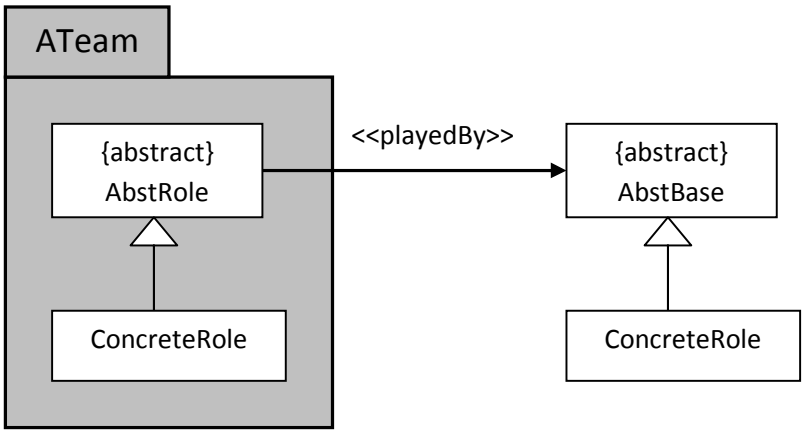


Figure 7 Role and Base hierarchies for smart-lifting example

Roles and their bases work as two sides of the same coin, i.e., they represent different guises of the same entity. But, an entity identity depends solely on the context where it is required. A role fully represents its base instance, within the context of the team, and liftings and lowerings are handled transparently by the system, avoiding the need for explicit navigation between entities. Control over instance identity is given to the client with team creation and activation (as explained in section 2.7). These mechanisms assure that object schizophrenia (see section 3.4) is not an issue in OT/J, since instance identity is handled transparently depending on the context, which can be de/activated.

2.6 Bindings between roles and bases

The binding mechanisms discussed next are the most perceptible and relevant concepts, of the OT/J programming language, for the programmer. These bindings are the constructs which allow a team to have access to members, objects and events that occur during the execution of base objects, belonging to the current context. Moreover, the existence of abstract methods in roles is possible, which are bound to base members by callout bindings.

Callout Bindings

Roles may access both state and behaviour of its base. This means that, base fields and methods might be shared with its roles, although access to these, from the role instances, must be explicitly declared. Sharing of base members is accomplished by forwarding field and method access from a role to its base. This is called a *callout binding*.

Callout bindings are accomplished by lowering the role instance to its corresponding base, i.e., following the playedBy link established between role and base. Callout bindings allow for a role to bind methods and fields from its base, i.e., the role declares (or inherits) abstract methods and fields but these are implemented by its base. The role “calls out” its base.

When calling out methods, simple forwarding to the base object is established, which means that method execution is requested by the role to the base object and there is no information that the method was requested by the role. This provides that, during the execution of called out base method,

calls to the current object (*this()*) will return the base instance. Calling out methods with simple forwarding solves the problem of broken delegation introduced in section 3.4.

Since callout bindings are to be seen from the role side, which calls out its base methods, these are represented as a role method that is "forwarding ->" to a base method:

```
01  void update(Subject s) -> void sendNotify(String selectedColor) with{
02  //mapping of arguments from Subject to String
03  }
```

Code Listing 5 Callout to a method example

In this code example, the update role method is forwarded to base changeColor method and executed by the base instance. For instance, this provides that a role method can be executed as any base method, which eases integration with existing classes.

Also, base fields may be shared by a role. To access a field from its base, a role must declare a new method which will either return the respective field or modify it. In a callout to a field, the modifiers **get** and **set**, will grant read or write access, respectively, to some field. For example, if there was need to the role access the field _color in its base instance the following syntax would be used:

```
01  private abstract Color getColorFromColorFrame();
02  getColorFromColorFrame -> get _color;

03  private abstract void setColorFromColorFrame(Color c);
04  setColorFromColorFrame -> set _color;
```

Code Listing 6 Callout to a field example

where *getColorFromColorFrame* (line 1) and *setColorFromColorFrame* (line 3) methods would respectively, get (line 2) or set (line 4), the base field _color.

When accessing its base fields, roles do not need to conform to the field access modifiers declared at the base class. This means that, even fields declared private at the base class, can be accessed and/or modified by roles played by instances of the base class. This is also true with base methods, i.e., these may be shared with roles no matter what access modifiers are defined to them.

As expected, a concrete role extending an abstract role, should implement abstract methods declared in the abstract class. Callout bindings allow roles to cover abstract methods by forwarding their execution to their base classes.

In both cases of field and method sharing, if it is not explicitly stated, base methods and fields will not be visible by the role. Thus, callout bindings provide great flexibility when sharing state and behaviour from the base object, since it allows only sharing methods and fields that are necessary to the team context.

Callin bindings

Overriding base methods within the role is also possible. Since one of OT/J's goals is ease the development of modules for *a-posteriori* integration into existing systems, methods may be explicitly overridden, i.e., methods may be overridden by methods with a *different* signature. Also, role specific behaviour can be composed into base code.

To perform overriding, the binding direction that should be followed is from the base to the role. Overriding is accomplished with the use of *callin bindings*. Technically, callin bindings are equivalent to triggers, in the way that they state "when *some condition* happens perform *this action*" (i.e., quantification as defined in section 2.1). Callin bindings involve implicit lifting of a base to its role instance, in order to look up the correct role for the actual base object.

Callin bindings, either *replace* base methods with role methods, or add role behaviour *before* or *after* base methods. Replace, before and after are the three possible modifiers for callins. The replace modifier, in a callin binding, states that some base method should be replaced (overridden) by a role method. In OO languages, in the context of regular inheritance, overriding methods can call the original method with super-calls. Similarly, in a replace method, invoking the original method is accomplished with base-calls (base.m() instead of super.m()). Replace methods must to be marked as *callin* methods, in order to use base-calls.

Callin bindings work as a way of intercepting base calls, i.e., make *quantified* statements about existing methods, which remain oblivious to the reactions/behaviour of the team. OT/J allows for black-box type quantification. The callin modifiers *after* and *before*, specify that an action should be taken after or before, respectively, of the *called-in* method. The "wrapped" methods work as a box with unknown content. This technique provides OT/J the flexibility of quantifying over class methods, even when these are already compiled and there is no means to access their code. It is also more likely to produce reusable and maintainable code [8]. Callin bindings allow great flexibility in the integration of new aspects in existing programs.

Callin bindings are represented in OT/J by:

```
role_method ← modifier base_method;
```

Code Listing 7 Callin binding syntax

The ← symbol, states that the role is instructing its base to "call into" its role. In the running Observer Pattern example (section 3.5), some callin bindings can be seen, for instance:

```
public void changeRadioButton(ItemEvent e){…}
changeRadioButton <- before itemStateChanged;
```

Code Listing 8 Callin binding example

In this code example, the role method *changeRadioButton* is executed prior to the base method *itemStateChanged*.

When calling in/out methods, method signatures may be adjusted to conform to team context specific needs, providing ease when working with *a-posteriori* integration into existing systems: Methods may be shared with a different name; base method parameters and return values may be mapped. This is done by a **with** sub-clause in the binding. Coming back to code example Code Listing 5 where a *with* clause (line 2) is used, one can see how parameters may be mapped:

```
01  String getColorFromSubject(Subject s){...}
02  void update(Subject s) -> void sendNotify(String selectedColor) with{
03    getColorFromSubject(s) -> selectedColor
04  }
```

Code Listing 9 with clause example

In this example, Subject parameter *s* is mapped to a String to conform with *selectedColor* type (line 3).

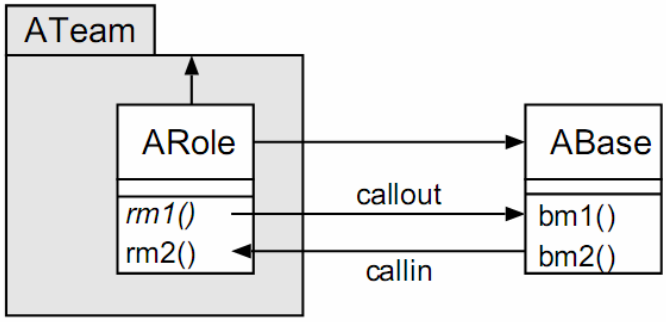


Figure 8 Representation of callin and callout bindings

Figure 8 illustrates the possible method bindings, between role and base, in OT/J.

2.7 Team activation

The context of interaction between role and base classes is made explicit in team classes. Not all contexts are supposed to be active during the complete execution of some program. If some kind of team context activation is provided, the relation between role and base instances may be established and removed at run-time [17]. As stated in section 2.4, teams may be either active or inactive.

Since callins are context-specific, in order to intercept base calls, callins must be in active teams. All callin bindings from an inactive team are disabled by default. Activating a team enables all bindings of roles within this team. Before and replace callins, from the team that has been most recently activated, are executed first, while after callins are executed last.

Teams may be activated either explicitly or implicitly [17]. The former is achieved invoking method activate() (deactivate() for the inverse function). Again, team activation can be seen in the running Observer Pattern example (section 3):

```
01  WatchSubject ws = new WatchSubject();
02  ws.activate(Team.ALL_THREADS);
```

Code Listing 10 Explicit team activation example

The Team.ALL_THREADS constant (line 2), states that ws team is active for all threads. Since in this example Swing objects are used (which run in a different thread), teams must be active for all running threads.

Implicit activation occurs when the control flow enters a team context. This happens whenever a team method is invoked. The team instance will be active until the method terminates its execution. All callin bindings of this team roles will succeed while the team method is executing. Even when a client is unaware of the existence of roles, when a base method is invoked there might exist a role in some active team which might affect the behaviour of the base object. Team activation and callin bindings assure that, role changes its base object behaviour, even if a client is oblivious about the existence of roles [12]. On the other hand, also callout bindings only occur when the enclosing team of the role is active. Team (context) activation provides great expressiveness, stating where objects should play certain role.

A team may be declared *static* if it should be active during the complete program execution. This team will permanently modify the program behaviour. No instances of this team may be created, and all its features are static features. Since the team is permanently active, all callin bindings of this team will be woven into the program, except if some guard predicate prevents the callin to be performed, thus preventing entry on the team context, maintaining the base behaviour unmodified.

2.8 Guard Predicates

Sometimes, base behaviour should only be affected by roles if some conditions are fulfilled that can only be evaluated at run-time. OT/J introduces the notion of *guard predicates* [16] as the means to control the activation of callin bindings. Guard predicates can be defined at team, role, role method and role method binding levels. Guard predicates are conditional expressions that verify whether a certain part of an aspect should be composed into base classes. Guard predicates are defined either referring to the role or to base instance.

As an example of a guard predicate, the base object (in the Observer example in section 3.5) would only play the Subject role, if the selected colour was different than red, i.e., it would only be observed if a different colour was chosen. The itemStateChanged base method would only be affected by the callin if this condition succeeded. This condition is specified at role method level, with the use of the *when* clause (line 3 of Code Listing 11):

```
01  protected class Subject playedBy Watch2LSubject{
02    public void changeRadioButton(ItemEvent e)
03    when (((JRadioButton)e.getSource()).getText() != "Red")
04    {
05      if (e.getStateChange() == ItemEvent.SELECTED){
06       selectedColor = ((JRadioButton)e.getSource()).getText();
07       this.changeOp();
08      }
09    }
10    changeRadioButton <- before itemStateChanged;
11  }
```

Code Listing 11 Guard Predicate (at role method level) example

2.9 Externalized Roles

As referred in section 2.4, role instances are not supposed to leave the context of their enclosing team, i.e., be referenced outside the team where they are created. The reason for this strict discipline of encapsulation is that roles are context dependent, and therefore roles from different teams could be mixed, giving rise to inconsistencies. However, exceptions are permitted, as it may be necessary to make roles visible outside their enclosing team in some cases. Roles used this way are called *externalized roles*.

Externalized roles must meet strict typing rules, to avoid inconsistent mixing of roles. Roles are context dependent, thus, even outside their enclosing team they must be denoted as relative to its team instance. Externalized roles use the concept of family polymorphism (see section 2.2.2), where a role is dependent of its team instance.

To maintain type consistency of an externalized role, the team instance to where it belongs must be *immutable*, i.e., marked as **final** [5]. Teams are used as *type anchors* [14], ensuring that a specific externalized role type stays dependent of the context where it belongs. Anchoring a role type to a team is achieved by declaring a variable of the desired role type, specifying its enclosing team instance:

```
final ATeam myTeam = expression;

ARoleType<@myTeam> myRole = expression;
```

Code Listing 12 Externalized role example

Since roles may be passed outside their enclosing team, they may also be passed back inside. If a role type is argument of a team method, this role type is anchored to this team. Consequently, team methods with roles as arguments must receive role instances anchored to the team itself.

A good example for the need to use externalized roles is the Adapter pattern. The aim of this pattern is to adapt a certain class interface to conform to another interface the client expects. Implementation of this pattern is relatively straightforward, since roles can implement interfaces and access their base class methods. As such, in order to adapt a given class, a role Adaptee is created and bound to the class to adapt. This role implements the interface expected by the client, and delegates its methods to the existing base class methods. In order to make this role accessible to the client, it must be externalized from its enclosing team, working as the adapted class.

For instance, consider the HK [11] example for this pattern. In Code Listing 13 the respective OT/J code for this example is shown. HK example aims at adapting the interface of *SystemOutPrinter* class, which can print strings to System.out, to conform to interface *Writer*. SystemOutPrinter uses method *printToSystemOut(String s)* to print strings, while the client expects to call the *write(String s)* method from Writer interface, for the same effect. Thus, a role Adaptee implementing interface Writer is created and bound to SystemOutPrinter (line 2), which delegates the implementation of its method to the base class (line 3). To create this role, the base class SystemOutPrinter is passed inside the team context, being lifted (line 5) to Adaptee role. This role is then externalized (line 15), allowing the client to access its methods (line 18), which conform to the interface the client expects. As referred above, the externalized role is denoted as relative to its team instance (lines 14-15).

```
01  public team class PrinterAdapterTeam{
02    public class Adaptee implements Writer playedBy SystemOutPrinter{
03      write -> printToSystemOut;
04    }
05    public Adaptee getAdaptee(SystemOutPrinter as Adaptee adaptee){
06      return adaptee;
07    }
08  }
09
10  public class Main {
11    public static void main(String[] args) {
12      private SystemOutPrinter classToAdapt = new SystemOutPrinter();
13
14      final PrinterAdapterTeam pat = new PrinterAdapterTeam();
15      Adaptee<@pat> adaptedClass = pat.getAdaptee(classToAdapt);
16
17      private Writer myTarget = adaptedClass;
18      myTarget.write("OT/J Test successful.");
19    }
20  }
```

Code Listing 13 Adapter example

2.10 Confinement

Object confinement strives for a kind of alias control where certain objects are owned by another object, which has exclusive access to its elements [15].

As referred in 2.4, and following family polymorphism rules stating ownership of inner classes by family objects, roles are owned by their enclosing team. Also, as stated in 2.9 roles can be externalized, thus it may be necessary to ensure that certain roles will be confined in its enclosing team, i.e., will not escape its scope. Role confinement should provide the enclosing team the privilege of being the only object with access to its roles features.

OT/J provides mechanisms to ensure strict role confinement and strict encapsulation of representation. The former guarantees that certain role objects are completely inaccessible from outside the context of its team. In some cases, this approach may be too extreme. Thus, the latter mechanism defines the enclosing team as the unique owner of certain roles, while allowing external objects to access their representation but not to modify them.

The mechanisms referred above are put into use either by extending class *Confined* from package org.objectteams.Team or implementing *IConfined* interface, creating *confined* and *opaque* roles, respectively. *Confined roles* guarantee that no object outside the enclosing team will ever have a reference to this role, while *opaque roles* can be passed outside a team instance while ensuring that external clients cannot access any features of this role. These mechanisms provide sufficient means to ensure different levels of role protection by teams.

Memento pattern proves to be a good example of the use of role confinement. In section 4.2 a concrete implementation of this pattern, using opaque roles, is described.

2.11 Concluding Remarks

OT/J offers the means to improved separation of concerns, distinguishing object definition from its behaviour in different contexts. Mechanisms used by OT/J which AspectJ does not have, for instance family polymorphism, seem to provide better support for class collaboration. OT/J seems promising as a means to handle pattern implementation in an efficient and modular way.

3. Background to the study

This chapter presents some background information used for this dissertation. Section 3.1 introduces the notion of design patterns and describes their aims. Section 3.2 provides a summary of the study by Hannemann and Kiczales, focused on design pattern implementation in AspectJ and Java programming languages [11]. Section 3.3 provides a brief description of the design pattern repository by James Cooper implemented in Java [3]. Object schizophrenia and broken delegation problems, are presented in section 3.4, as hurdles which appear with design pattern usage. Section 3.5 presents an implementation of the Observer pattern in OT/J, which uses an already existing reusable module.

3.1 Gang-of-Four Design Patterns

Software design, as a part of computer science, also entails problems to be solved. There are a several types of design problems that keep recurring in software design, which have already been solved, only the context where they happen is different. Recurring problems may be solved with a pattern solution which worked to solve similar problems, within different contexts. These solutions are called *design patterns*. A design pattern describes a recurring design problem, and systematically explains a solution to this problem, describing when to apply this solution and the consequences it entails [27]. Design patterns aim at producing reusable designs for systems with crosscutting concerns, helping developers to save time when designing software. Moreover, design patterns are not new and untested solutions to problems. Rather, they are proven concepts that survived over time and lots of try-outs.

The GoF is the name usually given to the group of authors, Gamma, Helm, Johnson, and Vlissides, of the widely cited book on the subject of design patterns [9]. This book introduces 23 different design patterns for common problems in software design.

One way to evaluate both strengths and limitations of the aspect-oriented programming languages is implementing design patterns and analysing the results obtained. Case studies, based on pattern implementation, have been successful in bringing insights on the relative advantages and potential contributions of aspect-oriented paradigm and some programming languages [10][20]. Moreover, since design patterns aim at producing reusable design for a system with crosscutting concerns, they are a good case study to assert the potential for modularization of the language where they are implemented.

To date, few complete repositories of design pattern implementations have been produced. The production of new repositories of design patterns in different programming languages would open ways for several comparative studies, for instance between different languages and respective language features, as well as studies on which language constructs would provide better solutions for certain problems.

3.2 The study by Hannemann and Kiczales

This section summarizes the study by Hannemann et al. (HK) on design pattern implementation in AspectJ and Java programming languages [11]. This study asserts the improvements introduced by the implementation of design patterns in AspectJ, when compared to Java, with the implementation of a complete repository of the GoF design patterns in both programming languages (freely available on the web). The study by HK provides comparison material and introduces metrics that are used in the present study.

Hannemann et al. say that benefits are mainly introduced in AspectJ pattern implementation by *inverting dependencies*, i.e., making pattern code dependent of participants rather than the opposite, maintaining dependencies contained in the pattern code. Benefits introduced are *locality*, *reusability*, *composition transparency* and *(un)pluggability*. These benefits are discussed in this section.

If a participant instance or class is free of pattern-specific code, making it oblivious about playing some role in a pattern, it can be used in different contexts without being modified. Adding it to or removing it from a pattern instance can be done by simply removing the pattern-related module from the system and performing a new build, which makes this participant *(un)pluggable* as well as *reusable*. In order to benefit from this, the participants in the pattern must have functionalities and responsibilities outside the pattern context, i.e., the roles they play must be *superimposed* [11]. For instance Observer and Subject roles in the Observer pattern are examples of such roles. Participants are not restricted to a single role or pattern instance since they have a meaning outside the pattern context. Roles played by participants that have no functionality outside the pattern are called *defining* roles.

Design pattern modularity, achieved by maintaining modules that exclusively contain pattern code, allows for better documentation, as well as, easy identification of what patterns are being used in certain system. Since all code related to a pattern is maintained in a single module, i.e., the pattern description is *localized*, the core parts of pattern implementation may be abstracted into reusable code. Pattern code becomes itself (un)pluggable, since existing participants can be promptly incorporated into a pattern and if any changes are necessary, these are to be performed in the pattern instance, not in the participants. Localizing pattern implementation makes its presence and structure more explicit, allowing for global policies to be easily imposed to patterns. Moreover, it allows for multiple instances of the same pattern, as well as, different patterns, to co-exist in an application without being confused, i.e., makes pattern instances *composition transparent*.

Hannemann et al. analyse all GoF patterns in question of the following criteria: locality, reusability, composition transparency and (un)pluggability. They distinguish between patterns with defining roles, superimposed roles and both. It is concluded that in patterns with only defining roles, no benefits are introduced by the AOP language AspectJ. On the contrary, in patterns with superimposed roles (or both), which crosscut other classes other than the pattern classes themselves, potential for

modularization exists, benefiting of all or almost all the modularity properties referred (locality, reusability, composition transparency and (un)pluggability).

For the pattern implementations in OT/J, the concepts of super-imposed and defining roles allow for an *a-priori* identification of which patterns have potential for modularization, i.e., may produce reusable modules. Moreover, concepts such locality, which permit one to achieve reusability and (un)pluggability, are important when implementing pattern examples (see section 4), as well as, to the conclusions drawn at the implementations analysis.

The present study differs from the one by HK, since besides analysing the OT/J language performance given the pattern implementations, it draws a comparison between OT/J and programming languages AspectJ and Java. The bases for this comparison are the existing design pattern repositories in both AspectJ and Java [11], as well as the implementation of the same repository in OT/J (implemented for this dissertation).

3.3 The James Cooper repository

The design pattern repository by James Cooper is presented in James Cooper book [3]. This book provides and discusses Java implementations of the 23 GoF design patterns.

This repository differs from the one by HK [11], since all 23 Java design pattern implementations by James Cooper are systematically based on classes from the standard Java *swing* library. These implementations provide material to assess OT/J capabilities to handle examples with graphical objects, proprietary classes (such as the Java *swing* library) and several threads running simultaneously.

Although the intent of James Cooper pattern implementations well serves the above mentioned purpose, the existing Java code does not follow a good coding style. For instance, in several implementations the main method and GUI functionalities of the example are all in a single class. This goes against good OOP coding style, since each single class should maintain a single scope of functionalities. For this reason, some refactoring has been done by Miguel P. Monteiro to the original Java code presented in James Cooper book[2], since good OOP coding style is essential to apply AOP to any example [28].

A few patterns from this repository (Builder, Façade, Factory Method, Interpreter and State) have not been implemented in OT/J, since their original code would need a lot more refactoring than the other examples.

[2] Refactored version available as an eclipse/JDT project at: http://ctp.di.fct.unl.pt/~mpm/PatternsJamesCooper.rar

3.4 Object schizophrenia and Broken delegation problem

Sometimes patterns introduce extra complexity to the problem to be solved. One cause for extra complexity is *object schizophrenia*, i.e., splitting what initially was supposed to be a single object [4]. For instance, several patterns encapsulate an object within another, for example the Decorator which decorates an object, by wrapping it with a new one. This creates an indirection between the wrapper and the original object. Thus two object identities must be maintained, causing extra complexity and increasing the possibility of errors to occur, for example passing the encapsulated object identity to the outside of the wrapper. This also introduces a problem called *broken delegation*. When an algorithm is part of an object and it must send a request to the object it is part of, it will call the self variable. If the algorithm is called by the wrapper object, again extra complexity is introduced, having to provide the algorithm the information it needs from the original object. Object schizophrenia and broken delegation problems are no hurdle in OT/J, as explained in sections 2.5 and 2.6, respectively.

3.5 Object Teams example

The team class presented in this example is a concrete implementation of the reusable Observer Pattern module. This example is originally taken from the James Cooper book [3], and subsequently refactored by Miguel P. Monteiro[3]. This reusable module has been found at the OT/J webpage[4], and is shown here has an example of an existing OT/J pattern example.

The Observer reusable module declares two abstract roles, Subject and Observer, to be played by base classes. The Subject role maintains a list of its Observers (line 3 from Code Listing 14), and provides methods for adding (line 4) and removing (line 9) them. Whenever there is a state change, Subject calls the changeOp() method (line 14) which notifies the Observers of this change (variant for multiple changes in one method call has been omitted). The Observer role provides interface of an update method (line 20), called by changeOp(), to be realized into a base method when there is a Subject state change. Besides declaring these two roles, this abstract team declares a method to add an Observer to a certain Subject (line 22).

```
01  public abstract team class ObserverPattern {
02    protected abstract class Subject {
03      private LinkedList<Observer> observers = new LinkedList<Observer>();
04      public void addObserver (Observer o) {
05       observers.add(o);
06      }
08      public void removeObserver (Observer o) {
09       observers.remove(o);
```

[3] These refactorings have been made with the intent of conveying more structure to the GUI related existing code, since this was initially in a flat form, i.e., all the members were at the same level in a single class.

[4] http://www.objectteams.org/

```
10      }
11      public void removeAllObservers() {
12       observers.removeAll(observers);
13      }
14      public void changeOp() {
15       for (Observer observer : observers)
16             observer.update(this);
17      }
18    }
19    protected abstract class Observer {
20      abstract void update(Subject s);
21    }
22    public void addObserver(Observer obs, Subject sub){
23      sub.addObserver(obs);
24    }
25  }
```

Code Listing 14 Observer pattern reusable mode

This concrete implementation of the Observer is an example already existing in Java, by James Cooper. For this example, using Java Swing objects, three windows are created. Watch2LSubject has three radio buttons which allow for the selection of a colour. This will be the base class for the Subject role, and a state change occurs when a new colour is chosen (line 5 from Code Listing 15).

```
01  public class Watch2LSubject extends JxFrame implements ItemListener{
02    //declaration of private fields
03    public Watch2LSubject(){...}
04    private class RadioButtonsGroup extends Box {...}
05    public void itemStateChanged(ItemEvent e) {...}
06  }
```

Code Listing 15 Class Watch2LSubject, which plays Subject role

The other two windows, WindowListFrameObserver and ColorFrameObserver will change their appearance according to the selected colour in Watch2LSubject. Thus, they play the role of Observers. Both classes have a sendNotify method (lines 4 and 9, respectively, from Code Listing 16), where update actions to state changes in the Subject, should be taken.

```
01  public class WindowListFrameObserver extends JFrame{
02    //declaration of private fields
03    public WindowListFrameObserver () {...}
04    public void sendNotify(String s) {...}
05  }
06  public class ColorFrameObserver Observer extends JFrame{
07    //declaration of private fields
08    public ColorFrameObserver Observer () {...}
09    public void sendNotify(String color) {...}
10  }
```

Code Listing 16 Classes which play Observer role

These three classes work as base classes for the roles Subject and Observer in team Watch2LSubject, which extends the reusable module class ObserverPattern. Watch2LSubject plays the Subject role (line 2 from Code Listing 17), and when a state change occurs (itemStateChanged on base class) changeRadioButton is called (line 12) which calls update on Observers (line 9). Observer role is played by ColorFrameObserver and WindowListFrameObserver classes (lines 15 and 20-21). Update method on these classes will call-out sendNotify methods on the base classes (lines 16, 22), mapping the entry parameter from Subject to String to agree with base methods signature (lines 17, 23). addObserver and remObserver methods, which respectively add and remove Observers from a certain Subject, receive as arguments base objects which play the role of Observer and Subject. These are lifted to the corresponding roles in order to work within the team (lines 27, 31).

```
01  public team class WatchSubject extends ObserverPattern{
02    protected class Subject playedBy Watch2LSubject{
03      private String selectedColor = "none";
04      public String getSelectedColor() {...}
05      public void changeRadioButton(ItemEvent e){
06       if (e.getStateChange() == ItemEvent.SELECTED){
08             selectedColor = ((JRadioButton)e.getSource()).getText();
09             this.changeOp();
10       }
11      }
12      changeRadioButton <- before itemStateChanged;
13    }
14    private String getColorFromSubject(Subject s){...}
15    protected class ColorObserver extends Observer playedBy  ColorFrameObserver{
16      void update(Subject s) -> void sendNotify(String selectedColor) with{
17       getColorFromSubject(s)->selectedColor
18      }
19    }
20    protected class ColorListObserver extends Observer playedBy
21  WindowListFrameObserver {
22      void update(Subject s) -> void sendNotify(String s) with{
23       getColorFromSubject(s)->s
24      }
25    }
26    public <AnyBase base Observer>
27  void addObserver(AnyBase as Observer obs, Watch2LSubject as Subject sub){
28      sub.addObserver(obs);
29    }
30    public <AnyBase base Observer>
31  void remObserver(AnyBase as Observer obs, Watch2LSubject as Subject sub){
32      sub.removeObserver(obs);
33    }
34  }
```

Code Listing 17 Concrete implementation of Observer reusable module

In the Main class, the team and base instances are created (lines 3, 5-7 from Code Listing 18), the team instance ws, is explicitly activated (line 4) and the base objects which play the Observer role, cframeObs and lframeObs, are added as observers to the subject (lines 8, 9), which is represented by the base object Watch2LSubject which plays the Subject role.

```
01  public class Main {
02    public static void main(String[] args) {
03      WatchSubject ws = new WatchSubject ();
04      ws.activate(Team.ALL_THREADS);
05      Watch2LSubject subject = new Watch2LSubject();
06      ColorFrameObserver cframeObs = new ColorFrameObserver ();
07      WindowListFrameObserver lframeObs = new WindowListFrameObserver ();
08      ws.addObserver(cframeObs, Watch2LSubject);
09      ws.addObserver(lframeObs, Watch2LSubject);
10    }
11  }
```

Code Listing 18 Main class of the Object Teams Observer implementation

Although there are more classes involved, this implementation is easier to comprehend than the Java version by HK [11], since it introduces a better separation of concerns. It separates the windows specific code from the roles they play, i.e., from the Observer pattern code, producing a much more readable and understandable code.

4. Analysis of the implementations

This chapter provides an analysis of the pattern implementations obtained in OT/J.

To ensure consistency, two complete Java pattern repositories from independent authors have been implemented in OT/J. Since one of the aims of the study presented in this document is assess the support for module reusability in OT/J, having two different scenarios, guarantees consistency in terms of reusability. Moreover, the choice of implementing already existing examples, rather than creating new ones, guarantees non-biasing.

The two repositories chosen to be implemented in OT/J are the following:

- The HK repository [11], since functionally equivalent implementations in AspectJ are also available. This provides a first source of material for comparisons between both AOP languages (AspectJ and OT/J), one of the aims of this document.
- The collection by James Cooper [3], because scenarios from this repository are systematically based on classes from the standard Java *swing* library. Invasive composition on such classes is not supported (see section 4.6.2), which poses additional hurdles. As a consequence, criteria for categorizing a given module as reusable (see section 4.3) are rather stringent and have an impact on the results presented in this chapter.

Patterns have been divided into a few groups, which seamed to best provide information about the capabilities of OT/J in different scenarios, and discussed in the scope of the group where they are. These groups focus on assessing language support for specific patterns and whether a successful modularization was attained; if yes, whether the modularization yielded a reusable module.

Section 4.1 provides a brief description of the format in which each group is described. Each other section (up to 4.6) represents one of these groups:

- Section 4.2 groups patterns which are directly supported by the language mechanisms of OT/J;
- Section 4.3 groups patterns for which a reusable module has been produced;
- Section 4.4 groups the ones which have not produced a reusable module;
- Section 4.5 groups the patterns for which the OT/J implementation provided no further advantageous than the Java one.

In section 4.6 drawbacks of the implementations in OT/J are discussed.

Sections 4.7 and 4.8 present a comparison between OT/J and Java and OT/J and AspectJ, respectively.

4.1 Format of the groups in this chapter

Each group in this chapter has its own section, with its title and the patterns which fit in the group. In these sections a brief description of the group is provided, specifying which type of patterns fit in it and why. The patterns that fit in a group are discussed in terms of group specific characteristics. Moreover, an implementation example of one of the patterns in this group is discussed, providing an illustrative example for the group.

Note that language support for specific patterns and issues related to reusability do not necessarily yield disjoint groups: *Memento* features in both.

These groups aim at providing the reader a good understanding of the implementation results obtained in OT/J.

4.2 Direct Language Support: Abstract Factory, Factory Method and Memento

Three patterns have been identified, whose purposes are directly supported by the language constructs of OT/J. Language mechanisms, such as family polymorphism (section 2.2.2) and confinement (section 2.10), provide the means for the implementation of these patterns to be directly supported by OT/J. Direct language support for these patterns makes them inherent to the language, (usually) allowing for a better representation of the pattern intent. These patterns are discussed in this section.

Abstract Factory and *Factory Method* are implemented through family polymorphism, which makes use of virtual classes.

Abstract Factory: supported by family polymorphism

Abstract Factory purpose is to create factories of objects with a common theme, i.e., all from the same family, prohibiting that objects from different families mix. That is the purpose of family polymorphism [5]. In order to do so, an abstract team is created, comprising a virtual class for each object factory. This abstract team represents the family class, in terms of family polymorphism. As referred in section 2.2.2, the family class acts as a capsule for its virtual classes. Therefore, this abstract team defines which classes are related to each other, preventing classes from different themes to get mix. The abstract team is to be extended by concrete teams, while family polymorphism with direct support from the type checker, guarantees that these sub-classes maintain family consistency.

Factory Method: supported by virtual classes

Factory Method's aims at emulating *polymorphic constructors*, i.e., create a super-class that defines in its constructor the creation of a certain type of object, but lets its subclasses specify the concrete objects they create. The super-class maintains an abstract role representing the object to create, which is sub-classed making the role concrete. Since the concrete roles are virtual classes, which depend on the enclosing team (the super-class) they can be polymorphically instantiated using the same

constructor call, i.e., using polymorphic constructors. The intended effect is thus directly supported by the language.

Memento: supported by confined types

The purpose of Memento is to save a *snapshot*, i.e., the current internal state, of a given object, and externalize it, ensuring that only the originating object can access the saved memento, i.e., without violating the original object encapsulation. Mementos are to be kept by a *caretaker* that must not modify the mementos it keeps. In many languages, keeping an object's internal state outside the object is hard or impossible to implement without violating encapsulation to some extent, due to limited language support for this kind of enhanced encapsulation.

In OT/J there are two mechanisms, confined and opaque roles (see section 2.10), which provide the strict protection required for implementing Memento. Confined and opaque roles prove perfect to implement Memento pattern, since this pattern aims at maintaining a Memento role without violating its encapsulation [9].

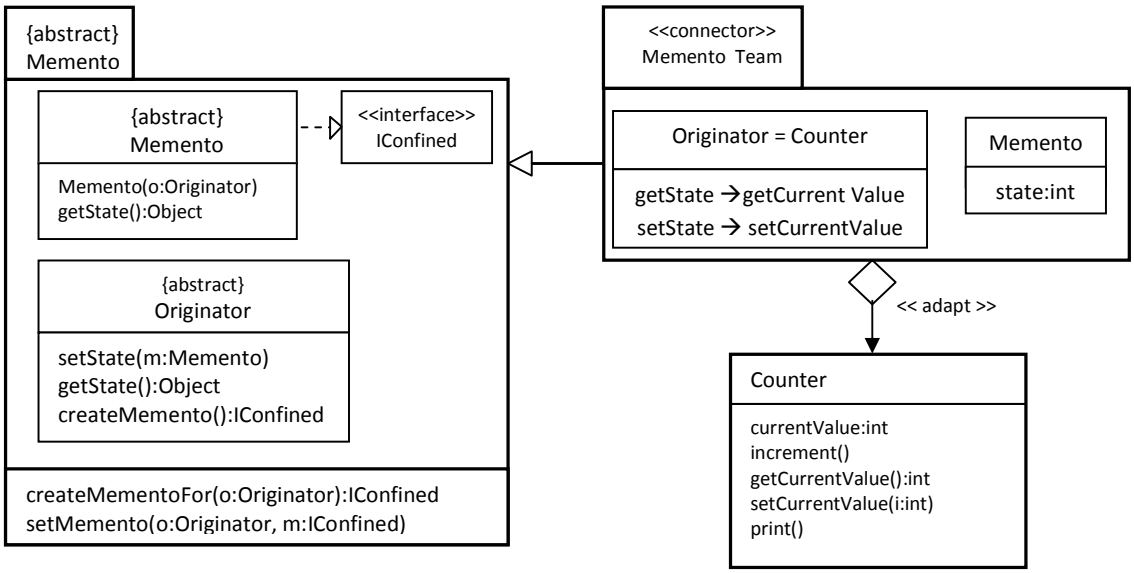
Creating a confined Memento role will guarantee that instances of this role will never be passed outside their team scope. Creating an opaque Memento role, allows this role to be passed outside its enclosing team, as an externalized role, and be referenced by any class, having the guarantee that only the memento's originator role instance has access to any of its features.

Example of Memento

Memento pattern implementation in OT/J is described below as an illustrative example of a pattern directly supported by the language.

As referred in 4.3, an abstract reusable team module was created for Memento. As the other patterns that yielded reusable team module, Memento has some common parts to all of its concrete instances. These parts are declarations of two abstract roles, an unbound Memento role and an Originator role, as well as team methods to save and set mementos, given a specific originator. Role methods to save and set mementos are to be concretized in teams implementing the abstract team.

Since the scope of this section is the language mechanisms which directly support pattern implementations, the reusable team module code will not be shown. On the other hand, an example of a Memento concrete team is considered relevant, since it shows confinement mechanisms at use. Both OT/J confinement mechanisms (confined and opaque roles, see section 2.10) have been used in distinct versions of the same Memento example, although, for this example only the version using opaque roles will be considered. The complete concrete team code is listed in Code Listing 19. Abstract and concrete teams are illustrated in Class diagram 1.



Class diagram 1 Memento implementation class diagram

Originator and Memento roles

In concrete teams, the Originator role is to be bound to a base object whose internal state we want to save (line 12 in Code Listing 19 Memento concrete team example). Also, Originator methods to get and set its state must be delegated to its base class, as shown in Class diagram 1.The Memento role keeps the internal state of the Originator and must assure that only its originator has access to this it.

Concrete Memento team

In order to allow for mementos to be kept outside the pattern context, the abstract team provides a method to pass a memento object outside its boundaries, i.e., to the participant classes, which is overridden with explicit lifting in the concrete team (line 20). As referred above, encapsulation of mementos must be preserved, therefore the Memento role is declared as implementing IConfined interface, making it an opaque role (line 2). As such, this role can passed outside its enclosing team as an IConfined object (line 21), guaranteeing that objects outside the team context will not access any of its features.

In this concrete example (see Code Listing 19 Memento concrete team example), the Originator role is played by an instance of Counter class (line 12), whose internal state is simply an integer, and which provides a method to increment this value (see Class diagram 1). At some point, the client asks for a memento of the current internal state of a Counter by calling the team method *createMementoFor* (line 20). The Counter instance is passed into this method and lifted to its corresponding Originator role, which creates a memento of its internal state and returns it in form of an IConfined object to the external client. Since this memento is an opaque role object, external clients cannot access any of its features, preserving its encapsulation.

```
01  public team class MementoTeam extends MementoProtocol{
02    public class Memento implements IConfined{
03      private int state;
04      public Memento(Originator o){
05       this.state = (Integer)o.getState();
06      }
07      public Object getState() {
08       return state;
09      }
10    }
11
12    protected class Originator playedBy Counter{
13      public abstract Object getState();
14      getState -> getCurrentValue;
15      public void setMemento(Memento m){
16       currentValue = (Integer)m.getState() ;
17      }
18    }
19
20    public IConfined createMementoFor(Counter as Originator o) {
21      return super.createMementoFor(o);
22    }
23    public void setMemento(Counter as Originator o, IConfined m) {
24      super.setMemento(o, m);
25    }
26  }
```

Code Listing 19 Memento concrete team example

4.3 Reusable modularizations: Chain of Responsibility, Command, Composite, Flyweight, Mediator, Memento, Observer, Prototype, Strategy and Visitor

Some patterns have common parts to any of its instances and (inevitably) parts that are instance-specific. As expected, in terms of reusability, the common parts are the interesting ones. These can be abstracted into reusable abstract modules, which are to be concretized by each concrete part of the pattern.

Every pattern in this section has common parts, which have been abstracted, thus producing reusable teams. Concrete instances of the pattern must extend these reusable modules, specifying which participating classes play which roles, thereby giving rise to case-specific concrete teams.

A pattern is considered to yield a reusable module, if more than abstract declarations can be obtained in common to multiple instances of a given pattern. Also, modules are only considered reusable if they can be used in implementations from both scenarios, HK [11] and James Cooper [3].

The patterns which yielded reusable modules can be separated into two groups: The ones with only super-imposed roles (Observer, Mediator, Chain of Responsibility and Prototype), and the ones which also have defining roles (Memento, Composite, Visitor, Command, Strategy and Flyweight).

In the Java implementation of the patterns with defining roles, these defining roles had to be made concrete by participating classes, whose sole functionality is to represent these roles. On the other hand, in OT/J, apart from Command and Strategy patterns, participating classes can avoid holding pattern specific code in. Either the team itself represents the defining role:

- *Component role* in *Composite*;
- *Visitor role* in *Visitor*;
- *Caretaker* in *Memento;*
- *FlyweightFactory* in *Flyweight;*

or there are unbound roles representing them:

- *Memento* in *Memento*.

The fact that all roles in these two groups of patterns (except for Command and Strategy) are either super-imposed or represented by teams and OT/J roles, allowed for the removal of pattern-specific code from participating Java classes. This is possible, since participating classes also have functionality outside the pattern context, other than inside. Therefore, pattern-related functionality can be located in the pattern module. Moreover, there is no need to have participating classes representing defining roles, which again allows for the criteria of locality to be met by the patterns from this section.

Although, Command pattern implementation produced a reusable team module, all that could be abstracted (other than mere abstract declaration of pattern roles) was the creation and maintenance of auxiliary data structures. The purpose of these data structures is to keep a mapping between Commands and its Invokers and Commands and its Receivers. Moreover, concrete instances of Command pattern have no other functionality than keeping these mappings, and specifying which events on the base classes trigger the execution of a certain Command. For this reason, and since several commands can exist in the same example, the defining role Command is represented by participant classes, allowing them to be used in several pattern instances. The same happens with the defining Strategy role in Strategy. Since each existing concrete Strategy does not depend on any pattern instance, their implementation is placed on participant classes, allowing them to be used in multiple pattern instances.

The separation of concerns between pattern behaviour and participating classes', allows for (un)pluggable pattern instances. Since pattern code is removed from participating classes, patterns can be easily composed into (and removed from) participating classes.

Example of Chain of Responsibility

Chain of Responsibility (CoR) is one of the patterns which yielded a reusable module. Its intent is to allow for a request to be passed along a chain of request handler objects – until one handler accepts the request or the end of the chain is reached – while avoiding a tight coupling between sender and receiver.

CoR prescribes the Handler role, played by all participants in the pattern. Each handler has a link to its successor in the chain and has operations to check if it should handle a specific request and if so, to handle it.

The Handler role is plainly a superimposed role, and therefore lends itself to be represented in its own OT/J module, separately from example-specific classes.

CoR is a pattern which produced a reusable module, since it has a number of parts that are common to any instance of the pattern, and example-specific parts.

The common parts to all instances of CoR are:

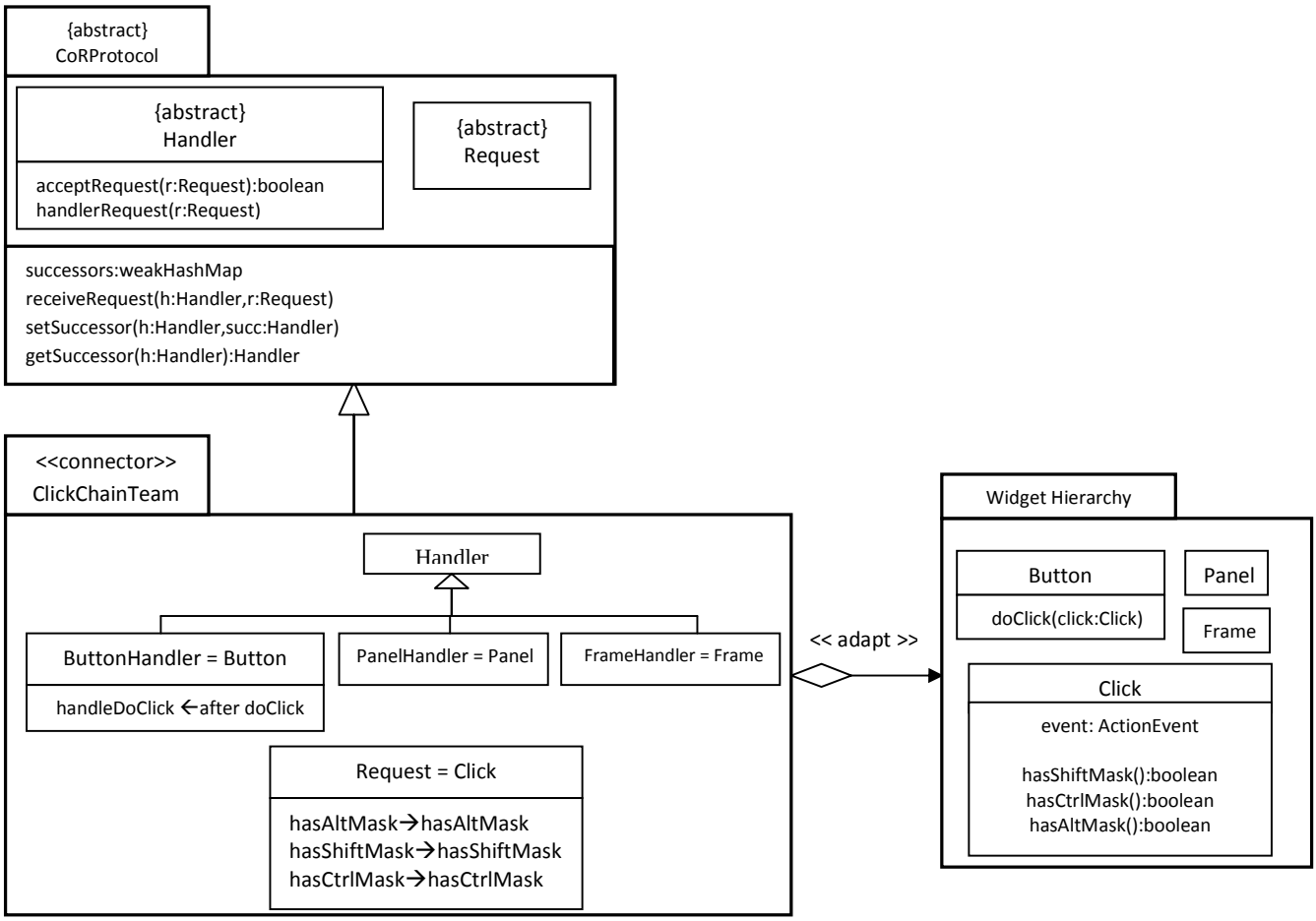
- Logic for the Handler role.
- A Handler operation to handle requests from other handlers.
- Management of the mapping between handlers and their successors.
- The logic to either handle a request or pass it along the chain.
- The Request role.

In the abstract team module, besides the Handler role, there is a Request role. The purpose of this role is to allow for requests to be treated as first class entities in the context of the pattern. This provides better control over the requests, which are to be passed from Handler to Handler.

The parts specific to each pattern instance:

- Which classes can play the role of Handler and which can play the Request role.
- The implementation of accept and handle request methods.
- What action(s) on the participant classes initiate the handling of a request.

The abstract reusable team module for this pattern, which reflects the above mentioned commonalities, is illustrated in Class diagram 2 by CoRProtocol and its code is shown in Code Listing 20.



Class diagram 2 CoR implementation class diagram

```
01  public abstract team class ChainOfResponsibilityProtocol {
02   protected abstract class Handler {
03    public boolean acceptRequest(Request request) {
04      return false;
05    }
06
07    public abstract void handleRequest(Request request);
08   }
09
10   protected abstract class Request { }
11
12   protected void receiveRequest
13    (Handler handler, Request request) {
14    if (handler.acceptRequest(request)) {
15      handler.handleRequest(request);
16    } else {
17       Handler successor = getSuccessor(handler);
18       if (successor == null) {
19        System.err.println("END OF CHAIN REACHED)\n");
20       } else { receiveRequest(successor, request); }
21   }
22
23   private WeakHashMap successors = new WeakHashMap();
24
25   public void setSuccessor (Handler handler, Handler successor) {
26    successors.put(handler, successor);
27  }
28
29   protected Handler getSuccessor(Handler handler) {
30      return ((Handler) successors.get(handler));
31   }
32  }
```

Code Listing 20 Reusable ChainOfResponsibilityProtocol team module

Handler and Request roles

Handler and Request roles are realized as abstract roles in the reusable team ChainOfResponsibilityProtocol (lines 2-8, 10 respectively, from Code Listing 20). These are used to represent the roles played by participat classes in the context of this pattern.

The abstract Handler role defines an *acceptRequest* method (lines 3-5) to confirm handling of requests, which returns false by default, and an abstract *handleRequest* method (line 7) to handle requests. Both methods are to be implemented in concrete instances of the pattern, or have their execution delegated to the classes playing the Handler role.

The Request role is declared as an empty role with no methods, since it will only be used to allow a base class to represent a request in the pattern context, which has no other intent than to be passed from Handler to Handler.

In concrete teams extending this abstract team, Handler and Request roles are to be bound to participating base classes (see Class diagram 2), adapting their behaviour according to the context of the concrete Chain of Responsibility instance.

Keeping a chain of handlers

The abstract team keeps a data structure to maintain a mapping of Handlers and their respective successors, representing a chain of handlers (see Figure 9). This data structure is a weak hash map (line 23), which keeps a Handler and its successor (if it has one) as the pair (key, value). Each concrete instance of the pattern, extending ChainOfResponsibilityProtocol, will have its own data structure to maintain the chain of handlers.

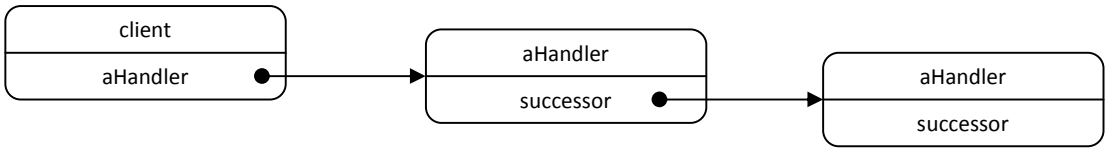


Figure 9 Chain of handlers' representation

The abstract team public method *setSuccessor* (lines 25-27 of Code Listing 20) sets a mapping from Handler to its successor. This method is to be called from outside the context of the team instance. Therefore, this method is to be overridden by concrete instances of the pattern, introducing explicit lifting in its signature, from base instance to the role it plays in the pattern (line 42 from Code Listing 21).

The protected method *getSuccessor* (lines 29-31 of Code Listing 20) is only used internally by *receiveRequest* method (lines 12-21), to retrieve the successor of a given Handler.

Handle requests and pass along the chain logic

The *receiveRequest* method (lines 12-21), implemented in the body of the abstract team module, reflects the handling/pass-along logic.

This method receives a Handler *handler* and a Request *request* as arguments (line 12), and starting on *handler* it checks whether this handler will handle the request (line 14). If it does, handleRequest method is called (line 15) on the current handler instance. If not, the request is passed to the successor of the current handler, and so on, until it is handled or the end of the chain is reached.

receiveRequest method is to be executed upon certain points of program execution. These are to be specified by concrete roles, in concrete teams extending ChainOfResponsibilityProtocol. In OT/J, these are specified by means of callin bindings, stating that when a certain point of execution of the program is reached, the receiveRequest method should execute. Although callin bindings are defined in terms of base class methods, these classes are oblivious about the execution of any action at the team level, such as executing receiveRequest.

Specific parts to each CoR instance

The specific parts of each pattern instance are implemented in concrete teams extending the abstract team module.

In each concrete team, a context specific chain of Handlers is defined. This chain can have any number of Handlers, and can possibly handle any type of requests (which was not tested, but would only require a switch clause at the beginning of the handler methods).

In concrete teams it is also defined:

- What kinds of concrete Handlers exist, which is done by subclassing the abstract Handler role.
- Which classes play the role of Handler and which represent a Request.
- How each Handler handles a request and what requests will it handle. Handling and accepting requests is done either by simply implementing the abstract methods handleRequest and acceptRequest, or by delegating their execution to participant base classes, via callout bindings.
- What triggers the execution of the "handle request or pass along the chain" logic, provided by receiveRequest method. This method is usually invoked by the Handler at the top of the chain, when certain operations are performed at base class level, for instance, after the execution of some method at the base class playing this Handler role. The execution of the base method is captured by the Handler role via a callin binding.

Code Listing 21 shows a concrete team, extending ChainOfResponsibilityProtocol.

This team implements an example based on swing graphical objects, where a click on a button (line 6 from Code Listing 21) triggers a request (line 4 from Code Listing 21) that is passed along a chain of Handlers, composed by the widget hierarchy Button (line 2) -> Panel (line 15) -> Frame (line 24).

The Request role (line 33) is played by Click class, which knows if any key was being pressed at the time the button was clicked. Depending on the pressed key a different Handler will handle the Request (lines 34-39) (see Class diagram 2).

The method *setSuccessor* is overridden in the concrete team (line 42 from Code Listing 21), using explicit lifting in its entry arguments. This allows it to be called from outside the team, where base classes are at hand, rather than role classes.

```
01  public team class ClickChainTeam extends ChainOfResponsibilityProtocol {
02    protected class ButtonHandler extends Handler playedBy Button{
03      public void handleDoClick(Click c) {
04       receiveRequest(this, new Request(c));
05      }
06      void handleDoClick(Click c) <- after void doClick(Click c);
07      public boolean acceptRequest(Request request) {
08       System.out.println("Button is asked to accept the request...");
09       return request.hasShiftMask();
10      }
11      public void handleRequest(Request request) {
12       System.out.println("Button is handling the event.\n");
13      }
14    }
15    protected class PanelHandler extends Handler playedBy Panel{
16      public boolean acceptRequest(Request request) {
17       System.out.println("Panel is asked to accept the request...");
18       return request.hasCtrlMask();
19  }
20      public void handleRequest(Request request) {
21       System.out.println("Panel is handling the event.\n");
22      }
23    }
24    protected class FrameHandler extends Handler playedBy Frame{
25      public void handleRequest(Request request) {
26       System.out.println("Frame is handling the event.\n");
27      }
28      public boolean acceptRequest(Request request) {
29       System.out.println("Frame is asked to accept the request...");
30       return request.hasAltMask();
31      }
32    }
33    protected class Request implements ILowerable playedBy Click {
34      public abstract boolean hasAltMask();
35      hasAltMask -> hasAltMask;
36      public abstract boolean hasCtrlMask();
37      hasCtrlMask -> hasCtrlMask;
38      public abstract boolean hasShiftMask();
39      hasShiftMask -> hasShiftMask;
40    }
41    public <AnyBase base Handler, AnyBase1 base Handler>
42  void setSucessor(AnyBase as Handler handler, AnyBase1 as Handler successor){
43      super.setSuccessor(handler, successor);
44    }
45  }
```

Code Listing 21 CoR concrete team example

4.4 Non-reusable modularizations: Adapter, Bridge, Builder, Decorator, Façade, Interpreter, Iterator, Proxy, State and Template Method

In the patterns from this group, the OT/J implementations were successfully modularized but could not be used in more than one scenario, i.e., did not yield a reusable module. In some cases, all that could be achieved would be the abstract declaration of operations. The absence of the possibility to create a reusable module for these patterns is due to the nature of the pattern instances being very scenario-specific.

Adapter, Bridge, Decorator, Proxy and State

The purpose of *Adapter*, *Bridge*, *Decorator* and *Proxy* is to adapt a given class (which is concretized in OT/J by callin bindings). A reusable module for these patterns was not produced since adaptations are non-reusable by their very nature, as associated code serves to adapt case-specific classes (i.e., *glue code*). However, some language constructs from OT/J have been used to concretize these patterns implementation, such as Guard Predicates (Decorator) and Externalized Roles (Adapter).

State keeps track of the current state of a given base class. This state can be related to any member of the base class (callin bindings, in the style of AspectJ around advice, are used to determine when and what state to maintain), which is again a case-specific decision. Since both state and how to keep track of it are case-specific (depending on what the client needs) this pattern cannot be abstracted into a reusable module.

Template Method and Builder

The intent of *Template Method* is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method can also be viewed as a technique to prevent duplication through the use of traditional inheritance, by factoring out common code to a (possibly abstract) super-class and leaving case-specific code in the subclasses. OT/J implementation of *Template Method* uses a module composition mechanism *other than inheritance*: the *playedBy* binding between role and base class (see section 2.3). This relation has similarities to the relation between a (possibly abstract) super-class and a concrete subclass, to the extent that a role can defer to the base the definition of the operations it declares. The OT/J version of the HK *Template Method* illustrates this: the team has a role standing for the abstract class and an example-specific class that stands for the concrete sub-class. This approach could not be used on the example from the Cooper collection because the abstract class has a single constructor declaring several arguments. Using this approach on this example would be tricky, because it would entail providing the role with a constructor with parameters and explicitly instantiate it, instead of usual practice of instantiating the base and let the corresponding role instance to be implicitly created.

The intent of Builder is to separate the construction of a complex object from its representation so that the same construction process can create different representations [9]. Structurally, the HK Builder is another instance of Template Method. The OT/J version is similar, but is structured in such

a way that the representation is a virtual class (i.e., an unbound role), with the consequence that code for the actual instantiation of different representations become variation points, as with Factory Method (see section 4.2).

Façade, Interpreter and Iterator

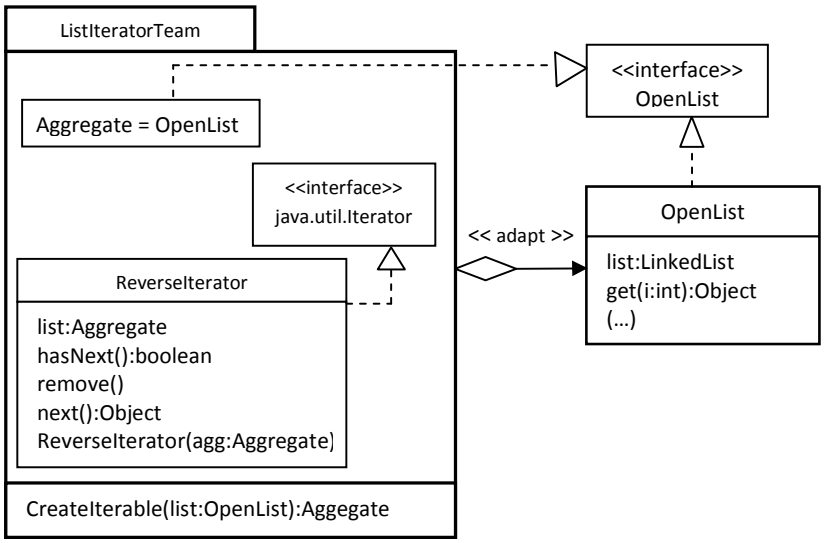
Iterator provides the means to iterate over elements of an aggregation of objects. In the analysis by HK [11] to their AspectJ implementations, it is stated that for the Iterator implementation, a reusable module was produced. However, in the OT/J implementation, no reusable module was derived for this pattern, thus it is considered that Iterator does not respect the Reusability criteria. This assumption is due to consider that just extending the Iterator Java interface in pattern instances does not count as reusability, since this reusability comes from the Java platform rather than from OT/J. What could be abstracted was mere declaration of operations of a common Iterator (done in Java's Iterator interface). The primary advantage brought by OT/J to these examples is the ability to package together the *Aggregate* and *Iterator* pattern roles into a common team module. This packaging capability also marks the difference between the OT/J implementations of *Interpreter* and *Façade* and the corresponding implementations in Java and AspectJ – note that *Façade* is the sole pattern from the HK study in which the Java and AspectJ implementations are identical.

Iterator example

Iterator makes use of family polymorphism and externalized roles in its concrete implementations. Thus, it is a good example of a pattern for which a reusable module was not produced, however, its OT/J implementation allowed for a separation of concerns, between participant classes and pattern code.

A team module was produced, reflecting the participants in this pattern. This team declares an Aggregate role, which is bound to the base class to iterate and any number of role classes implementing Java Iterator class, representing concrete iterators which can iterate over Aggregate instances of this team. Family polymorphism guarantees that Aggregate instances and concrete Iterators from different team instances do not mix, ensuring type consistency among class collaboration.

Having the Aggregate role bound to the base class to iterate in the same team as the Iterator role which iterates it, provides pattern locality, i.e., all pattern related code is located in the team module. This avoids the need to have Iterator related code scattered through base classes.



Class diagram 3 Iterator implementation class diagram

In this specific scenario, there is an OpenList class instance, implementing SimpleList interface, over which the client wants to iterate. This list plays the role of Aggregate, providing the role access to its methods via callouts, avoiding need to have pattern code in OpenList class (see Class diagram 3). The iteration in question is supposed to be from the end to the start of the list, i.e., performed by a ReverseIterator (line 7). This unbound role implements Java Iterator class, and it is instantiated with an instance of the Aggregate over which it will iterate (line 10).

Both Aggregate and ReverseIterator are externalized to the client (lines 30-31). Family polymorphism guarantees that only an Aggregate object anchored to the same team as the Iterator will collaborate. Moreover, both Aggregate and Iterator can be used for other tasks than the one presented in the example (line 34).

```
01  public team class ListIttTeam {
02    public Aggregate createIterable(OpenList as Aggregate list){
03      return list;
04    }
05    public class Aggregate implements SimpleList playedBy OpenList{ }
06
07    public class ReverseIterator implements Iterator{
08      protected int current;
09      protected Aggregate list;
10      public ReverseIterator(Aggregate list) {
11       super();
12       this.list = list;
13       current = list.count();
14      }
15      public boolean hasNext() {...}
16      public void remove() {...}
17      public Object next() {...}
18    }
19  }
20
21  public class Main{
22    private static void print(Iterator iter) {
23      while(iter.hasNext()) {
24       System.out.println(iter.next());
25      }
26    }
27    public static void main(String[] args) {
28      final ListIttTeam listItt = new ListIttTeam();
29      OpenList openList = new OpenList();
30      Aggregate<@listItt> iterable = listItt.createIterable(openList);
31      Iterator iter = new ReverseIterator<@listItt>(iterable);
32      print(iter);
33    }
34  }
```

Code Listing 22 Iterator pattern example

4.5 Same implementation as in Java: Singleton

Singleton is the sole pattern whose implementation resulted identical to that in Java. The intent of Singleton is to ensure a class only has one instance, and provide a global point of access to it [9]. The standard way to implement Singleton in Java is to block access to constructors through non-public visibility and to provide an accessor method that always returns the same class instance whenever it is called.

The AspectJ implementation of Singleton uses advice to intercept calls to the constructor and always return the same class instance, kept in the aspect. However, OT/J does not provide a means to

intercept class constructors. For this reason, OT/J does not bring any benefits for Singleton, compared to Java.

4.6 Limitations detected

This section provides insights on some language drawbacks, which have been noticed while producing design patterns implementations.

4.6.1 Binding class constructors

Although OT/J supports composing context-dependent code into already existing base classes, it does not support composing additional (role) behaviour into base class constructors. This is due to the fact that the base class instance is not yet created at the time the constructor is called, therefore neither the corresponding role instance is. If composing code into class constructors were possible, it would allow roles to instantiate its base classes, providing direct language support for the Singleton pattern.

In order to overcome this shortcoming, one might program the class constructor in an auxiliary method that is called in the constructor itself. Contrary to the constructor, code may be composed into this auxiliary method, thus allowing roles to access it, providing a solution for this limitation.

4.6.2 Invasive composition of Java proprietary binaries

Invasive composition of Java proprietary binaries is constrained by to legal issues, whereas by composing context code into these classes the programmer is changing them, which is not legal in cases where the resulting binaries are to be shipped to clients. Therefore, composing code into Java proprietary classes is not supported in OT/J.

This issue introduces hurdles when trying to compose context-dependent functionalities into existing examples which use some of the aforementioned classes.

For instance, all the pattern examples by James Cooper are based on classes from the Java Swing library, as referred in section 4. While implementing James Cooper examples in OTJ language, when it was necessary to compose code into swing classes, there was need for some workaround. Often, the hurdles were dealt with by adapting the example, through the use of sub-classes of the swing classes.

4.6.3 Roles playedBy interfaces

A feature in OTJ yet to be implemented is the possibility of roles being playedBy interfaces. If a role could be playedBy an interface, any participant class implementing the chosen interface could play this role.

Bounding roles to interfaces would allow roles to reference any methods of the interface, without actually reference their concrete implementation. It would provide the programmer ease to modify concrete participant classes (always respecting the interface), with no need to change anything in team modules.

For instance, in the Iterator implementation example (see Code Listing 22), the Aggregate role could be bound to a list interface, allowing this role to be played by any concrete instance of this interface. Since this is not possible, the role must be bound to a concrete class, reducing variability.

4.7 Comparison between OT/J and Java

OT/J offers all the support and mechanisms provided by Java, since it is backwards compatible with Java. However, OT/J provides more options to the user.

Team classes and inner role classes, in conjunction with the concept of family polymorphism, provides a type safe way to maintain families of collaborating classes. To implement this kind of class collaboration in Java, the programmer needs to create a lot of hand-made conditions, or resort to some design patterns, such as Abstract Factory.

The usage of virtual classes and family polymorphism make OT/J a more polymorphic and reusable language, when compared to Java, since it is possible to polymorphically refine both outer (teams) and inner (roles) classes. For instance, with OT/J it is possible to declare a set of collaborating classes, via a team and its roles, and use this team in different scenarios, simply by sub-classing the existing roles, in sub classes of the team. In Java, in order to declare a set of classes which collaborate with each other, a new set of collaborating classes would have to be created for each refinement of any of the classes in the set. Moreover, as consequence of supporting family polymorphism, OT/J supports polymorphic constructors for its roles, i.e., in team refinement, not only role methods and fields but also constructors are implicitly inherited (see section 2.4). Language features like family polymorphism and polymorphic constructors provides the means for OT/J to offer direct language support to Abstract Factory and Factory Method, as stated in section 4.2.

With the use of teams to define contexts of interaction, and roles which can access all members of base classes, OT/J introduces a separation of concerns that Java lacks. In OT/J, it is possible to isolate context-independent code in base classes (code that is similar in all base class instances and does not depend on a specific context), while keeping all context-dependent code in roles and their respective teams. Moreover, context-dependent code may be composed into existing classes, while keeping these classes oblivious of this composition. This means that improved modularity is introduced, by keeping code from different concerns in distinct modules, and allowing new code to be easily composed into existing classes. The concept of role modelling and contexts of collaboration provides the means to compose distinct hierarchy structures, allowing the possibility of interaction/integration of systems which have not been built to work together. For instance, with OT/J it is possible to produce classes for a system without the need to regard any future requirements, since future requirements can be produced in a separated module and later integrated into existing classes without changing their code, adapting their behaviour to what is needed. In Java, adapting a system or even class behaviour would mean modifying its code, which is usually a painful process. As an illustrating example, consider the Adapter (see section 2.9 for a code example), which aims at adapting the interface of a given class to one the client expect. In OT/J this is done straightforward by creating a

role, to be externalized, which implements the expected interface and delegates its methods to the class to adapt, always maintaining this class oblivious of the existence of the pattern. This way, the programmer only has to maintain the role, which respects the expected interface. On the other hand, the solution in Java involves creating a class implementing the expected interface, which encapsulates the class to adapt and calls its methods, making the programmer have to manage two distinct objects.

OT/J also provides better means for object confinement (see section 2.10), allowing the programmer to specify that certain classes should not be accessible outside a certain context. Again, in Java this is would be a painful job to implement, since Java does not provide means for the creation of contexts, and the strongest form of encapsulation provided is declaring a class and its fields as private, which either way lets this class be instantiated anywhere.

4.8 Comparison between OT/J and AspectJ

In contrast to AspectJ, OT/J aspect team modules are always, and by default, entities accessible through an object reference. This difference in handling aspects instances gives OT/J some additional flexibility with respect to AspectJ:

- Team instance creation is controlled by the programmer, allowing for a more precise control over which aspects are executing. As referred in 2.7, team composition into participant classes may be “turned on/off” at run-time. In design pattern implementations, if patterns meet the locality criteria, i.e., if all pattern related code is placed enclosed within the team, pattern instances are easily (un)pluggable from base classes, merely by (de)activating them. For instance, in Memento, this would let the programmer decide when to have the Memento team instance saving mementos, without changing the team code. By contrast, in AspectJ advice cannot be activated or deactivated dynamically, which has an impact on the implementation of some patterns – *Decorator* is a good example, since in AspectJ it is not possible to decide at run-time whether or not to decorate a certain object.

- Team instances are straightforward to reference, allowing the programmer to build his own systems based on team instances. Several instances of the same team can be created and easily distinguished by their unique instance name. For instance, in the Observer pattern, this would enable the use of different instances for each observing relationship.

Moreover, the use of roles in OT/J allows for a more fine-grained control over participant class instances. It is possible to specify if all instances, or only specific ones, of a certain participant class play some role. Roles are mapped to participant instances via team methods, without the need to keep any extra data structure, since teams keep a mapping from roles to participant instances (see section 2.5). In contrast with this, AspectJ offers no way to assert which participant instances play which role in the pattern, without keeping an extra data structure for this mapping. This means that, either a data structure is kept in the pattern module to know which participant instances play which roles, or the pattern is applied to all participant instances. For some cases this may not be the desirable effect.

Consider for instance the Adapter example (see section 2.9). While in the OT/J version it is specified which SystemOutPrinter instance is adapted (line of 15 from Code Listing 13), in the AspectJ version by HK [11] (not shown in this document), all instances of SystemOutPrinter are adapted, since there is no way to specify (without the use of an extra data structure) which instances should be adapted.

4.8.1 Comparison based on Locality, Reusability, Composition Transparency and (Un)pluggability

Table 1 was built having as basis the table presented in [11] by HK. It covers all 23 GoF patterns, having each pattern analysed in terms of the criteria used by HK (easing the comparison between results obtained in OT/J and AspectJ): *Locality*, *Reusability*, *Composition Transparency* and *(Un)pluggability* (see section 3 for further detail on these criteria). Moreover, classification of pattern roles into defining and superimposed is also included in the table (when the distinction of the two kinds of role is not totally clear, role names are shown in parentheses in either or both categories). Table 1 presents the results obtained from the OT/J implementations (identified in the "Implementation" column by the label "OT/J") and provides the results obtained by HK in AspectJ (identified in the "Implementation" column by "AspectJ"). This correspondence paves the way to the comparison between OT/J and AspectJ.

Since there are qualifications to be pointed out for both languages, a few clarifications are provided next. In the results obtained from the OT/J implementations, a few "no" entries relative to properties *reusability* and *composition transparency* are qualified with an asterisk. This is merely to indicate that the reason why the pattern instance does not enjoy the given property is due to the nature of the pattern and not attributable to limitations of the language. Also, in the AspectJ results, HK classify some pattern properties with "(yes)" instead of a plain "yes" to indicate that limitations of some sort apply [11]. In general, these are caused by the presence of defining pattern roles. For instance, take the *locality* property: though AspectJ successfully enables the separation of superimposed roles, defining roles remain in multiple classes (e.g. *State* classes for the *State* pattern).

The above limitation is not felt in the OT/J implementations, since teams make it possible to group multiple pattern components into a single cohesive scope. This capability, granted by family polymorphism, has a wide-ranging impact on the OT/J implementations. Family polymorphism allows one to enclose pattern roles (from patterns with more than one significant role) within a single team, providing enhanced cohesion. For this reason, most pattern implementations in OT/J theoretically consist of a team with (possible abstract) roles representing pattern roles. Thus, all OT/J pattern implementations respect the *locality* property, since it always seems possible to package and encapsulate pattern participants into a larger module.

Pattern	Kinds of roles		Implementation	Locality	Reusability	Composition Transparency	Unpluggability
	Defining	Superimposed					
Abstract Factory	Factory, Product	–	OT/J	yes	no	yes	no*
			AspectJ	no	no	no	no
Adapter	Target, Adapter	Adaptee	OT/J	yes	no	yes	yes
			AspectJ	yes	no	yes	yes
Bridge	Abstraction, Implementor	–	OT/J	yes	no	yes	yes
			AspectJ	no	no	no	no
Builder	Builder, (Director)	–	OT/J	yes	no	yes	no*
			AspectJ	no	no	no	no
Chain of responsibility	–	Handler	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Command	Command	Commanding, Receiver	OT/J	yes	yes	no*	yes
			AspectJ	(yes)	yes	yes	yes
Composite	(Component)	(Composite, Leaf)	OT/J	yes	yes	yes	(yes)
			AspectJ	yes	yes	yes	(yes)
Decorator	Component, Decorator	ConcreteComponent	OT/J	yes	no	yes	yes
			AspectJ	yes	no	yes	yes
Façade	Façade	–	OT/J	yes	no	yes	yes
			AspectJ	Same implementation for Java and AspectJ			
Factory Method	Product, Creator	–	OT/J	yes	no	yes	no*
			AspectJ	no	no	no	no
Flyweight	FlyweightFactory	Flyweight	OT/J	yes	yes	no*	yes
			AspectJ	yes	yes	yes	yes
Interpreter	Memento	Originator	OT/J	yes	no	yes	no*
			AspectJ	no	no	n/a	no
Iterator	(Iterator)	Aggregate	OT/J	yes	no	yes	yes
			AspectJ	yes	yes	yes	yes
Mediator	–	(Mediator), Colleague	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Memento	Memento	Originator	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Observer	–	Subject, Observer	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Prototype	–	Prototype	OT/J	yes	yes	no*	yes
			AspectJ	yes	yes	(yes)	yes
Proxy	(Proxy)	(Proxy)	OT/J	yes	no	yes	yes
			AspectJ	(yes)	no	(yes)	(yes)
Singleton	–	Singleton	OT/J	Same implementation for Java and OT/J			
			AspectJ	yes	yes	n/a	yes
State	State	Context	OT/J	yes	no	yes	yes
			AspectJ	(yes)	no	n/a	(yes)
Strategy	Strategy	Context	OT/J	yes	yes	no*	yes
			AspectJ	yes	yes	yes	yes
Template Method	(AbstractClass), (ConcreteClass)	(AbstractClass), (ConcreteClass)	OT/J	(yes)	no	(yes)	(yes)
			AspectJ	(yes)	no	no	(yes)
Visitor	Visitor	Element	OT/J	yes	yes	yes	yes
			AspectJ	(yes)	yes	yes	(yes)

Table 1 Result comparison between implementations by OT/J and AspectJ

For the reusability criteria column in Table 1, results obtained in OT/J largely match the ones by HK. All the patterns that produced a reusable module in the AspectJ implementations also produced reusable modules in OT/J, except for Singleton (for which the Java and OT/J implementations are the same) and Iterator (see section 4.4).

A column with the criteria of extensibility for each implementation is not included in Table 1, since values are always “no” for AspectJ and “yes” for OT/J. *Interpreter* provides a good illustrating example. Interpreter implementation in OT/J is organized as a super-team and a sub-team, illustrating a different benefit brought by family polymorphism: OT/J modules can always be extended through team inheritance, and team instances (enclosing role objects) can be used polymorphically (see section 2.4). The same is not possible in AspectJ examples because concrete aspect modules cannot be further extended.

4.9 Analysis conclusions

In the HK study results are grouped differently than in this document, reflecting differences in language mechanisms. To facilitate comparisons, Table 2 presents the aggregate results of both studies (the present one and HK studies), organized into groups according to the criteria presented in this chapter. There is also a group called “Pattern not modularized” in Table 2. This group contains, for each programming language (OT/J and AspectJ), the patterns that did not yield a successful modularization.

	AspectJ		**Object Teams for Java**	
criterion	**Nr. of cases**	**Patterns in the group**	**Nr. of cases**	**Patterns in the group**
Direct language support	4	Adapter, Decorator, Strategy, Visitor, Proxy	3	Abstract factory, Factory method, Memento
Reusable modularization	12	Chain of responsibility, Command, Composite, Flyweight, Iterator, Mediator, Memento, Observer, Prototype, Singleton, Strategy, Visitor	10	Chain of Responsibility, Command, Composite, Flyweight, Mediator, Memento, Observer, Prototype, Strategy and Visitor
Non-reusable modularization	3	Proxy, State, Template method	10	Adapter, Bridge, Builder, Decorator, Façade, Interpreter, Iterator, Proxy, State and Template Method
Pattern not modularized	5	Abstract factory, Bridge, Builder, Factory method, Interpreter	0	—
Implementation identical to Java	1	Façade	1	Singleton

Table 2 Comparison of aggregate results in terms of modularization, reusability and language support

Five of the AspectJ examples do not have the locality property (see Table 1), which is a minimum requisite for deeming a modularization successful. For this reason, the patterns in those examples are placed in the group “Pattern not modularized”. In OT/J, all implementations provided a successful modularization, since they all respect the locality criteria, i.e., the pattern code is always localized in team modules.

An overall look at the analysis carried in this chapter shows OT/J has a clear advantage in terms of extensibility and, in general, of what can be done with the resulting modules. AspectJ aspect modules

are generally not extensible, while OT/J team modules seem to be always extensible, the only observed limitations being due to the specifics of a given pattern.

5. Related Work

This chapter provides a short survey of insights acquired from analyses of design pattern implementation in different aspect-oriented programming languages, namely AspectJ [18], Eos [23] and CaesarJ [20].

Hannemann et al. [11] developed a study which involved producing a repository of the GoF patterns implemented in an AOP language. Several studies have been performed on this repository. One of these studies is aimed to produce metrics for comparative results between OOP and AOP (using Java and AspectJ respectively), by Garcia et al. [10].

After the GoF patterns [9] were documented, implementations of the patterns were written with the existing knowledge about OOP languages at that time. In OO implementation of patterns, pattern code is scattered and tangled with participant code, making it hard to reuse the pattern code and to document code. In Hannemann et al. [11], the effect of AOP techniques on the GoF design patterns implementation is discussed. They state that, the bigger improvements appear in implementation of patterns that have crosscutting structures between participant classes and the roles they play in the pattern. Thus, they suggest that it is worthwhile to apply AO techniques to pattern implementation, in particular, using the AspectJ programming language, which was specifically designed to modularize crosscutting structures.

In Rajan [23] the effects of the Eos programming language constructs on the implementation of the GoF patterns are analysed and compared to the implementations on AspectJ. Since the Eos language model is in part inspired on that of AspectJ, the results of implementing the design patterns in this language are at least as good as the ones obtained in AspectJ. The improvements attained with the use of this language are mainly related with being able to state the intent of design patterns more clearly, as well as producing more concise implementation code.

Rajan H. argues that the AspectJ implementation of the design patterns could be improved [23]. In the Eos language a new language construct is introduced, called *classpect*. This is the basic unit of modularity in this language, which unifies the concepts of aspects and classes. Classpects have all the capabilities of both classes and aspects (in AspectJ-like languages).

The primary difference between the AspectJ and Eos implementations is that all concerns are modelled as classpects in Eos. In addition, classpects enable instance-level advising. Eos allows for the creation of aspect instances and for selectively advising object instances, creating an implicit relation in the aspect between participant instances. This enables a given aspect to refer to just some instances of a class, instead of the AspectJ way, where aspects affect all instances of the classes advised. OT/J also allows one to choose to which class instances the aspect should be composed. Bound roles can be instantiated just like plain Java classes, creating a relation between a specific role instance and a class instance. Decorator is one example of a case where it may not be desired to decorate all the participant class instances, i.e., the aspect should not be composed into all the

participant class instances. Moreover, while in AspectJ relations between participant instances have to be emulated using, for instance, hash maps to store and retrieve relationships, in Eos a direct representation of the pattern relationship is provided, relating participant instances with each other in the classpect, as implicit advising structures. Likewise, OT/J Team instances maintain a mapping from base to role instances (see section 2.5), avoiding the need for extra data structures to keep these mappings.

For the analysis of the Eos implementation results, the criteria used by Hannemann et al. was used, as well as the *Lines of Code* metric, to measure size and *Close match to pattern intent* which evaluates to true if the intent of the pattern implementation closely matches the pattern specification. Rajan concludes that Eos implementation of some patterns becomes clearer than the ones in AspectJ. Also, the use of classpects provides better representation of the relationships between pattern participants (without resorting to the use of data structures), and allow for selective instance advising, which provides closer matches to the intent of the patterns. Neither *Lines of code* nor *Close match to pattern* metrics are used in the analysis of OT/J pattern implementations in this dissertation. This dissertation aims at assessing OT/J modularity capabilities, and these two metrics show little contribution for this goal.

A master thesis has been carried out by Sérgio Braz [1], whose topic is related with design pattern implementation in the AOP language CaesarJ. His work is focused on the implementation of 11 of the GoF patterns in this language and to produce comparative analyses between the results obtained with these implementations and implementations in other AOP languages.

The analysis by Braz conveys which CaesarJ mechanisms have been used for each design pattern implementation, and asserts about the possible reusability level for each pattern. Criteria used for the analysis to the reusability level, is divided into several groups: Direct language support, reusable modules, composition flexibility and no reuse.

Braz implementation analysis showed that Bridge yielded a reusable CaesarJ module, in contrast to AspectJ's result where no reusable module was produced for this pattern. However, in the present dissertation, the criterion for a pattern to be considered to have yielded a reusable module (see section 4.3) is different from the one in Sérgio's dissertation [1]. A module for a given pattern should have more than mere abstract declarations to be considered reusable, which is not the case of Bridge implementation in OT/J or in CaesarJ's.

The family polymorphism mechanism (see section 2.2.2) is supported by both CaesarJ and OT/J programming languages. This mechanism allows both languages to directly support Abstract factory and Factory method patterns, since these patterns have to do with creating objects which belong to a certain theme or family.

On the other hand, Visitor implementation produced a reusable module on both AspectJ and OT/J and that was not the case for CaesarJ version. However, in the AspectJ version the only non-abstract code in the reusable module are new methods to be added to pattern base participants. In the OT/J Visitor

implementation, the reusable team module maintains the context for the visitor, and concrete teams represent concrete visitors. Contrary to the AspectJ Visitor implementation where there is an extra class to specify which classes are visitors and visitable, in the OT/J version the visitors are the teams themselves and classes to be visited are specified in the concrete teams, avoiding the need for extra classes.

The work by Braz paves the way for comparisons between CaesarJ and OT/J. However, a systematic analysis cannot be made since a complete pattern repository has not been implemented in CaesarJ.

From the related work study, one can conclude that different programming languages and different programming mechanisms offer different benefits to the implementation of design patterns. OT/J benefits from some of the mechanisms, which are also used in other languages, such as family polymorphism.

6. Conclusion and future work

AOP is a recent programming paradigm, still subject to research and maturation. Several programming languages exist, but few studies have been done focused on their support for modularity and on comparison of different AOP languages. The implementation of design patterns has been used to provide insights about programming languages mechanisms and their constructs. Moreover, since design patterns usually crosscut over distinct features of a sysem, these prove perfect to assess some language potential for modularity.

For this dissertation two repositories of the 23 GoF patterns have been implemented in Object Teams for Java. These implementations are described and analysed in chapter 4. Moreover, code examples from these implementations are used all over this document, either to illustrate some OT/J mechanism or to provide examples for the analysis, as well as class diagrams for some of these examples. This analysis assesses which patterns are given to reusability in OT/J. Also, the Java and AspectJ design pattern repositories by HK and the respective AspectJ study by HK provide enough material to the comparative analysis in sections 4.7 and 4.8.

The implementations done for this thesis, as well as the conducted analysis, pave the way for future researches. The development of new repositories of implementations of the same Design Patterns in different AOP languages opens way for several comparative studies, whether between different programming paradigms, for instance OT/J and Java, or between different AOP languages, for instance OT/J and AspectJ. Also, studies directly comparable with the results of Garcia et al. [10] are possible.

In this dissertation the composition ability of patterns is studied only in individual cases, having at the most two instances of the same pattern active at the same time. Supplementary tests can be made in order to assess the composition ability of different patterns in the same application.

Similar studies to this dissertation could be conducted focused on different AOP languages. On the grounds that design patterns implementation brings insights on the constructs and mechanisms of the language used for the implementation, the existence of design pattern repositories in other languages would provide more material for future analysis. Moreover, increasing the number of languages implementing complete design pattern repositories provides a broader basis of available information for comparisons between programming languages and respective constructs.

7. References

[1] Braz S., A Qualitative Assessment of Modularity in CaesarJ components based on Implementations of Design Patterns. M.Sc. thesis, Universidade Nova de Lisboa, Departamento de Informática, 2009.

[2] Brichau, G., Haupt, M., Report describing survey of aspect languages and models. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, 2005.

[3] Cooper J., Java Design Patterns: A Tutorial. Addison-Wesley 2000. (Available at http://www.patterndepot.com/put/8/DesignJava.PDF)

[4] Czarnecki K., Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. Germany, Ilmenau, 1998.

[5] Ernst E., Family Polymorphism. In Proceedings ECOOP 2001, LNCS 2072, pp.303-326, Springer-Verlag, Budapest, Hungary, 2001.

[6] Ernst E., Ostermann K., Cook, W. R., A Virtual Class Calculus. 33rd ACM Symposium on Principles of Programming Languages (POPL'06). ACM SIGPLAN-SIGACT, 2006.

[7] Ernst E., Lorenz D., Aspects and polymorphism in AspectJ. AOSD 2003, Boston, Massachusetts, USA, 2003.

[8] Filman R. and Friedman D., Aspect-Oriented Programming is Quantification and Obliviousness. In Workshop on Advanced Separation of Concerns, OOPSLA 2000, pp. 21-35, Minneapolis, 2000.

[9] Gamma E., Helm R., Johnson R. and Vlissides J., Design Patterns: Elements of Reusable Object-oriented Software. Professional Computing Series, Addison-Wesley, 1995.

[10] Garcia A., Sant'Anna C., Figueiredo E., Lucena C., Staa A.v., Modularizing Design Patterns with Aspects: A Quantitative Study. In LNCS Transactions on Aspect-Oriented Software Development, Springer vol.1, pp.36-74, 2006.

[11] Hannemann J., Kiczales G., Design Pattern Implementation in Java and AspectJ. In OOPSLA 02, Seattle, USA, 2002.

[12] Herrmann S., A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. In Applied Ontology, Volume 2, Number 2 / 2007, pp. 181-207, IOS Press , Berlin, 2007.

[13] Herrmann S., Balancing Language Concerns: Who Decides? In SPLAT Workshop at AOSD'08, Brussels, Belgium, 2008.

[14] Herrmann S., Christine Hundt, Katharina Mehner, Translation Polymorphism in Object Teams. Technical Report 2004/05, Fak. IV, Technical University Berlin, 2004.

[15] Herrmann S., Confinement and representation encapsulation in Object Teams. Berlin, 2004.

[16] Herrmann S., Hundt C., Mehner K. and Wloka J, Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In AOSD 05, Chicago, 2005.

[17] Herrmann S., Object Teams: Improving Modularity for Crosscutting Collaborations. In Proceedings of Net.ObjectDays, Erfurt, 2002.

[18] Kiczales G., Hilsdale E., Hugunin J., Kersten M, Palm J. and Griswold W., An overview of AspectJ. In ECOOP 2001, 2001.

[19] Madsen O. L., Møller-Pedersen B., Virtual classes: a powerful mechanism in object-oriented programming, OOPSLA'89, New Orleans, Louisiana, USA, 1989.

[20] Mezini M, Ostermann K., Conquering aspects with Caesar. In Proceedings of AOSD 2003, pp. 90–99, Boston, USA, 2003.

[21] Monteiro M., Fernandes J., Towards a Catalogue of Refactorings and Code Smells for AspectJ. Transactions on Aspect-Oriented Software Development, Springer LNCS vol. 3880/2006, p. 214-258, 2006.

[22] Parnas D. L., On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), pp. 1053-1059, December 1972.

[23] Rajan H., Design Pattern Implementations in Eos. In PLoP '07, Conference on Pattern Languages of Programs, Monticello, September 2007.

[24] Riehle D., Gross T., Role Model Based Framework Design and Integration. In Proceedings of OOPSLA '98. ACM Press, Page 117-133, 1998.

[25] Reenskaug T., Wold P., Lehene A., Working with Objects—The OOram Software Engineering Method, Addison-Wesley/Manning, 1996.

[26] Steimann F, Wissensverarbeitung R., Role = Interface - A merger of concepts (2001). In Journal of Object-Oriented Programming, Vol.14, Page 23-32, 2001.

[27] Tešanovic A., What is a pattern? Paper in Design Pattern seminar, IDA, 2001.

[28] Yuen I., Robillard M., Bridging the gap between aspect mining and refactoring. In LATE '07, Proceedings of the 3rd workshop on Linking aspect technology and evolution, Vancouver, Canada, 2007.