Universidade do Porto
Faculdade de Engenharia

Generic Roles: Reducing Code Replication

Fernando Sérgio Barbosa

September 2013

Scientific Supervision by

Doctor Ademar Aguiar, Assistant Professor
Department of Informatics Engineering

In partial fulfillment of requirements for the degree of
Doctor of Philosophy in Informatics Enginnering
by the Doctoral Program in Informatics Engineering

Contact Information:

Fernando Sérgio Barbosa
Escola Superior de Tecnologia
Instituto Politécnico de Castelo Branco

Avenida do Empresário, s/n
6000 - 767 Castelo Branco
Portugal

Tel.: +351 272 339 300
Fax.: +351 272 339 399
Email: fsergio@ipcb.pt

This thesis was typeset using the free LaTeX typesetting system, originally developed by Leslie Lamport based on TeX created by Donald Knuth. The body text is set in Latin Modern, a Computer Modern derived font originally designed by Donald Knuth. Other fonts include Sans and Typewriter from the Computer Modern family, and Courier, a monospaced font originally designed by Howard Kettler at IBM and later redrawn by Adrian Frutiger.

. . . to my wife, Teresa

. . . to my daughters, Margarida and Rita

This page was intentionally left mostly blank.

Abstract

For many years the concept of modularity has been considered a very important part in the development of large software systems. Modules help to manage the system's complexity by decomposing it in smaller parts. These parts can be assigned to individuals or teams for development. Modules hide the information they manipulate behind an interface, allowing its developers to develop the module independently of any other module in the system. Developers can change the information their module manipulates and even the way it does it without the need to consult other developers, and breaking their code. This enables the interchangeability of modules, allowing one module to be substituted by another without further modifications to the system. It also reduces compiling time as modules can be precompiled.

The concept of modularization lead to the dawn of several decompositions techniques, each with its own ideas on how a system should be decomposed into modules. For each decomposition technique and to help programmers extract the most benefits from modularization, several programming languages provide support for expressing modules. In object-oriented decomposition, for example, several programming languages provide support to define abstract data types, usually in the form of classes.

Ideally, each module would capture one coherent concept that would deal with a set of coherent concerns related to the module's concept. Unfortunately that is not always true. Sometimes, modules need to interact in more complicated and intertwined manners. Sometimes, the need to communicate with other modules lead one module to assume concerns that are not related to its main concern. This is, as argued by many authors, because only a single decomposition strategy is used . To avoid this we may need to use more than one decompositions strategy or extend an existing one.

Code clones are an outcome of the lack of other decomposition strategies, among others sources. Code cloning in a system is considered a bad thing with multiple drawbacks. One of the more known problems is the inconsistent maintenance: bugs are fixed in some clones, but not in others. Another major objection to the use of cloning is that it degrades the design of the system over time. Awkward, verbose designs lead to the accumulation of irrelevant code that ends up obscuring the original intent of the code.

In this dissertation we study the reduction of the code replication using modularity as a cornerstone around which our solution must be based. We intend to reduce code replication using another kind of module, which we can use to extend the object-oriented strategy. The module that we will study is the role. Roles have been used to specify an object's behavior related to a specific collaboration in the modeling stages of a system. But in the implementation that specification is merged with all the object's other collaboration behaviors, inside a class. It is a purpose of this dissertation to take the specification from the design phase to the implementation phase and study its impact on the code replication problem.

The solution proposed in this dissertation is to use roles as a way to compose classes and thus reducing code replication. To pursue this goal a role language is designed and a suitable compiler is implemented. The JavaStage language allows a programmer to "program with roles". It introduces several features like a powerful renaming mechanism and to state role dependencies easily.

Using JavaStage and roles we were able to identify several refactorings that enables us to write the replicated code as roles thus removing the code clone. The use of these refactorings proved their value when applied to a series of case studies developed to assess the amount of duplicated code that could be removed using roles.

As already mentioned, the modularity principles were followed throughout the design of the proposed solution and, to assess the reusability of roles, a role library was started. Its evolution corroborates the idea that roles are reusable modules and can be used as a compositional element.

Resumo

A modularidade é, já há muitos anos, vista como uma peça importante no desenvolvimento de sistemas informáticos de grandes dimensões. Os módulos ajudam a gerir a complexidade do sistema decompondo-o em várias partes mais pequenas que podem ser adjudicadas a um indivíduo ou a uma equipa para desenvolvimento. Os módulos encapsulam a informação que manipulam e o modo como a manipulam, permitindo assim aos seus autores desenvolvê-lo independentemente dos outros módulos do sistema. Os autores de um módulo podem, inclusivé, alterar o modo de representação da informação à medida das suas necessidades, ou até alterar o modo de a manipular, sem consultar os autores dos outros módulos e sem quebrar os módulos destes. Isto permite a permutação de módulos, podendo assim um módulo ser substituído por outro sem que o sistema sofra outras alterações. Também permite um tempo de compilação mais rápido devido a poder-se pré-compilar os vários módulos.

O conceito de modularização levou ao surgimento de várias estratégias de decomposição, cada uma com as suas ideias sobre como um sistema deveria ser decomposto em módulos. Para cada uma destas estragégias, e para ajudar os programadores a usufruir ao máximo dos benefícios da modularização, surgiram várias linguagens de programação com suporte para a representação de módulos. Por exemplo, na decomposição orientada a objetos, muitas linguagens permitem a definição de tipos abstratos de dados, normalmente na forma de classes.

Idealmente, cada módulo capturaria um conceito coerente e esse conceito lidaria com um conjunto coerente de responsabilidades ligadas ao conceito principal do módulo. Infelizmente, não é assim. Por vezes os módulos precisam de interagir de forma mais complexa e interligada com outros módulos. Outras vezes a necessidade de comunicar com outros módulos leva-os a assumir responsabiliades que não estão de acordo com a sua "razão de ser". Isto, como é argumentado por muitos autores, é uma consequência do uso de uma única estratégia de decomposição. Para evitar isto provavelmente necessita-se de uma outra forma de decomposição ou expandir uma já existente.

O aparecimento de clones no código é uma consequência do uso de uma única estratégia de decomposição, entre outras causas. A presença de código repetido num sistema é

considerada um mau indício e acarta muitas consequências nefastas. Um dos problemas mais conhecidos é a inconsistência na manutenção: os bugs são reparados em algumas instâncias do código repetido mas não em todas. Outro grande problema do uso de cópias de código é que o seu uso irá degradar o design do sistema a longo prazo. Isto leva à acumulação de código irrelevante que acaba por obscurecer o objectivo inicial do código.

Nesta tese tenta-se estudar a redução de código repetido usando a modularidade como "pedra de toque" em torno da qual a nossa solução se baseará. Tencionámos reduzir a repetição de código usando outro tipo de módulo, que nos permita expandir a decomposição orientada a objetos. O módulo que iremos estudar será o role. Os roles têm vindo a ser usados para especificar o comportamento de um objeto em relação a outro objeto, dentro de uma colaboração, na etapa de modelação do sistema. Mas, na fase de implementação, essa especificação é agrupada com todos os comportamentos que o objeto exibe em todas as suas colaborações, dentro de uma classe. É um dos propósitos desta tese levar esta especificação desde a fase de modelação até à fase de implementação e estudar o seu impacto no problema de repetição de código.

A solução proposta nesta dissertação é usar os roles como uma maneira de compor classes e assim reduzir a repetição de código. Para atingir este objectivo uma linguagem de programação que suporta roles será idealizada e um compilador adequado desenvolvido. A linguagem JavaStage permite a um programador "programar com roles". Ela introduz algumas funcionalidades como um mecanismo de renomeaçao de métodos poderoso e uma forma de exprimir as dependências do role facilmente.

Usando a JavaStage e os roles fomos capazes de identificar várias refactorings que nos permitem expressar o código presente nas várias repetições como roles e assim remover o código repetido associado. Estas refactorings mostraram o seu valor quando usadas numa série de casos de estudo que foram desenvolvidos com o propósito de avaliar a quantidade de código repetido que se poderia remover usando roles.

Como já foi mencionado os princípios da modularidade foram seguidos ao longo da concepção da solução proposta e para avaliar a reutilização dos roles a construção de uma biblioteca de roles foi iniciada. A sua evolução corrobora a ideia de que os roles são módulos reutilizáveis e podem ser usados como um elemento composicional de classes.

Contents

List of Figures

List of Tables

Preface

Like many in my generation, the computer world opened to me by a small and slim black box that we connected to a television: the ZX Spectrum that my parents bought me and my sisters when I was fifteen. Initially it was used to play games but as I learned that one could program it, it became even more interesting. Besides playing the games I could write my own games and other programs. I still remember my first attempt at programming, even before reading the BASIC manual: a game where a car chased another. It was a very naive approach as I assumed that the computer would understand what I wanted and so it was only 5 lines of code! It did not work (obviously!) and I solved the errors it had by beginning each line with a "rem" - it actually commented the line so no wonder the errors were gone! Another memory of those early days involved taking the computer to the classroom where I showed my Biology teacher and colleagues a program I wrote that simulated the connection of several proteins, with animations and all! It was incredible what you could do with only 48K of memory. I love programming ever since.

And while every programmer does enjoy programming there is one thing they do not like: writing the same code over and over. The evolution of programming languages, especially under the Object-Oriented Paradigm, brought code reuse to a higher level especially by the extensive use of libraries. Still the problem remains. This is the case even when we are programming using best practices, or using design patterns. Every time we program an Observer or a Singleton Pattern we feel that "we have done this before". With patterns we don't need to solve the same problem over and over, we just reuse the solution several times. But because the code for implementing the patterns is very similar between pattern instances, the feeling that we are repeating the same code over and over is present.

When studying Aspect-Oriented Programming (AOP), for a possible future thesis in the area, and analyzing several aspect examples I felt the same "déjà vu" feeling. A number of aspects were implemented in very identical ways, or with only little changes. This lead me to believe that AOP is not enough to remove code replication and could benefit from techniques for avoiding code replication as well. The study of AOP continued but this problem remained in the background, and possible solutions were germinating. One such

solution lead to the role research field that was unknown to me at the time. It was a shock to see that several of my new ideas were already debated long ago! The core ones were not covered, though, and the code replication problem stepped from the background to the foreground. The AOP study was put on hold and, as new ideas associated with roles appeared and the role study progressed, it was completely put aside. Over the years that took me to undertake this work the original ideas have matured and evolved, but they all reached the conclusion in a recognizable form.

I thank my supervisor Ademar Aguiar for its initial guidance and ensuring me that the path I was leading would take me where I wanted to go,. His steering over the years was also very important in focusing my research effort, even in spite of the distance. I must also acknowledge Professors Eugénio Oliveira and Augusto de Sousa, of the Doctoral Program in Informatics Engineering (ProDEI) for all the support provided in my first year and as representatives of all the teachers in the Doctoral Program, to whom I also thank. My thanks also to the members of the supervising committee, Professor João Miguel Fernandes and Professor João Pascoal Faria, for their valuable advices and suggestions for improvement. I also thank my institution - Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco - for the financial support and for alleviating my work load so that I could better proceed with my thesis work.

On a more personal note I thank my wife Teresa and daughters Margarida and Rita for their unconditional support even when things were not going as well as I liked them to be. I especially thank them for their sympathy and understanding when the work took me away from them often and for long periods.

Fernando Sérgio Barbosa

Porto, Portugal

July 2013

Chapter 1

Introduction

1.1 Software Reuse

Every software engineer's goal is to deliver a working system that fulfills all users' requirements. But this is not enough. The system must also be reliable, easy to maintain, easy to evolve, and as cheap as possible to develop. Over the time software engineers have developed techniques and methodologies that reduce software complexity, improve readability, enhance reuse of code and facilitate evolution. On the basis of many of the techniques is code reuse, because with code reuse both the cost and the production time of the software are decreased.

Modularization [Par72] is the mean by which code reuse has been made possible. Modularization also leads to decomposition and composition phases in the software life cycle. Decomposition enables us to identify the modules present in a system by taking the system and breaking it into smaller systems. Decomposition also contributes greatly to program comprehension because we only need to understand a part of a system. If a module is well written it is complete and highly, if not completely, independent from other modules. We can therefore analyze each module per se. We can also develop each module independently, thus reducing time in the implementation phase, by assigning different modules to different developing teams.

Composition, on the other hand, is used to produce the final system by putting together the various modules. There are also many ways to compose a system from its modules but they tend to depend on the decomposition strategy used.

Over the time several decomposition methodologies have been proposed, each presenting its own definition of a module or set of modules. Nowadays, object-oriented (OO) decomposition is the dominant methodology. In OO we can assume that the smallest module available is the class. In the functional paradigm, another popular decomposition strategy, the smallest module is the function.

These evolutions lead to a significant improvement in code reuse but there is a problem that still persists: that of code replication. This is a real problem, especially in large systems [MLM96, BYM+98, Bak95, JMSG07, KG06b, LLMZ06]. Code replication has negative effects on development time and programmers motivation [MLM96, BYM+98, JDHW09]. Programmers like to write code, what they do not like is to write the same code over and over. If a programmer has to do the same code over and over it leads to tiredness and to errors. This leads to the reuse of code on the copy-paste principle with all the associated problems, like having to trace every instance of the replicated code to fix a bug or to introduce a new feature. This problem of code replication will be the focus of study of this work. There are several reasons for code replication but we will not deal with all of them [RC07]. We are interested in those situations where code replication is due to a poor modularization of the system, underlying language or programming paradigm.

One reason for code clones that we will try to overcome are modularization limitations. Even though OO is the dominant decomposition strategy it has restrictions in modularization. There are always concerns that do not fall neatly into a class and are shared by a set of classes. These are called the crosscutting concerns because they exist across all the classes that deal with them. These crosscutting concerns may lead to severe code replication. This code replication may appear because classes must have the same code for dealing with the same crosscutting concern.

There is also code that is very much the same because classes have to do similar tasks. Developers that use design patterns [GHJV95] in their systems tend to use the same, or similar, code for each instance of the same pattern, whenever possible. In a system that uses the same pattern several times it is likely that some of the code is replicated. Even if the system only uses an instance of a design pattern the developer has a déjà vu feeling that she is writing the same code again. To reduce this code replication in the design pattern instantiation several tools were developed, but we feel that if we can capture each participant of a design pattern into a module and reuse that module the outcome is more effective than using tools.

We need a new way to further reduce code replication and we believe that it is possible

to find a way to do it. This new way must cope with the modularization principles as well. It must also provide a way to decompose the system into new modules and then compose the new modules to form the system.

Even though the several decomposition strategies are worthy, we will focus our discussion in the OO decomposition, or the more relevant that derived from it, as it is the dominant one. We will take a more detailed look at the role approach [RWL96, RG98, Ste00], as we believe that roles are the most likely technique to produce good results in reducing code replication as they offer a good way to decompose a system.

Objects interact with each other through collaborations. In each collaboration it enters, the object plays a specific role. The role represents the behavior of an object with respect to each collaboration. The same object may play several roles, depending on the objects it interacts with. Classes can therefore be defined by the several roles its objects may play in a system. Some of these roles are defining, that is, the class primary concern is to deal with that type of interactions. These roles can be directly implemented in the class itself. Other roles the class plays are superimposed, that is, its objects only assumes these roles because they are required to interact with other objects. These superimposed roles eventually lead to crosscutting concerns and to replicated code, if we implement them in the class.

If we decompose a class into roles we can describe the superimposed roles independently of the class that plays them. This way we can develop an independent role with the crosscutting code. When composing the class the role behavior is added to the class behavior. This is done for every class that plays the role. Classes can therefore share the role implementation, thus reusing the code instead of replicating it.

Roles retain all of the OO features [CLD05] but offer an even more granular way of decomposition as several classes can play the same role. Thus we can place a crosscutting concern in a role and make all the classes that need to address that concern play that role. To compose the system, roles also have advantages because they can be seen as a smaller module than the class so they act at a finer grain level and can influence classes as well as other roles.

Many role researchers have concentrated in dynamic roles and evolving objects of a system at run-time by attaching, or detaching, roles to them. Even if this is a good way of using roles we will not directly address dynamic issues but limit our work to the problems of code reuse to which a static approach is sufficient. Our goal is to build generic roles: roles that can be easily tuned to fit a particular class.

1.2 Research Goals

Taking into account the previous concerns this dissertation aims at contributing to the software engineering body of knowledge by proposing new ways of reusing software.

The main research goal is to ***reduce replicated code*** by providing a new way of ***composing classes*** that is an ***extension to the object-oriented paradigm***. More concretely we intend to use ***roles as a component for classes***. This way we intend to ***increase code reuse*** while following all the ***modularity*** guidelines and retain all its advantages. We intend to explore how roles can be made modular and therefore reusable, thus contributing to diminish code replication. An intended outcome of the work is to build a library of reusable roles. This will show how roles can be used to reduce code replication and provide other ways of reusing code.

These research goals are further detailed in Chapter 5.

1.3 Research Strategy

Software engineering is still maturing as a research area. Software development has specific characteristics that suggests its own research paradigm, combining aspects from other disciplines: it is a human creative phenomenon; software projects are costly and usually have long cycle times; it is difficult to control all relevant parameters; technology changes very frequently, so old knowledge becomes obsolete fast; it is difficult to replicate studies; and there are few common ground theories.

A categorization proposed at Dagstuhl workshop [THP92], groups research methods in four general categories, quoted from Zelkowitz and Wallace [ZW98]:

- **Scientific method.** *"Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis."*
- **Engineering method.** *"Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement."*
- **Empirical method.** *"A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis."*
- **Analytical method.** *"A formal theory is developed, and results derived from that theory can be compared with empirical observations."*

These categories apply to science in general. Effective experimentation in software engineering requires more specific approaches. Software engineering research comprises computer science issues, human issues and organizational issues. It is thus often convenient to use combinations of research approaches both from computer science and social sciences. The taxonomy described by Zelkowitz and Wallace [ZW98] identifies twelve different types of experimental approaches for software engineering, grouped into three broad categories:

- **Observational methods.** *"An observational method collects relevant data as a project develops. There is relatively little control over the development process other than through using the new technology that is being studied"*. There are four types: project monitoring, case study, assertion, and field study.
- **Historical methods.** *"A historical method collects data from projects that have already been completed. The data already exist; it is only necessary to analyze what has already been collected"*. There are four methods: literature search, legacy data, lessons learned, and static analysis.
- **Controlled methods.** *"A controlled method provides multiple instances of an observation for statistical validity of the results. This method is the classical method of experimental design in other scientific disciplines"*. There are four types of controlled methods: replicated experiment, synthetic environment experiment, dynamic analysis, and simulation.

The best combination of methods to use in a concrete research approach is strongly dependent on the specific characteristics of the research study to perform, viz. its purpose, environment and resources. Hereafter, the research methods referred will use this terminology. Further description of each method can be found in [ZW98].

Based on the expected results and contributions of the work presented in this dissertation, the research strategy comprised a mix of *observational* and *historical* methods (case studies, literature search, and lessons learned) to substantiate the replicated code reduction contribution (Chapter 7 (p. 111)) and the *observational* method to provide evidence for the generic roles and role library contributions (Chapter 8 (p. 135)). A detailed rationale of the methods is presented in Chapter 5 (p. 67).

1.4 Expected Results

The expected outcomes of this thesis are the following contributions to the body of knowledge in software engineering:

- **Providing roles with a supporting language.** Roles are used for modeling and documentation of systems but not in its implementation. This introduces a gap between modeling and implementation as the artifacts from the earlier stages are not used in the final solution. Each developer must find its own way of implementing the role view used in the design stage. This opens the door to misconceptions and ultimately to errors. With a programming language that supports roles developers can reproduce the design decisions more accurately and thus less error prone.

- **Refactorings for removing code clones.** Nowadays refactoring code is a powerful technique to remove a number of flaws out of the system code, making it more adaptable, easy to understand, etc. Code cloning is identified as a flaw in a system's code and some known refactorings can be used to reduce it. We intend to extend the known collection of refactorings by adding role related clone reduction refactorings.

- **A role library.** To demonstrate the reusability of roles there is no better way than to build a library of reusable roles. This way we can provide a reusable solution for recurrent problems that classes cannot cope with. To start the role library we intend to analyze the design patterns present on the Gang of Four book of design patterns [GHJV95]. These patterns provide a solution to recurrent problems so if we can develop generic roles for these patterns they can be used in the several instances of the pattern or, at least, a great number of them.

- **Evaluation of the extent roles can reduce code clones in a system.** We intend to verify whether or not roles can reduce replicated code in a system. For that we will conduct a series of case studies where we will try to reduce the replicated code present in the target system. We expect that the amount of replicated code is significantly reduced in every system we analyze. We also expect to reduce the number of code lines in the system.

1.5 How to Read this Dissertation

The remaining of this dissertation is logically organized into three parts, with the following overall structure:

Part 1: State of the art. The first part reviews the most important concepts and issues relevant to the thesis:

- Chapter 2, "Code clones" (p. 11), provides a literature review on the field of code clones, its definitions, types, detecting technologies and ways to remove them.

- Chapter 3, "Class composition" (p. 25), provides a description of some of the most used composition techniques in software development.
- Chapter 4, "Roles" (p. 47), focus on the role concept, discussing the dynamic view of roles and the static view of roles, their problems and advantages, and how to model with roles.

Part 2: Problem & solution. The second part states the problem researched and the proposed solution:

- Chapter 5, "Research problem and solution" (p. 67), lays both the fundamental and specific research questions in scope for this thesis, and overviews the proposed solution.
- Chapter 6, "JavaStage" (p. 77), presents the JavaStage language, its syntax, how to program with roles using it, the way it is implemented and its limitations. It also compares it to other approaches.
- Chapter 7, "Removing Clones" (p. 111), presents the clones that are still unresolved by traditional techniques and the proposed refactorings that use roles and how they can be used to remove code clones.

Part 3: Validation The third part presents the validation of the thesis and the conclusions of the dissertation:

- Chapter 8, "Towards a Role Library" (p. 135), discusses the building of a role library based on design patterns.
- Chapter 9, "Case studies" (p. 135), presents the case studies used to validate the dissertation, their setup and results obtained.
- Chapter 10, "Conclusions" (p. 197), drafts the main conclusions of this dissertation, and points to further work.

For a comprehensive understanding of this dissertation, all the parts should be read in the same order as they are presented. Those already familiar with code clones, class composition or roles, and only want to get a fast but detailed impression of the work, may skip the first part, and go directly to Chapter 5 (p. 67).

Some typographical conventions are used to improve the readability of this document. Refactory names always appear in Small Case style. Relevant concepts are usually introduced in *italics*. Book titles and acronyms are type-faced in ALLCAPS. References and citations appear inside [square brackets] and in highlight color — when viewing this document on a computer, these will also act as *hyperlinks*.

Part I

State of the art

Chapter 2

Code clones

When developing a system, programmers usually reuse solutions by copying code and then modifying it to fit a new purpose. This leads to code cloning because several fragments of a system will be identical or, at least, very similar. Even if this has immediate advantages like a reduced development time in the long run it will reclaim its toll. An obvious problem of code clones is the increased system size. A system with code clones is also more difficult to maintain [MLM96, BYM+98, JDHW09] and more error prone [JDHW09]. Nevertheless code clones are found in several systems, especially in large ones [DRD99, Kon97, MLM96, KdMM+96, KKI02, Bak92, BYM+98], ranging from 5% to 23% of total system code [MLM96, BYM+98, Bak95, JMSG07, KG06b, LLMZ06].

There are many reasons for the occurrence of clones [KdMM+96, BYM+98, Bak95, KBLN04], one of them are crosscutting concerns, that is, concerns that a class must deal with that are not its main concern. When several classes deal with the same concern they tend to use similar code. This is more frequent in languages that do not support multiple inheritance. With multiple inheritance we could place the concern in a superclass and all subclasses inherited the same behavior. With single inheritance we tend to replicate the common behavior in all classes, if we cannot find a common superclass.

To identify clones there are several techniques and tools . There are also many proposals on how to remove clones [FR99, HKKI04, KH00, JH06, RD04] and even semi-automated

tools [HKK+04]. Nevertheless those techniques and tools are not capable of removing all the replicated code.

All the problems associated with code clones lead them to be considered as a bad smell, hinting that the system needs to be refactored [Fow99]. Fowler in [Fow99] says that:

> *Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.*

This motivated the development of several techniques to identify code clones [Kon97, Bak95, LLMZ06, Kri01]. The removal of code clones has been the subject of several studies [FR99, HKKI04, KH00, JH06, RD04]. This section presents an overview of the code cloning problem, its origins, types of clones, what problems arise when we use code cloning, when it is beneficial to use them, how can clones be removed and what type of clones cannot be removed by current techniques.

In this chapter we give an overview of code cloning, namely reasons for its occurrence, types of clones, ways to detect clones and techniques to remove them.

2.1 Origins of Clones

There are many reasons for the appearance of code clone, ranging from intentional cloning to independently developed code. Roy and Cordy in [RC07] present 24 factors for the presence of duplicated code. Those factors are grouped in categories and subcategories. In table 2.1 we present those factors, albeit in a different format from the one used by Roy and Cordy. We then proceed to briefly explain each one.

2.1.1 Development Strategy

The simpler way of duplicating code is by copy-pasting it and, optionally, modifying it. It is a fast way of reusing safe code, and used when dealing with crosscutting concerns [KSN05]. Copy-paste is also used when forking a system, the code is copied and expected to be altered to cope with different specifications. Depending on the nature of the new specifications the code can remain pretty much the same or suffer many transformations. Sometimes related components have similar functionality/logic so it is useful to clone and modify existing components to create new components when other forms of reuse are not available.

Clones also occur due to the use of tools that generate code. Tools have a uniform way of generating code so it is likely to have clones. After merging two systems, the merged system may contain clones due to the fact that similar functionalities may have used

CATEGORY	SUBCATEGORY	FACTOR
Development Strategy	Reuse Aproach	Simple reuse by copy and paste
		Forking
		Design reuse
		Functionalities/Logic reuse
	Programming Approach	Generative programming
		Merging similar systems
		Delay in restructuring
Maintenance Benefits	Avoiding	Risk
		Unwanted design dependencies
	Ensuring	Robustness
		Better performance in real time programs
	Reflecting	Design decisions (e.g., crosscutting)
Overcoming Underlying Limitations	Language Limitations	Lack of reuse mechanisms
		Abstraction creates complexity
		Abstraction is error-prone
		Significant efforts in making abstractions
	Programmers' Limitations	Time Limitations
		Performance by LOC
		Lack of ownership
		Lack of knowledge in the domain
		Difficulty in understanding large systems
Cloning by Accident	Language Paradigm	Protocols to interact with API and Libraries
	Programmers Working Style	Programmer's mental model
		Unknowingly implementing the same logic by different programmers

Table 2.1: Clone Factors (adapted from [RC07]).

similar implementations in both systems, even when developed by different programmers. The fact that developers usually postpone code restructuring can also introduce clones (for example instead of creating a method they simply copy the code).

2.1.2 Maintenance Benefits

Sometimes the benefits of duplicating code cannot be ignored. For example, in financial situations [Cor03], and given the fact that most financial applications have little differences between them, the risk of developing new code, instead of duplicating and changing a thoroughly tested code, is too high. Also code clones can be used to reduce complexity in source code whenever abstractions are difficult to form [TBG04].

System with clones can be more robust, especially when redundancy is required. In these systems different teams may develop the same functionality, or it can be done in

slightly different ways by the same team. Therefore clones are likely to appear. In these situations clones cannot be removed as they add to the system robustness. Also in real time systems the performance cost associated with calling methods or use of other code reuse techniques may result in code being cloned.

2.1.3 Overcoming Underlying Limitations

Other sources of clones are language limitations, like languages that only support single inheritance. In these languages, classes that have similar concerns but inherit from different superclasses tend to use code clones. Writing reusable code requires a bit of an effort and it may be simpler to maintain a set of clones than to write a general solution. This is especially true if the code in question is a critical system functionality. It is less error prone to copy the critical code than to try and reuse it with the possible introduction of errors.

Programmers' limitations can also contribute to replicated code. For example, when time limits are tight it is easier and faster to copy and modify than to write a general solution. Also a programmer may have its productivity assessed by the amount of code written. It is therefore beneficial to copy code than to write a generic solution with fewer lines of code. When the system is hard to understand a programmer, unfamiliar with the system, tends to duplicate the code that already exists and modifies it to his purposes, in an example-oriented implementation. This can also occur when a developer has a loose grasp in the problem domain and copies the solution from a similar problem. Finally a developer can be asked to modify a system but cannot modify its source code due to various factors related to ownership. Again the solution is to replicate and then modify the code.

2.1.4 Cloning by Accident

Clones can appear by chance when developers independently use the same solution for similar problems. The use of design patterns [GHJV95] may contribute to this phenomenon as they describe solutions to recurrent problems in software systems and programmers familiar with them tend to use the same designs and solutions. Equally a developer may reuse a solution he had used in another problem without realizing it. The use of APIs and libraries often need the programmer to follow a series of steps that inherently will lead to similar code when the API/library is used several times.

2.2 Consequences of Clones

An obvious consequence of clones is the increased system size. This can be quite significant in systems that have size limitations, as for example systems for mobile devices [Joh94].

Cloning code can lead to the propagation of bugs if the cloned segment is faulty. Even when the segment is modified, the bug tends to persist as the developers focus is on the changes and not on the overall code. If cloning is heavily used then the bug propagation probability is significantly increased [Joh93, LLMZ06].

Code clone impairs maintenance [MLM96, BYM+98, JDHW09] and evolution [GFGP06] of a system. To maintain a system, developers need to identify all the clones and change each instance accordingly. Thus code duplication also means duplication of correction efforts as the time spent in correcting a clone is duplicated in all clone occurrences. This contributes to a greater maintenance cost [MNK+02].

Clones may also raise a particular maintenance problem: the inconsistency in updating. As said above, a bug in a code block is propagated to all its clones. If the bug is fixed in most, but not all, occurrences then an inconsistency in the update is made in maintenance [LLMZ06]. These inconsistencies may be hard to detect if the unchanged code is rarely used. Inconsistencies can also occur when modifying the clones to introduce new behavior. All, but a few, clone instances are updated with the new behavior, leading to inconsistent behavior.

A related problem is the introduction of new bugs. A developer may copy the code to implement a new functionality, but his lack of expertise in the system and not fully understanding the initial code will eventually lead to new bugs. Even for experienced programmers, this procedure is error prone [BYM+98].

Code clones also have negative effects in program comprehensibility [Gie07, KdMM+96], mostly because the principle of locality of functionality is violated, i.e., one specific aspect of functionality is implemented by each instance of a clone. To fully understand the system one must know all clone instances. This can also affect the evolution of a system.

The use of clones can also indicate that the design if flawed or it may prevent design improvement [RC07]. It can indicate that not enough reuse mechanisms have been used, like inheritance or delegation, and that the efficient reuse of such code is therefore compromised.

Clones are traditionally regarded as having a bad impact on a system, and while this is generally true, it is not necessarily so for all clones. The use of clones can also have positive influences in a system. Some studies lead their authors [KG06a, KBLN04, KSN05, Cor03] to state that clones can often be used in a positive way. In a recent study with fifteen open source systems [HSHK10] the authors claimed that the presence of duplicate code did not have a negative impact on software evolution. Kapser and Godfrey describe in [KG06a]

eight cloning patterns. For each pattern they studied their advantages and disadvantages to software development and maintenance. In this respect their results show that not all patterns are harmful, and some are even beneficial. Therefore, they conclude, that when considering the refactoring of a clone, some concerns such as stability, code ownership, and design clarity need to be considered.

Whether they have a good or bad impact in a system, it is important to be aware of code clones in the system. For this purpose, clone detection techniques have been proposed and widely applied.

Clones can either be removed or managed. As mentioned, some studies showed that some clones do not affect maintainability and even increase code comprehension. These clones should be maintained rather than eliminated. Harmful clones should be eliminated.

2.3 Types of Clones

The definition of what is a clone differs between authors as they tend to use a definition that suits their target language or detection techniques [RC07]. Several authors define clones as code fragments that are similar, but the definition of similar varies [BYM+98, BB02, KKI02, KG04, Kon97, LLMZ06].

There are many terms used to clone categorization [BMD+99, MLM96], but throughout the remainder of this section and the remainder of this thesis we will use Roy and Cordy [RC07] categories. Roy and Cordy use 4 clone types:

Type I - Similar code fragments, except layout and comments.

Type II - Structurally similar code segments with differences in identifiers, literals or types.

Type III - Clones with added, removed or changed statements. They may have differences in identifiers, literals and types.

Type IV - Fragments have the same functionality but the code is different.

This categorization has the advantage to provide an increasing level of differences between the clones. Type I clones are the most similar and Type IV clones are the most different. These levels also correspond to an increasing difficulty in detecting code clones.

2.3.1 Type I Clones

Code fragments of Type I clones are identical, with the exception of layout, variations in white space and comments. The following examples are based on [RC07].

```
if( a >= b ){
  c = d + b;     // comment on this
  d = d + 1;
```

```
}
else {
  c = d - a;     // comment on that
}
```

The following code is an exact copy of the previous code, with only the comments changing place

```
if( a >= b ){
  // comment on this
  c = d + b;
  d = d + 1;
}
else {
  // comment on that
  c = d - a;
}
```

The same code could be written with a different layout but it would be, nevertheless, a Type I clone.

```
if( a >= b )
{
  c = d + b;     // comment on this
  d = d + 1;
}
else
{
  c = d - a;     // comment on that
}
```

2.3.2 Type II Clones

In Type II clones, fragments are modified by changing identifier names (variables, methods, etc.), literals and types, and all changes admitted to Type I as comments and layouts. The code segment

```
if( a >= b ){
  c = d + b;     // comment on this
  d = d + 1;
}
else {
  c = d - a;     // comment on that
}
```

has a Type II clone in the fragment

```
if( m >= n )
{ // comment on this
  y = x + n;
  x = x + 5;
}
else
{ // comment on that
  y = x - m;
}
```

2.3.3 Type III Clones

In Type III clones, fragments are modified by changing, introducing or removing statements and all modifications made to Type II clones.

The code

```
if( a >= b ){
  c = d + b;     // comment on this
  d = d + 1;
}
else {
  c = d - a;     // comment on that
}
```

could be copied and then modified to a Type III clone as

```
if( a >= b ){
  c = d + b;     // comment on this
  d = d + 1;
  e = 1;         // added statement
}
else {
  c = d - a;     // comment on that
}
```

2.3.4 Type IV Clones

Type IV clones are clones that provide the same functionality but differ in the code. These clones are usually not copied, but can occur when different developers implement the same logic.

The following methods are Type IV clones as they both calculate the factorial of a number.

```
int factorial( int n ){
  int fact = 1;
```

```
  for( int i = 2; i <= n; i++ )
      fact *= i;
  return fact;
}

int recFactorial( int n ){
  if( n == 0 ) return 1;
  return n * recFactorial( n - 1 );
}
```

While these types are useful for clone categorization they may be a bit restrict as many kinds of clones fall into the same type. It is hard to describe the clones that refactoring cannot remove in terms of these types only. When necessary we will add other clone descriptions to better identify clones.

2.4 Detecting Clones

Clone detection is an essential activity for all clone activities be its removal or management. There are several techniques to detect clones which can be categorized as:

Line-based - a line by line comparison is made in the source code [DRD99, Bak95, Joh93]. Each line is compared with all other lines. If a minimum number of consecutive lines of code are identical to other lines of code they are marked as code clones. Depending on the granularity of the algorithm, or if they treat lines as textual lines or instruction lines, these techniques can or cannot detect layout differences between code. This means that techniques that rely only in a source code line by line technique cannot detect Type I clones with different layouts as in our third example of these clones. These techniques are vulnerable to different coding styles.

Token-based - The source code is scanned and divided into tokens. The tokens are then searched for identical token sequences. When identical token sequences have more than a minimum number of tokens the sequences are marked as clones [KKI02, LLMZ06]. Representing a source code as a token sequence enables the detection of clones with different line structures, which cannot be detected by line-by-line algorithm. These techniques have the advantage of overlooking the code layout and also detecting clones with different identifier names. On the other hand, they can identify as clones sequences of tokens that are semantically different.

Abstract Syntax Tree (AST) based - An abstract Syntax Tree is built from the source code. The algorithms then compare subtrees and marks as clones subtrees that have the same structure [BYM+98, Yan91, WSGF04, EFM09]. Like token bases techniques, AST based techniques are capable of identifying clones with different identifiers and are

impervious to layout. AST can also detect consecutive elements in the same scope (and thus ignore them). It can also detect cloned methods or other cohesive structures. The drawback is the time needed to perform the comparisons between the subtrees. The larger the system the more time is consumed in this task.

Program Dependency Graph (PDG) based -. A PDG [FOW87] is built using semantic information, like control flow, from the source code and then subgraphs that are isomorphic are marked as clones [KH01, Kri01, LCHY06]. They have all the benefits of the previous approaches, and can detect reordered statements and modifications like addition and removal of code. The problem with PDG approaches is the time consumed in the graphs comparison making it hard to scale to large systems.

Metric based - These approaches measure several metrics from code fragments. Then they compare the metrics between them and equal or similar metrics are regarded as clones [Kon97, MLM96]. Metrics can include number of lines of code, function calls or even control flow metrics. These techniques can produce many false positives, and can fail to identify fragments of methods as clones as they take the metrics for the entire method.

Table 2.2 shows a summary of the techniques in regard to four parameters. The portability parameter indicates how well a technique responds to the change of the programming language. Precision reflects the number of false positives the technique detects, while recall indicates the number of clones it detects. Scalability reflects how the technique copes if the size of the system to be analyzed increases.

Technique	Portability	Precision	Recall	Scalability
Line-based	High	100% exact clones No false positives	Low, only detects exact copies	Depends on comparison algorithms
Token-based	Medium, needs lexer transformation rules	Low, returns many false positives	High, can detect most clones	High
AST-based	Low, needs parser	High	Low	Depends how comparison is made
PDG-based	Low, needs PDG generator	High	Medium, cannot detect all clones	Low
Metrics-based	Low, needs parser/PDG generator for the metrics	Medium, returns false positives	Low, cannot detect many clones	High, metrics comparison is fast

Table 2.2: Detection Techniques Comparison (adapted from [RC07]).

2.5 Dealing with Code Clones

Code clones can either be removed or managed. But the preferred way of dealing with clones would be to avoid them in the first place.

2.5.1 Removing code clones

Code clones can be eliminated by better design [BMD+99] or refactoring [Fow99]. Refactoring consists in changing a system's code without changing its external behavior. There are also several tools, with different automation levels, to assist in clone removal [RC07]. Many of the clone removal tools use refactorings, especially Extract Method [BMD+00, FR99, HKKI04, HKI08, KH00]. Refactoring techniques that can be used to remove code clones are described next.

Extract Method

When blocks of code are similar or differ only in identifiers we can put that block inside a method and use the method instead of the code (see figure 2.1). For this to work, though, the code must be part of the same class and the clones must have consecutive statements. It can also be used when preparing for some other technique that uses a method granularity.

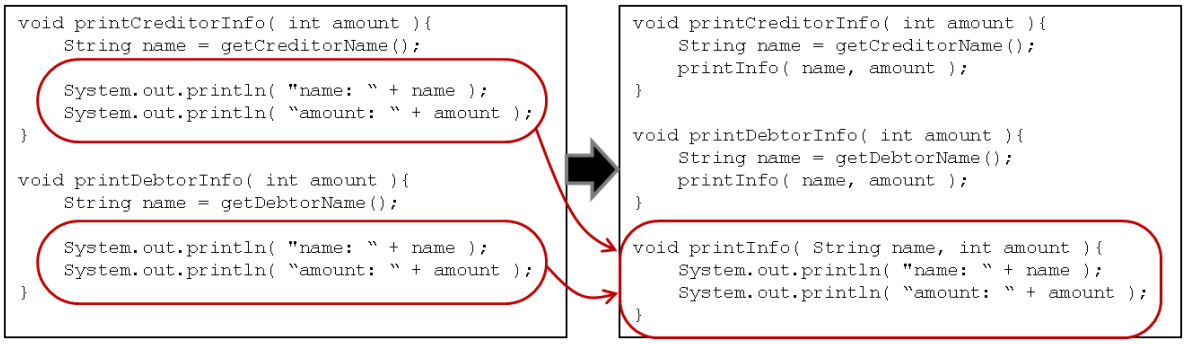

Figure 2.1: Example of Extract Method usage.

Extract Superclass

If classes duplicate structure and behavior the duplicated code is placed in a superclass and reused using inheritance (see figure 2.2). If classes already have a common superclass then Pull Up Method or Pull Up Field can be used. If classes have different superclasses them Extract Class can be used.

Pull Up Method + Pull Up Field

When classes have an identical method it can be moved to a common superclass. The same procedure is used when dealing with fields. This procedure is depicted in figure 2.3. This refactoring can be used if classes are related and share a common superclass and developers have access to that superclass. If classes do not have superclasses Extract Superclass can be used.

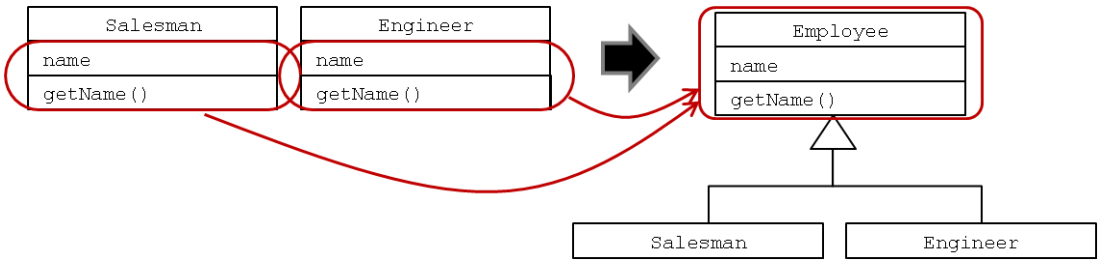

Figure 2.2: Example of EXTRACT SUPERCLASS usage.

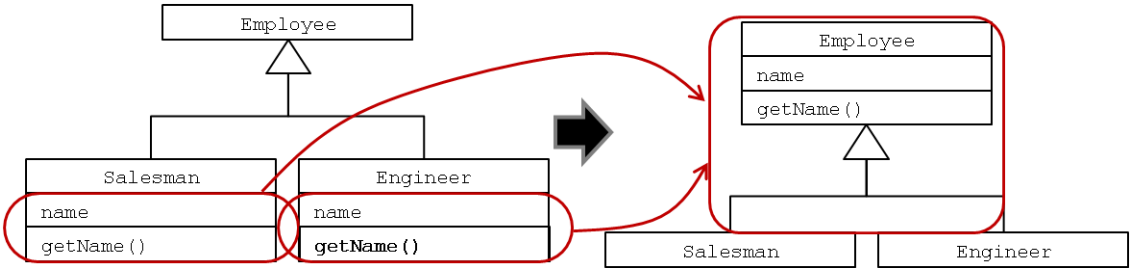

Figure 2.3: Example of PULL UP METHOD and PULL UP FIELD usage.

Extract Class

If classes share structure and behavior but have different superclasses, or inheritance is not conceptually an option, the common code is moved to a new class. The new class is then used by the original classes as shown in figure 2.4. This forces original classes to have methods that call the corresponding methods of the new class.

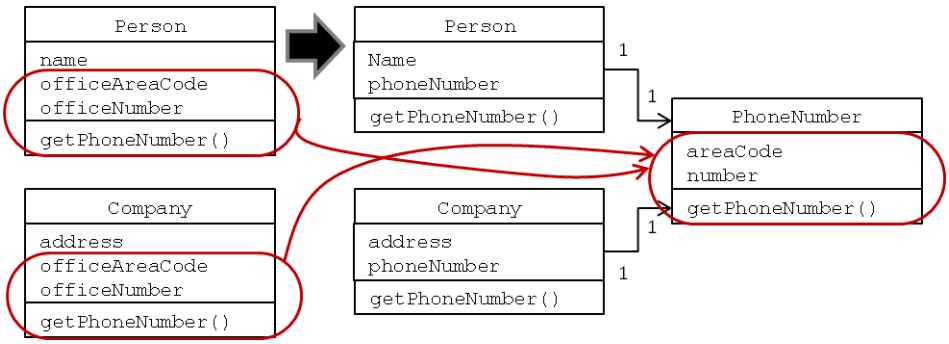

Figure 2.4: Example of Extract Class usage.

Form Template Method

Used when subclasses have a method that perform the same steps in the same order, but each step is done slightly different in each class. To overcome this, each step is placed inside a method with the same signature. The original method is modified to call the step methods in the intended order. This way the original method becomes identical in all subclasses (see figure 2.5), and PULL UP METHOD can be used.

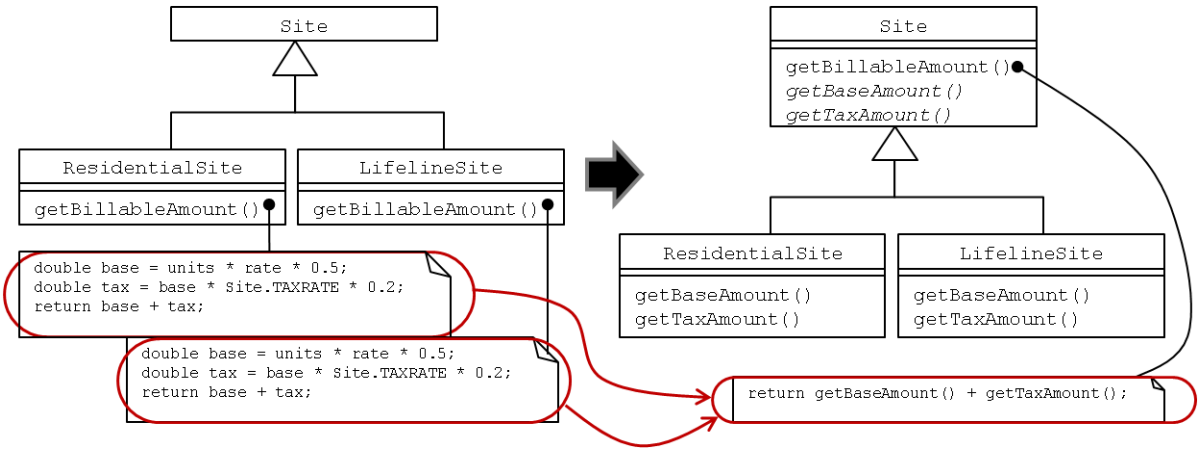

Figure 2.5: Example of Form Template Method usage.

2.5.2 Avoiding Code Clones

To avoid clones there are some available options. The clone detection tools can be used in the development process to ensure that when adding new functionalities (or simple methods) to the system they do not introduce clones. Detected clones can either be fixed or they can remain in special, and thoroughly thought of, circumstances [LPM+97].

Some clones could be avoided if a language had other composition mechanisms. In particular crosscutting concerns would benefit from the use of a composition mechanism where a class could be composed of several pieces of software. Several proposals for composing class are available, like inheritance, mixins [BC90], traits [SDNB03, DNSB06], features [AK09], aspects [KHH+01] and roles [Kri95, Rie00, Gra06].

2.5.3 Managing Code Clones

When the introduction of clones is necessary, due to performance reasons, cost, limitations, etc., they must be managed to reduce the risk of update inconsistencies.

Some clone management techniques involve the use of simultaneous editing. With the help of some tool the developer selects and links specific code regions. The developer can then edit and see how all the regions will be affected [NS03, TBG04]. This greatly reduces the time spent in updating several pieces of code.

Some automated tools keep track of the clones and of the changes made to the clones. They can notify developers whenever they change cloned code. They also support the simultaneous editing of clone regions [CH07, DER07].

2.6 Summary

Code clones have significant drawbacks in a system, ranging from program comprehensibility to a higher maintenance cost and a greater evolution effort. So much so that they are considered a major indicator that the system needs to be refactored and ways of eliminating the code clones have to be devised.

To remove clones from a system, first they must be detected. The clone detection problem lead to a vast research field and several techniques were proposed, underlining the importance of this problem. The techniques range from simple line by line comparisons to the construction of abstract syntax trees later compared for similarities. Clones can be categorized into four types, each type being more complex than the previous and harder for the detection techniques to find.

Tools to aid the removing of clones have also been proposed. The techniques used to remove the clone code, either with tools or by hand, rely mostly on the use of refactorings. We listed and explained the most used refactorings to remove clones. But these refactorings are not capable of removing all the clones, so we need a new categorization of these clones.

To tackle the clone code problem right from the development stage, and not only after the system is implemented, we need to understand why they appear in the first place. Only after this understanding can we begin to devise ways of preventing them to appear. We can also devise ways to manage the appearance and maintenance of code clones when they are required, for example, due to performance reasons.

Chapter 3

Decomposition Techniques

To show how several decomposition techniques work and their key aspects we will present two frameworks developed using three decomposition techniques. We opted to use object-oriented decomposition and some popular derivations, because it is the most used technique today. The decomposition strategies that we will address are:

- Object-Oriented;
- Aspect-Oriented.
- Roles;

From these we present, in this chapter, the OO and AOP versions. The role version is presented in its own chapter (4). We will briefly present each technique. We then discuss how the various decomposition techniques facilitate code reuse but still fall short in reducing some code replication, i.e., there are always some (sometimes large) pieces of code that are almost identical in several modules.

We will also present other approaches but not at the same level of the previous.

3.1 Sample Frameworks

In this section we present the sample frameworks that will be analyzed in the light of the three different decomposition techniques. We selected two frameworks from different domains to show how each technique deals with the problems posed in each framework. From the study we can show how code replication is present in each framework and that it does not depend on the domain. Another reason to select two frameworks was to show that there is code replication between them, even if they are from different domains.

3.1.1 Figure Handling Framework

The first framework presented is based on the JHotDraw Framework and represents a framework for dealing with technical and structured graphics applications. For the purpose of this thesis we will refer only to the figures that the user can create and manipulate. The user can create different kinds of figures such as lines, rectangles, hand drawn lines, circles, text, etc. The user can also aggregate some figures into one single figure – a group figure. The figures that the user creates are all part of a drawing. The drawing is drawn in the client part of the application's window. The user manipulates the figures by using the mouse. Every figure may have a different line color, width and style as well as different filling patterns. Every figure also has a bounding box – the smallest possible box that contains the entire figure - useful for manipulating the figure's dimension and position. Figure 3.1 shows a possible application.

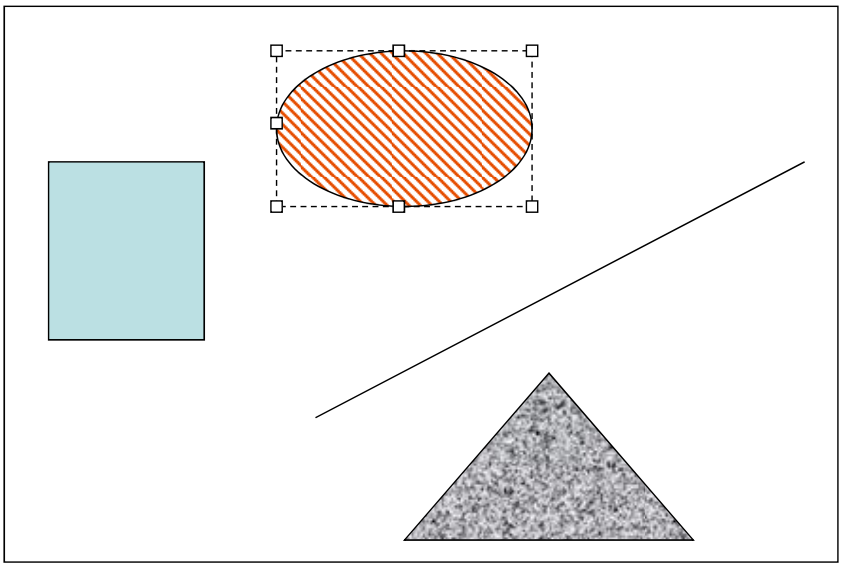

Figure 3.1: Example of an application that uses the figure framework

3.1.2 Graphical User Interface Framework

The second framework presented is based on the Java AWT/Swing frameworks and represents a framework to build graphical user interfaces (GUI). The elements present in the framework represent all the widgets (referred to as components) that usually appear in a GUI, like windows, buttons, menus, toolbars, scrollbars, etc. This is a large framework but we will focus only on some of the elements and the basic structure of the framework.

Some components may own other components. For example, a window may own several toolbars, and a toolbar may own several buttons. An example is shown in figure 3.2.

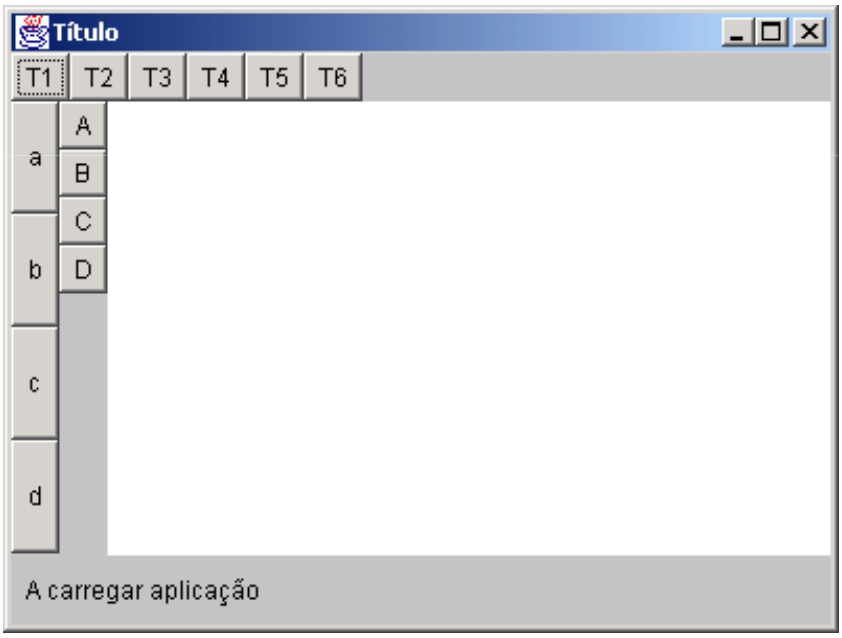

Figure 3.2: Example of a GUI created with the GUI framework.

3.2 Object-Oriented Decomposition

This section does not intend to review all the object-oriented (OO) concepts but merely present a summary of the most relevant ones.

Objects and classes are the traditional modeling concepts of object-oriented software systems. A class defines a new type of data, specifying its structure and behavior. The class defines the operations that may be used on its instances and how these operations affect the state of those instances. Ideally a class should model one, and only one, specific concept. From our sample frameworks we can use as an example the Figure class that models the concept of a figure in the Figure framework. An object is an instance of a class. In most OO based programming languages objects are the only entities that exist in run time.

Another, very important, concept associated with OO is inheritance. Inheritance is the capability that a class (subclass) has to inherit all the structure and behavior from another class (superclass). From a code reuse point of view this is a great advantage as the methods defined in a superclass are inherited by the subclass, preventing them from being duplicated in both classes. The same holds for the structure of the superclass. Inheritance is also called specialization as the subclass is, conceptually speaking, a specialization of the concept modeled by the superclass. A LineFigure class, for example, is said to be a specialization of the Figure class.

We must distinguish between single inheritance and multiple inheritance. Single inheritance occurs when a class has only one superclass. In multiple inheritance a class may have two or more superclasses. Many programming languages allow single inheritance only, because multiple inheritance also brings multiple problems, namely, name collisions and the diamond problem. Name collisions occur when a class inherits from two classes that have either fields or methods with the same name. Whenever this occurs the compiler generates an ambiguity error. The diamond problem occurs when a class inherits twice from the same class through different paths. For example: consider the classes from figure 3.3. Class D inherits from B and C and it is inheriting from A twice: from D->B->A and D->C->A. The diamond problem means that all members of A will be doubled in class D. When foo() is called on a D object the compiler cannot disambiguate between the foo() inherited from B or that inherited from C and raises an error. There are solutions to these problems but they imply knowing the inheritance structure, and that is either not always known or is not supposed to be known. Other solutions, like virtual base classes in C++, need developers of the A class to foresee this problem and prevent it when first developing the class. Over the time multiple inheritance problems have been considered to overcome its benefits, and modern languages like Java and C# don't support it.

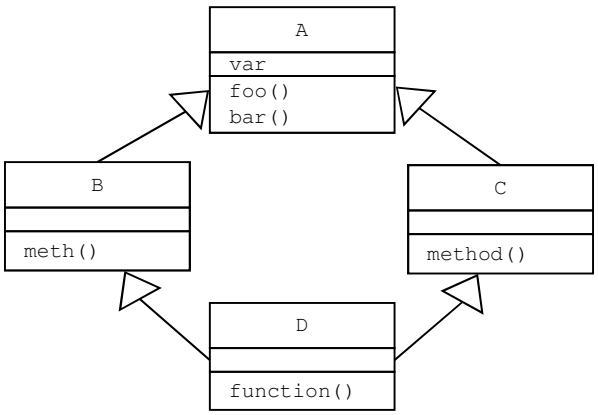

Figure 3.3: The classic diamond problem.

Another form of inheritance is interface inheritance. An interface defines a set of

methods that classes must implement to be considered implementers of that interface. Since the methods in the interface are not implemented, and interfaces are not allowed to declare any structure, except constants, it is possible to allow multiple interface inheritance without the problems associated with multiple inheritance. Hence programming languages that do not support multiple inheritance often offer support for multiple interface inheritance. It is common in interface inheritance to place an interface at the top of every hierarchy. The interface defines the interface that all classes in the hierarchy must adhere to, and then a class (usually an abstract class) implements the default behavior. Normally all the classes in the hierarchy inherit from the abstract base class to get the default behavior but they may choose to rewrite all the code from scratch and implement the interface directly. This overcomes some of the limitations of not having multiple inheritance, as implementing the interface frees the class to inherit from a class not belonging to that hierarchy. The downside is that some code replication is, probably, needed, as we will discuss in 3.2.3.

With inheritance comes polymorphism, also known as substitutability. With polymorphism an object of a more specific class may substitute an object from a more generic class. For example, whenever a Figure is expected a LineFigure can be used. This allows for a uniform treatment of objects from the class hierarchy, without knowing the real type of the objects.

Delegation also plays a very important role in reusing code in Object Oriented systems. An object instead of doing the work itself delegates the work on another object. This mechanism is shown in figure 3.4. This way the class does not have to replicate the behavior, just define to whom it delegates the work on. In the example the B class delegates the work to the A class.

The implementation of delegation depends on the programming language itself. Some languages use delegation in its purest form while others just use a simple variation that we may call forwarding. The difference between the two can be made by what is known as the self problem [Lie86]. Again, consider the example from figure 3.4: the B class delegates the work of foo to the A class. The A and B classes have a bar method each. The difference between the pure delegation and simple forwarding is done when we call foo() on a B instance. Which bar method is called from the foo method in A? Or in other words the this reference in A.foo refers to which object: the A instance or the B instance? In pure delegation this should always refer to the original object - the B instance - and thus print "b.bar". In forwarding the this reference refers to the A instance - and thus prints "a.bar".

There are authors that recommend the use of delegation instead of inheritance [KS08, SG99, JO93]. With inheritance the subclass cannot choose which members to inherit and so inherits all methods and fields even if they are useless, or even unwanted, for its purposes. With delegation the delegator does not have to inherit all the interface of the

```
class A {
  void foo() {
    this.bar();
  }
  void bar() {
    System.out.println("a.bar");
  }
}

class B {
  private A a; // the delegate

  public B(A a) {
    this.a = a;
  }
  void foo() {
    a.foo();   // call foo() on the delegate
  }
  void bar() {
    System.out.println("b.bar");
  }
}

a = new A();
b = new B(a); // defining the delegate object of b
```

Figure 3.4: Implementation of delegation.

delegatee and can choose which methods it actually uses. We can exemplify this with the common example of a stack. In Java the Stack class inherits from Vector, and so it inherits methods like insertAt or removeAt, that really do not make part of a stack interface. If delegation was used then the Stack class could implement only those methods it really wanted. This example is shown in figure 3.5.

Another problem with this implementation is that, due to polymorphism, we can use a stack whenever a Vector is used, so we could write the code

```
Vector v = new InheritanceStack();
...
v.insertAt( 5, object );
```

This would break the specifications of the Stack class as we are inserting elements into it. Looking at the code, however, it is not obvious that we are breaking the Stack because it seems that we are using a Vector.

The disadvantage of delegation is that we must define the interface of the class. This, if the interface is big, can be a tedious and error prone task. Again this can be implementation specific and depends on the delegation model of the underlying language. Some languages only require the definition of the delegatee and any method call that is undefined in the delegator is automatically searched for in the delegatee.

```
class StackUser {
  Stack s = new Stack();
  ...
  s.push(...);
  if (  s.size() ...
}
// class Stack using inheritance
public class InheritanceStack extends Vector {
  public Stack() {}
  public Object push(Object item) {
     addElement(item);
     return item;
  }
}
// class Stack using delegation
public class DelegationStack {
  protected Vector delegatee;
  public Stack() {
    delegatee = new Vector();
  }
  public Object push(Object item) {
    delegatee.addElement(item);
    return item;
  }
  public int size() {
    return delegatee.size();
  }
}
```

Figure 3.5: The impementation of a Stack class using delegation and inheritance. Adapted from [KS08]

Patterns [GHJV95] are not a concept of OO but are, nevertheless, a very important concept in the OO modeling and design. Every good software engineer knows and uses them. Most frameworks have them in their design. They allow the design reuse of a proven solution to a recurring problem.

3.2.1 Object-Oriented Figure Framework

The main concepts in the Figure framework are the figures. Therefore it is natural to create classes that model each figure. As all the figures share a common concept, and even may share code and behavior, an inheritance hierarchy is therefore created. Some figures may contain other figures, like the group figure or even the drawing (it may be considered a figure itself) so the Composite pattern [GHJV95] was used. Since the view has to draw the figures it must know when they have changed. We used the Observer pattern [GHJV95] for this purpose. Figure 3.6 shows a possible class diagram of this framework.

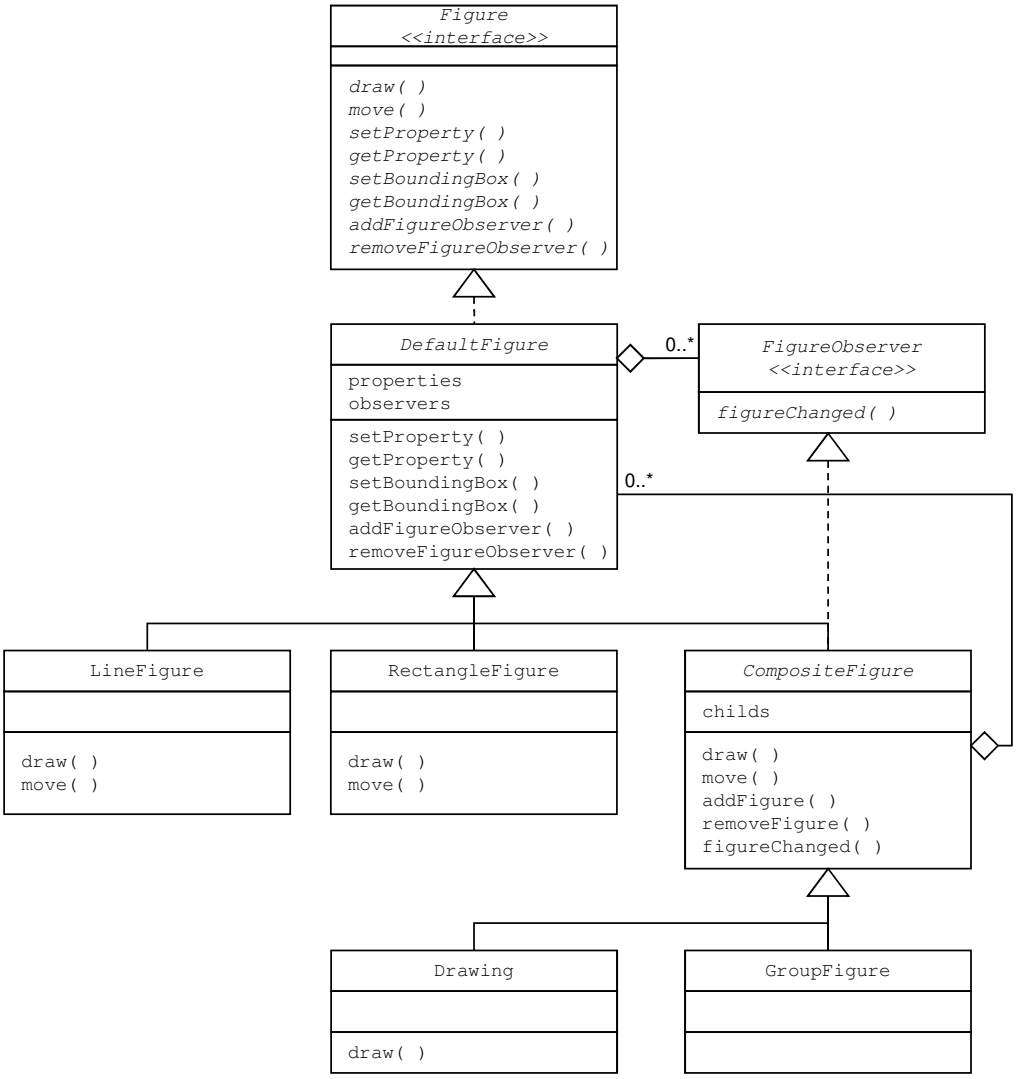

Figure 3.6: Class diagram of the Figure framework

3.2.2 Object-Oriented GUI Framework

The main concepts in the GUI framework are the components, therefore it is natural to create classes that model each component. As all the components share a common concept, and even may share code and behavior, an inheritance hierarchy is created. Some components have other components, as the toolbar or even the window so the Composite pattern was again used. Since the programmers may be interested to know when the mouse is hovering (or clicking) a component the Observer pattern is used. Other instances of the Observer pattern are used as the programmers may be interested in other user's actions as pressing a key or if a component has lost/gained focus, etc. Figure 3.7 shows a possible class diagram of this framework.

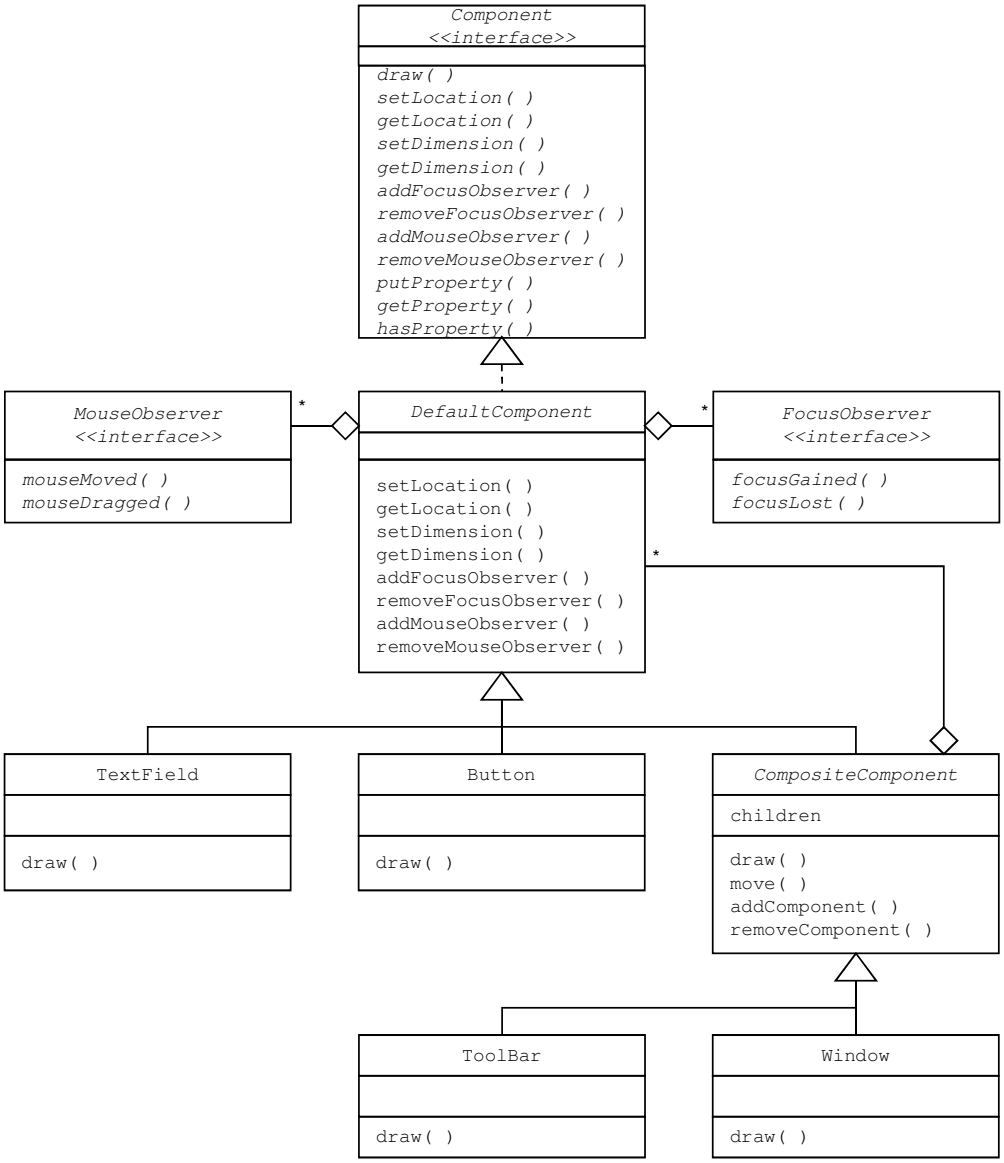

Figure 3.7: Class diagram of the GUI framework

3.2.3 Code Replication in the Object-Oriented Solution

Because we did not use multiple inheritance we used the interface approach on both solutions. As already mentioned this may lead to code replication. Suppose we pretend to introduce a Picture class that supports a number of known picture formats. Most likely that class already exists so we wish to use it in our application but as it is not part of the framework it does not comply with the interface of the Figure hierarchy.

With multiple inheritance we could define a new class, PictureFigure, inheriting from the Picture class and the Figure class, thus reusing the Picture and Figure code making it part of the hierarchy. This solution implies some adaptation code, but we can assume that a great part of the Figure code is reused.

With the interface approach we may choose to make the PictureFigure class inherit from the Picture class and implement the Figure interface. As interfaces do not provide code to its implementers we are forced to replicate the DefaultFigure code that applies to this class. This situation is depicted in Figure 3.8. One example of replicated code would be the implementation of the Observer pattern. Another solution would be to use the Adapter pattern [GHJV95].

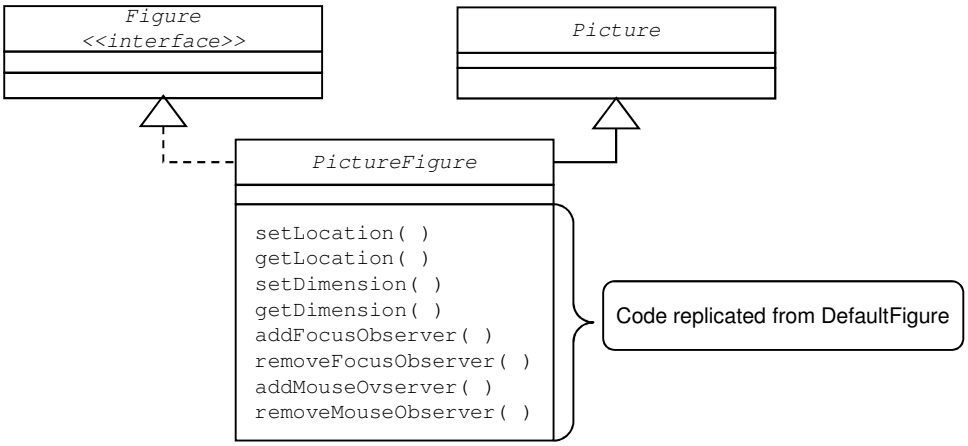

Figure 3.8: Code replication originated by not using multiple inheritance.

The same consideration applies to the component hierarchy if we wish to use a complex component that is not part of the Component hierarchy: to use the previous solution we must replicate most of the DefaultComponent's code.

Another source for code replication is the use of the patterns Composite and Observer. Not that the patterns forces us to replicate code or are bad solutions, quite the contrary. Since these patterns proved to be very good solutions they are widely used. But, in these patterns case, the code for each instance of the pattern is very similar and so the programmer tends to use the same mechanisms over and over.

The composite pattern uses a collection, to keep track of the children, and offers two methods: one for adding children and another for removing children. Most of the actions are processed in the same way: iterate through the collection and for each child the corresponding method is called. This is shown in figure 3.9. The Composite pattern in the component framework is very similar and is not shown.

The observer pattern uses a collection of observers to keep track of the registered observers and usually offers, at least, two methods: one for adding observers and another for removing observers. When an event occurs and the observers must be notified a notification method is called. This method iterates through the collection and calls the observer's update method. Figure 3.10 shows an extract of the observer code from the component framework where the similarities are evident between the focus observers and

```
public class CompositeFigure {
    private Collection<Figure> childs;
    public void addFigure( Figure f ){
        childs.add( f );
    }
    public void removeFigure( Figure f ){
        childs.remove( f );
    }
    public void move( int dx, int dy ){
        Iterator<Figure> iter = childs.iterator();
        while( iter.hasNext() ){
            iter.next().move( dx, dy );
        }
    }
    public void draw( ){
        Iterator<Figure> iter = childs.iterator();
        while( iter.hasNext() ){
            iter.next().draw( );
        }
    }
    // ...
}
```

Figure 3.9: Code from the composite pattern in the Figure Framework.

the mouse observers in the add, remove and notify methods.

We can also verify that a lot of the code from the observer and the composite patterns is very much the same, especially the use of a collection and the add and remove methods, but we have no means to avoid replicating it.

```
public class DefaultComponent implements Component {
  Collection<MouseObserver> mouseObs;   // collections to keep
  Collection<FocusObserver> focusObs;   // the observers

  public void addFocusObserver(FocusObserver fo) {
    focusObs.add( fo );
  }
  public void addMouseObserver(MouseObserver mo) {
    mouseObs.add( mo );
  }
  public void removeFocusObserver(FocusObserver fo) {
    focusObs.remove( fo );
  }
  public void removeMouseObserver(MouseObserver mo) {
    mouseObs.remove( mo );
  }
  private void notifyFocusLost( ){
    Iterator<FocusObserver> iter = focusObs.iterator();
    while( iter.hasNext() ){
      iter.next().focusLost( this );
    }
  }
  private void notifyFocusGained( ){
    Iterator<FocusObserver> iter = focusObs.iterator();
    while( iter.hasNext() ){
      iter.next().focusGained( this );
    }
  }
  private void notifyMouseMoved( ){
    Iterator<MouseObserver> iter = mouseObs.iterator();
    while( iter.hasNext() ){
      iter.next().mouseMoved( this );
    }
  }
  private void notifyMouseDragged( ){
    Iterator<MouseObserver> iter = mouseObs.iterator();
    while( iter.hasNext() ){
      iter.next().mouseDragged( this );
    }
  }
  // ...
}
```

Figure 3.10: Code from the two instances of the observer pattern in the Component Framework. The similarities between their add, remove and notify methods are clear.

3.3 Aspect-Oriented Programming

In the object-oriented decomposition programs are decomposed in terms of objects and classes. But there are some properties that often cannot be assigned to individual objects as memory management, synchronization, scheduling, persistence, communication, etc. As these properties affect several classes this leads to the spreading of the code that deals

with them. In Aspect-Oriented Programming (AOP) terms [KLM+97] such properties are referred to as concerns and as they are spread over several units they are referred as crosscutting concerns. The fact that the crosscutting code is spread over several classes and that code is not related to the main concerns of the class itself, but is often mixed with the main concern code, leads to what is called the code-tangling problem.

AOP also decomposes the system, but does so in two different dimensions: components and aspects. Components derive from those properties that can be pinpointed to a class or procedure. An aspect is a property that cannot be assigned to any particular class or procedure. This separation of components and aspects is the goal of AOP, allowing the programmer to handle them separately, and giving the mechanisms to compose them together to produce the final system.

In a number of AOP languages aspects include pointcuts and advices. The pointcut is a predicate over join points that defines at which join point, or sets of joint points, the aspect takes effect. The advice itself specifies what is to be executed at that particular joint point. The advice can then insert code before, after or instead of (around in some aspect languages) the referred join point, in most aspect languages. Like in the case of the join point, types of pointcuts and advices are language specific.

Aspect-oriented programming has become, in the last decade, very popular. Even though its focus is on the crosscutting concerns, we want to explore if it can be used to reduce code replication, as crosscutting concerns tend to have replicated code [BvDvET05].

AOP is related to OO but deviates somewhat from this approach and requires learning many new concepts. And while the modularization of crosscutting concerns is the flagship of AOP several authors disagree [Prz11, Ste06]. Concepts like pointcuts and advices are not easy to understand. Furthermore the effects of these constructs are more unpredictable than any OO concept. A particular one is the fragile pointcut [KS04]. This problem arises when simple changes made to a method's code make a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the necessary advice. Thus simple changes in the class code can have unsought effects [KAB07].

The obliviousness feature of AOP [FF00] means that a class is aspect unaware so aspects can be plugged or unplugged as needed. But it also introduces problems in comprehensibility [GSS+06]. To fully understand the system we must not only know the classes but also have to know the aspects that affect each class. This is a major drawback when maintaining a system, since the dependencies aren't always explicit and there isn't an explicit interface between both parts.

3.3.1 AOP Figure Framework

One hint that crosscutting concerns are present is multiple interface inheritance [MF06]. If a class implements several interfaces it is likely that it performs several roles, some of them superimposed, because the class participates in a collaboration [HK02]. This is the case with both our sample frameworks, if we implement an interface for each concerns each class deals with.

Hannemann and Kickzales propose in [HK02] some implementations of the GoF patterns [GHJV95]. We will adapt those implementations and apply them to the OO version of the example framework. One of the patterns used is the Observer. In the proposed solution the observer code is removed from the Figure (actually implemented in the DefaultFigure) and placed in an aspect. Figure 3.11 shows the code for the aspect as proposed in [HK02]. Figure 3.11 also shows the concrete FigureObserver aspect used in the framework.

The proposed solution in [HK02] for the Composite aspect, shown in figure 3.12 is somewhat awkward to use. Worse still: some methods that are indeed part of the Figure concept must be also moved to the aspect. It is important to say that deciding if a given method should be or not part of the class main concern is a developer decision. In our opinion some of the methods that are required to move to the aspect are indeed part of the core concept. That is the case with the setProperty method, for example. Thus we believe that the OO solution is better so we maintained it.

3.3.2 AOP GUI Framework

The AOP solution for the GUI framework is analogous to the Figure framework. The ObserverProtocol aspect is reused and a MouseObserver and FocusObserver are added. The composite solution is still too cumbersome to use, so we again opted to use the OO version.

3.3.3 Code Replication in the AOP solution

The AOP solution does reduce code replication. The code for the observer pattern is now placed inside an aspect that is reused by both frameworks.

To solve the Picture class problem, however, AOP does not offer a generic solution, even though some workarounds could be arranged.

But some code replication is still visible even in the AOP solution. A closer look at the code from the Observer and Composite generic aspects (see Figure 3.11 and Figure 3.12, respectively) reveals that both have a Collection for the observers/children and the addX and removeX methods.

```
public abstract aspect ObserverProtocol {
  protected interface Subject { }
  protected interface Observer { }
  private WeakHashMap perSubjectObservers;
  protected List getObservers(Subject s) {
    if (perSubjectObservers == null) {
      perSubjectObservers = new WeakHashMap();
    }
    List observers = (List)perSubjectObservers.get(s);
    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(s, observers);
    }
    return observers;
  }
  public void addObserver(Subject s,Observer o){
    getObservers(s).add(o);
  }
  public void removeObserver(Subject s,Observer o){
    getObservers(s).remove(o);
  }

  abstract protected pointcut subjectChange(Subject s);
  abstract protected void updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) {
      updateObserver(s, ((Observer)iter.next()));
    }
  }
}

public aspect FigureObserver extends ObserverProtocol {
  declare parents: LineFigure implements Subject;
  declare parents: RectangleFigure implements Subject;
  // add other figures
  declare parents: View implements Observer;

  protected pointcut subjectChange(Subject s):
    (call(void LineFigure.setProperty(String, Object )) ||
     call(void LineFigure.setBoundingBox( int , int , int , int ))
     call(void LineFigure.move( int , int ))
     // do the same for the other figures ...
    ) && target(s);
  protected void updateObserver(Subject s, Observer o) {
    ((View)o).updateView( );
  }
}
```

Figure 3.11: Code for the generalized observer aspect as proposed by Hannemann and Kickzales, and the concrete FigureObserver aspect for the figure framework.

```
public abstract aspect CompositeProtocol {
    public    interface Component {}
    protected interface Composite extends Component {}
    protected interface Leaf      extends Component {}

    private WeakHashMap perComponentChildren = new WeakHashMap();

    private Vector getChildren(Component s) {
        Vector children = (Vector)perComponentChildren.get(s);
        if ( children == null ) {
            children = new Vector();
            perComponentChildren.put(s, children);
        }
        return children;
    }
    public void    addChild(Composite composite, Component component) {
        getChildren(composite).add(component);
    }
    public void removeChild(Composite composite, Component component) {
        getChildren(composite).remove(component);
    }
    public Enumeration getAllChildren(Component c)  {
        return getChildren(c).elements();
    }
}
```

Figure 3.12: Code excerpt from the generalized composite aspect as proposed by Hannemann and Kickzales.

Furthermore the Aspect solution for the Observer is not modular. Whenever we add a Figure class we must change the aspect. Further changes to the aspect are needed if we alter the Figure interface. For each inserted method that may change the figure we must add it to the aspect, for each method removed we must remove it from the aspect, for each method renamed we must rename it in the aspect. Even if we don't rename a method but it no longer changes the figure it must be removed from the aspect. This means that the hierarchy builder must be aware of the aspect and change it. This is against the principle of obliviousness that is advocated for AOP [FF00]. Steimann is one of many that say AOP is not modular at all and even goes against modularity [Ste06]. We agree with him in this case.

The fact that the observer aspect must know the type of the observer is also a major flaw. In the code from figure 3.11 we can see that the observer aspect assumes that observers are of type View. If we add another type of observer this will break the aspect code. The only solution to this problem is to make all possible observers have the same update method (like in the original Observer pattern). But this means that observers must be aware of the pattern and then AOP solution rolls back to the OO solution.

Another drawback from the observer aspect is the lack of fine granularity. The join

point (in most languages) must be a method (setProperty, for example) but there are operations that may or may not change the subject. For example if we set a property with the same value a change does not occur, but the aspect still notifies every observer with, possibly large, costs in performance.

Java style observers, listeners, also associate, with every change, an Event class that holds important information from the change context. With AOP this context information is lost because the context is element specific. The aspect could compute the context, but that would need code replication and performance cost again. This is especially the case with the GUI framework where a lot of events may be generated and the context information is very important. The mouse observers, for example, may inform observers of the relative position of the mouse at the time of an event, along with other information. The aspect must query the component for this information because it may not be available elsewhere. This requires the component to provide an interface to supply this information to the aspect if it does not supply it already. Thus an invasive modification in the component is required which, again, goes against some AOP principles.

Yet another change that is required is, again from Java listeners, the possibility to use different notifying mechanisms for each change (mouseMoved or mouseDragged, for example). If a single method is capable of firing several events then the AOP solution is inefficient because it will signal all events, even if only one occurs, or it must compute, again, which event took place.

All things considered we believe that the AOP solution is not generally applicable and, for the sample frameworks, especially the GUI framework where many observers are expected, each of which may have several notification methods, it seems not to be a good solution and it is poorer than the OO version.

3.4 Traits

Traits are units of code reuse and a class can be constructed using several traits [DNSB06, SDNB03]. Traits have a flattening property: a class can be seen indifferently as a collection of methods or as composed by traits. The fact that the class can be seen as a whole promotes understanding and the fact that it can be composed promotes reuse. In Traits a class can be constructed by using inheritance and by adding traits. The class must supply all state variables and glue code. The glue code is the set of methods that the trait requires the class to provide (for example, accessor methods for the state variables). Thus a class can be decomposed into a set of coherent features and the glue code connects the various features together. According to [DNSB06], Traits have the following properties:

- A trait provides methods that implement behavior.

- A trait requires a set of methods that serve as parameters for the provided behavior.
- Traits do not specify state variables, and methods provided by traits never access state variables.
- Classes and traits can be composed from traits.
- The composition order of traits is irrelevant.
- Conflicting methods must be explicitly resolved.
- Trait composition does not affect the semantics of a class: the meaning of the class is the same as it would be if all of the methods obtained from the trait(s) were defined directly in the class.
- Similarly, trait composition does not affect the semantics of a trait.

A class can redefine its superclass's and its trait's methods. Conflicts arise when unrelated traits have methods with the same signature. The conflict must be solved explicitly by redefining the conflicting method in the class. The conflict is thus resolved locally. To access the conflicting methods Traits support aliases. It works by giving an alias to a method so it can be used in the class without trouble. To prevent conflicts from occurring in the first place traits also support the exclusion of methods.

Some attempts to bring traits into Java-like languages have been made [QB04, SD05]. The following traits examples are presented using the Chai syntax [SD05] and derive from the examples shown in [SD05]. Figure 3.13 shows trait declaration in Chai and its use by classes. We can see requirement of methods in the TEmptyCircle: it offers a draw method and requires the class to provide the drawPoint and getRadius, with the specified signature. The same methods are also required by TFilledCircle. The code also shows a Circle class, representing a circle, and two subclasses composed by traits and that inherit from Circle. The ScreenEmptyCircle class is an empty circle that can be drawn in the Screen, so it uses TEmptyCircle and TScreenShape. The methods required by TEmptyCircle are supplied by Circle and TScreenShape, so ScreenEmptyCircle does not need to provide them itself. PrintedFilledCircle is a filled circle than can be printed in a printer, so it inherits from Circle and uses TFilledCircle and TPrintedShape. TFilledCircle required methods are supplied by Circle and TPrintedShape. In the TPrintedShaped case the class needed to alias the trait method for the required name.

```
class Circle {
  int radius;
  int getRadius() { ... }
}
trait TEmptyCircle {
  requires {  void drawPoint(int x, int y);
              int getRadius();  }
  void draw() { ... }
}
trait TFilledCircle {
  requires {  void drawPoint(int x, int y);
              int getRadius();  }
  void draw() { ... }
}
trait TScreenShape {
  void drawPoint(int x, int y) {...}
}
trait TPrintedShape {
  void printPoint(int x, int y){...}
}
class ScreenEmptyCircle extends Circle uses TEmptyCircle,TScreenShape {
}
class PrintedFilledCircle  extends Circle uses TFilledCircle,
                                               TPrintedShape {
    alias { void printPoint(int x, int y) from TPrintedShape as
            void drawPoint(int x, int y)  }
}
```

Figure 3.13: Trait example, adapted from [SD05].

3.5 Feature-Oriented Programming

The decomposition strategy of Feature-Oriented Programming (FOP) is to decompose the system into features [AK09]. A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. Features are the main abstractions in FOP during design and implementation. A distinguishing property of FOP is that it aims at a one-to-one mapping between the representations of features across all phases of the software life cycle. That is, features specified during the analysis phase can be traced through design and implementation. Features reflect user requirements and incrementally refine each other. FOP relies on a step-wise refinement of applications by adding new features or refining existing ones.

Because features are the distinguishing product characteristics of Software Product Lines (SPL), FOP is mainly used for SPL and program generators. To compose a system we just state which features it has. The composition is made automatically with tool support, like AHEAD [BSR04]. AHEAD uses several tools for composing the code and extra files for configuring the composition step.

AHEAD can be used to compose classes. For example, we can develop a class that

defines the basic behavior of a class, indistinguishable from a normal Java class, except that it has a feature keyword indicating to which feature it is associated to (see figure 3.14). We can then construct several refinements to that class. Each refinement must indicate the associated feature and the class it refines (see figure 3.15). One technique used to implement features is Mixin layers [SB02]. The rationale is that a feature often appears in several classes that collaborate. Each mixin layer contains the code for the role(s) that each class plays in a given feature and composes them in a static component.

```
feature Single;

class List {
  Node head;
  void push( Node n ){
    n.next = head;
    head = n;
  }
}
class Node {
  Node next;
}
```

Figure 3.14: Class definitions

```
feature Reverse;

refines class List {
  Node tail;
  void hsup( Node n ){
    n.prev = tail;
    tail = n;
  }
}
class Node {
  Node prev;
}
```

Figure 3.15: Class refinements

3.6 Multiple Dimension Separation of Concerns

Multi-dimensional separation of concerns (MDSOC) allows developers to encapsulate overlapping, interacting and crosscutting concerns, including features, aspects, variants, roles, business rules, components, frameworks, etc., simultaneously [DBB+03]. One does not have to choose between a data decomposition and a feature decomposition, since both can coexist, and each can be used when appropriate. All concerns are first-class components that can be integrated flexibly. MDSOC is a natural evolution of subject programming [HO93].

MDSOC claims that crosscutting concerns are the consequences of the "tyranny of the dominant decomposition" [TOHS99]. A key reason is that one needs different decompositions according to different concerns at different times, but most languages and modularization approaches support only one "dominant" kind of modularization [OT00].

In MDSOC concerns are placed in hyperslices that are composed together in hypermodules following a set of composing rules. Hyperslices may be used by many hypermodules. Hypermodules may be reused and can contain other hypermodules.

In the figure framework a figure could be decomposed in several hyperslices, one for each concern. For example we could have the properties hyperslice, the observer hyperslice,

etc. The Figure hypermodule would compose the several hyperslices. Extensions to the application can be made by adding new hyperslices and composing them in a new hypermodule. For example, if we wish to save the figures in an SVG or any other format we need to add a hyperslice for each format defining how each figure should be saved in that format. In an OO version we would need to add a saveXFormat method in each class or use a Visitor pattern [GHJV95]. A further advantage of hyperslices is that we can mix-and-match them. This means we could add/remove the support for one format simply by including/not including the respective hyperslice.

Nevertheless code replication is still found between hyperslices. Several hyperslices that could implement observers would have code replication. And as MDSOC add another layer(s) to existing techniques if we can find a way of reducing code replication then MDSOC will also gain.

3.7 Other Approaches

Jiazzi [MFH01] is based on Units [FF98] and aims at building systems out of reusable components integrated with the language. Jiazzi has two types of units: Atoms (composed of java classes) and Compounds (composed of atoms or other compounds). Jiazzi supports the addition of features to classes without editing their source code.

Open classes as used by MultiJava [CLCM00, CMLC06] allow external methods to be added to a class, without changing the class. They also support multi method dispatching. So they are used to extend a class interface like our approach but from a different perspective: we focus on constructing the class and open classes in adding methods to existing classes.

Caesar [MO03] also uses aspect technology to modularize crosscutting concerns and enhance the reuse of aspects leading to a greater reduction of repeated code. Caesar uses an Aspect Collaboration Interface that decouples aspects binding and implementations by defining them in a separated module.

These approaches deal with the extensibility of classes. Our work's goal is not to extend existing classes but to reuse code that otherwise would be replicated in several classes. However these techniques reduce the amount of code that still is duplicated so we included them.

In Classboxes [BDN05] classes are defined within a kind of module, or unit of scoping. In each classbox we can define classes but can also import classes from other classboxes or refine other classes. Refinements may consist in adding or redefining new behavior or state from an imported class. Since these refinements are only visible within the classbox or classboxes that import from it, existing clients of the refined class are not affected.

Virtual classes [EOC06] are used by Caeser. A class can define nested classes. Nested classes are the virtual classes and they can be redefined by subclasses of the enclosing class. Each virtual class has therefore an enclosing object - the object of the container class or one of its subclasses. At run time, the inner class to use depends on the type of the outer object.

3.8 Summary

There are several approaches to composing or extending classes. We showed how some of them are used by applying them to two simple frameworks. This simple exercise showed that the decomposition capabilities of these solutions are able to produce a good solution for both frameworks. On the other hand it also showed that these solutions still suffer from code replication. In all the approaches we identified replicated code that no technique offered by the decomposition approaches could mitigate.

The conclusion we can draw from this chapter's discussion is that the existing decomposition techniques could be improved. This is the motivation for our work.

Chapter 4

Roles

Object-oriented decomposition assumes that a given concept may be modeled by a single entity, such as a class. This one to one mapping from concepts to classes is, however, too simplistic. Classes do not do all the work themselves and must collaborate with other classes. In a collaboration each class plays a certain role that falls somewhat outside the main concept of the class. For example the Figure class from the example frameworks (see section 3.1) plays the child role in the Composite pattern. This role is clearly not the class main concern and does not fit well with the idea of a figure. This means that a class may, in fact, play several roles, just because it must cooperate with other classes. Furthermore, objects from the same class may play different roles while interacting with each other. One such example is the CompositeFigure that may also play the child role in the composite pattern.

In order to better model a specific concept we need the notion of a role. The research of roles in the object-oriented area is extensive. But the definitions, modeling ways, examples and targets are often different [Ste00][Gra06].

In this chapter we discuss the various natures of roles and how can we model a system using roles, giving as an example the modeling of the frameworks described in section 3.1.

4.1 What are roles?

Currently it is hard to put forward a definition of a role, because there are too many. The role concept is considered to have been first introduced in the work of Bachman and Daya in [BD77] and [Bac80]. They stated in [BD77] that

> *most conventional file records and relational file n-tuples are role-oriented. These files typically deal with employees, customers, patients, or students, all of which are role types. This role orientation is in contrast with the integrated database theory, which has taught that each record should represent all aspects of some entity in the real world. This difference in viewpoint has caused a great deal of confusion. The reason for the confusion is understood when it is realized that neither the roles of the real world nor the entities of the real world are a subset of the other.*

Sowa [Sow84] introduced a distinction between natural type and role type. Natural types are related to the essence of an entity (today considered a class) and roles types are the characteristics that depend on an accidental relationship to other entities. This is a distinction that is still used today by many role models. Guarino [Gua92] developed Sowa's work further. To Guarino to be considered a role it is required that its individuals stand in relation to other individuals, and that they can enter and leave a extent of the concept without losing their identity. A natural type is characterized by semantic rigidity and lack of foundation, i. e. , an individual of a natural type cannot drop its type without losing its identity. Furthermore an individual of a natural type is not requested to stand in relationship to others. For example, A person is a natural type because a person individual will always be (and have been) a person, and being a person is independent of any relationship. A student is a role since to be a student it must be enrolled in a school and leaving school does not lead to a loss of identity.

Roles surveys are not very common and we can identify three attempts by Kniesel [Kni96], Steimann [Ste00] and Kappel et al [KR98]. An overview and comparison of the tree surveys can be found in [Gra06]. Neither presented a definite definition of what roles are. We present here the list of features proposed by Steimann. It is noteworthy that some of the features are conflicting and it is possible that a definition of roles does not integrate all of them (most do not). This is not meant to be a definition of what a role is, but a collection of what various authors assumed in their definitions.

- 1. *A role comes with its own properties and behavior.* This suggests that roles are types.
- 2. *Roles depend on relationships.* Many follow [Sow84] and Guarino [Gua92] work and consider that roles are meaningful only in the context of a relationship.

- 3. *An object may play different roles simultaneously.* This is one of the most accepted properties of roles concepts.
- 4. *An object may play the same role several times, simultaneously.* For example a student may be enrolled in different schools (a university, a language school and a music school), or an employee having two jobs.
- 5. *An object may acquire and abandon roles dynamically.* A person may become an employee at a given time then quit it later.
- 6. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* A person may only be a teaching assistant after becoming a student.
- 7. *Objects of unrelated types can play the same role.* This is not acknowledged by all authors, but complements items 3 and 4.
- 8. *Roles can play roles.* For example a teacher is an employee of a school.
- 9. *A role can be transferred from one object to another.* For example the role of club president may be transferred from a club member to another club member.
- 10. *The state of an object can be role-specific.* The state of an object may vary depending on the role in which it is being addressed.
- 11. *Features of an object can be role-specific.* Different roles may declare the same features but realize them in different ways.
- 12. *Roles restrict access.* When addressed in a certain role, features of the object itself may remain invisible.
- 13. *Different roles may share structure and behavior.* Roles can inherit from one another, or they can rely on the features of the object using delegation.
- 14. *An object and its roles share identity.* An object and its roles are viewed as the same entity.
- 15. *An object and its roles have different identities.*

Roles have been mainly used to express dynamic situations where the roles are attached to objects in order to provide new features or override the object's default behavior. The previous list enforces this view. There are works that proposed languages that support roles as a primitive type, such as Clovers [SZ89], Fibonnaci [AGO95], DOOR [WCL97], etc. There also a number of extensions to existing languages like Objects Teams [Her05], Chameleon [GB02], Powerjava [BSI07], etc.

Roles also have been used as a way to model the behavior of a class in a system. This is the primary use of roles in the OORam Method [RWL96] and in the work of Riehle [Rie00]. This is the use that we will make of roles. In our work we assume the definition of role by Riehle [Rie00]:

> *"a role is an observable behavioral aspect of an object".*

Riehle also defines a role type as something

> *"that defines the behavior of a role an object may play. It defines the operations and the state of the role, as well as the associated semantics".*

It should be clear though that Riehle only used roles as a modeling construct, not as a programming primitive. Actually, in the works of Rielhe and Reenskaug roles were used in modeling only. We intend to extend their work by using roles in the implementation phase also.

4.2 Modeling with Roles

To capture the complexities of the world abstractions are created. Abstractions try to model a concept present in the real world. Every language provides their own abstraction mechanisms. In OO languages usually the abstraction mechanism is the class. A concept can be specialized to a more specific concept. An animal can be specialized to a mammal or a reptile, or even to a quadruped or a biped, depending on the focus of the analysis. This specialization is, in OO systems, generally modeled by inheritance. Aggregating other concepts can also compose a concept. These are known as the "is-a" or "has-a" hierarchies [Boo95].

But as an approximation of the real world OO modeling techniques cannot capture all the dynamics present. OO Systems are founded on the Aristotelian view of the world with ideas (classes) and phenomena (objects). Each phenomenon is a manifestation of an idea: a particular chair is a manifestation of the idea of a chair. Ideas and phenomena do not exist with one to one correspondence, though. A phenomena can be classified in several different ideas: a river may be viewed as a food resource by a fisherman, a living place by a fish, a transport route for boats, etc. In programming terms it is not possible to accommodate all possible views of a phenomena with a single idea as we cannot foresee all its uses. Thus objects need to evolve overtime.

Steimann [Ste00] states that Lodwick was the first to break with the Aristotelian vision according to which the nouns of a language govern its structure and meaning. As an example the act of murder has the roles "murderer" and "murdered". These names define

the individuals involved in the murder context. Outside this context the individuals have their own proper names. This calls for another modeling construct than classes. One that can model the roles some phenomena plays when in a given context. One such construct are roles.

Roles represent the behavior of an object within a specific object collaboration task. As we have seen, objects behave in different ways when acting in different contexts. Therefore, in each context the object plays a different role. This introduces the notion of multiple views. The role is determined by the view the client holds on the object that plays the role. The view is a set of the properties of the object, modeled by a set of methods. Other objects in the collaboration can access the selected set of methods. Furthermore views can change dynamically. This means that an object's set of methods may have additive and subtractive properties [Kri95]. Thus, roles allow objects to evolve over time.

If roles allow such evolution of objects a question that may occur is: are classes a superfluous concept or do roles need classes? An argument for classes and roles is that classes represent what is static and roles describe what is dynamic. A class defines an entity, while a role only refines an entity in a certain context. Another argument is that with both concepts separated we can have inheritance hierarchies of both classes and roles, as shown in figure 4.1.

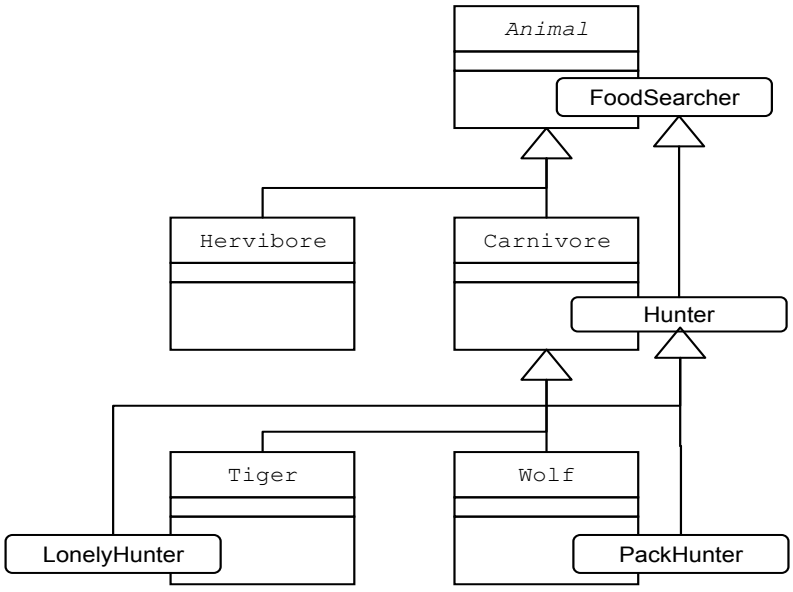

Figure 4.1: An example of inheritance hierarchies of both roles and classes.

Since both roles and classes contain properties and methods a class may be reduced to a mere container of roles - an entity whose sole purpose is to be extended by roles. If this is the case then roles can be promoted to classes as they are the only concept that contains properties.

Some languages took the approach of unifying roles and classes. Fibonacci [AGO95] used the concept of null-object because that allows the writing of roles for a more abstract entity than a concrete class. Pernicci [Per90] define a class as consisting in one to many role descriptions, where the first role description is equivalent to a "normal" class.

4.2.1 Advantages of Role Modeling

Role modeling has several advantages in system comprehension, reuse, development and documentation [Rie00]. Describing a class as a set of roles helps to separate the various ways in which a class is used. Documentation can be done in these terms as well. This helps clients to focus on whichever aspect they are interested in, providing a better understand and use of the class. Designing the class can also be done in role terms, thus developers are able to focus only on one aspect of the class. This enables independent development of a class with all its benefits in terms of reduced development time and complexity.

Class relationships are reduced to role relationships. Since roles focus on a particular view of a class we need not to understand the player in its whole. This eases the understanding and development of these relationships. Whenever needed the broader perspective can also be used. Role modeling allows shifting between role level and class level without any information loss.

Role modeling also allows for better understanding using previous experiences. When a developer knows how to use roles that have a relationship in a system, then when he encounters different roles with similar relationships the past experience will allow a quicker understanding. One such example is the use of the Observer pattern. When experienced with a FigureSubject and how it works with a FigureObserver to use another instance of the pattern is much simpler and straightforward.

Roles can also be used to model crosscutting concerns. Because a role is a smaller composition unit than a class we can put the crosscutting concern in a role, or a set of roles, and the classes that have the crosscutting concern play those roles. Any changes to the crosscutting concern are limited to the roles thus greatly improving maintenance and reducing change propagation.

4.2.2 Role Figure Framework

We can modify the Figure framework by introducing the roles that Figures play. It is not the figure's main concern to act like a Subject but it has that role superimposed on it. That pattern defines two roles: the subject role and the observer role. A Figure plays the subject role and the view plays the observer role. With roles we are able to extract those

concerns from the class and reduce code scattering. Furthermore, those roles are reusable whenever we need a class to address those concerns, even if it is not a Figure.

We can also argue that managing properties is not the figure's main concern. Thus we can model that behavior using a PropertyProvider role. Another used pattern was the Composite pattern. This pattern defines two roles: the parent role and the child role. CompositeFigure plays the parent role and every Figure plays the child role. The child role does not impose a specific interface nor semantic, so it is considered a no-semantic role type [Rie00].

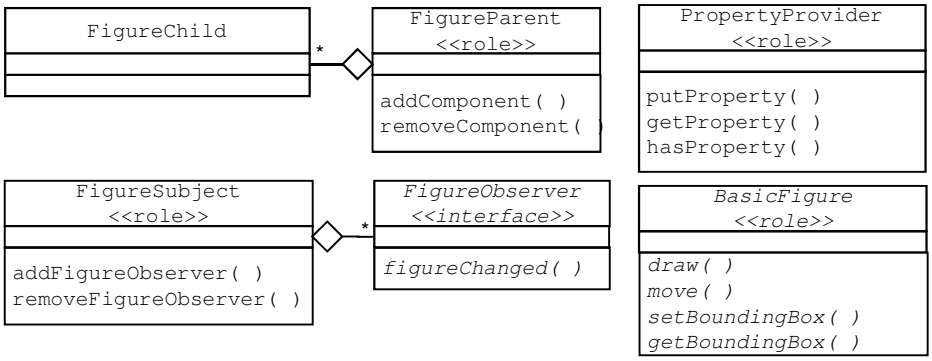

Figure 4.2: Roles created for the Figure framework.

Besides the roles defined by the patterns we can assume that the Figure concept is itself a role, named BasicFigure. Since roles are components of classes and are not visible to the class clients we prefer to name the role BasicFigure and maintain the name Figure for the interface. This role defines only the structure and behavior of a figure, and is played by all the classes in the hierarchy. The best way to model this is still using an inheritance hierarchy, but somewhat modified. Figure 4.2 shows the roles defined for this framework and figure 4.3 shows the class diagram of the relevant part of the Figure hierarchy.

4.2.3 Role GUI Framework

The transformations for the GUI framework are similar to the Figure framework. We identified the several concerns that components deal with and place each set in a role. There will be a role for the Composite concept, a Subject role and an Observer role for each of the various Observer patterns, and a Child and Parent role for the Composite pattern. Also a PropertyProvider role is used. The roles used in this framework are shown in 4.4 and the class diagram of this framework is shown in 4.5.

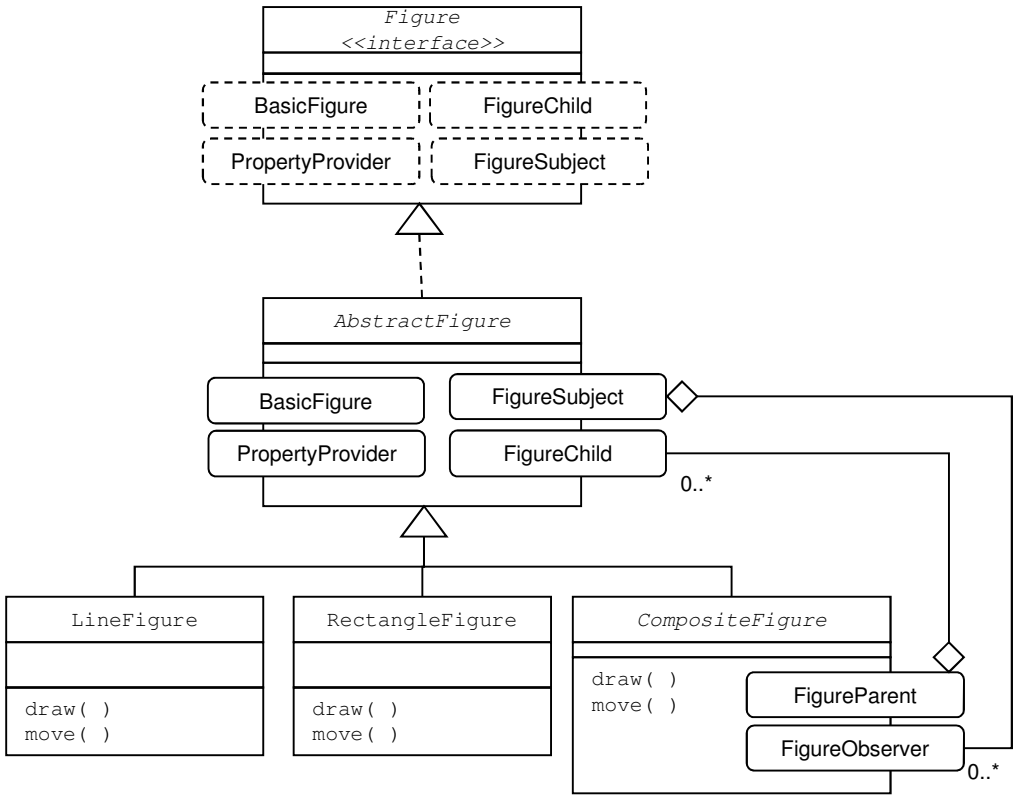

Figure 4.3: The Figure hierarchy (excerpt) with roles.

4.2.4 Code Replication in the Role Solution

Roles overcome the problems with multiple inheritance for the languages that do not provide it. That is the case of the Picture class mentioned in the OO version. We created a PictureFigure class and made it inherit from the Picture class and implement the Figure interface. But we had to replicate the code common to all figures (the one in the DefaultFigure class). Now we can make PictureFigure play the role of BasicFigure thus reusing the code in the role, avoiding such code replication. Figure 4.6 shows the class diagram of this solution. We can verify how roles can in fact be used to perform a kind of multiple inheritance.

The roles for the observer patterns are reusable too. Whenever we need to use a FigureObserver subject we only have to make the class play the subject role. That too is a code replication saver, but somewhat limited: FigureObservers are unlikely to be used in another context.

The various observer roles, from both frameworks, still share a lot of semi-identical code. The same is still true for both composite patterns.

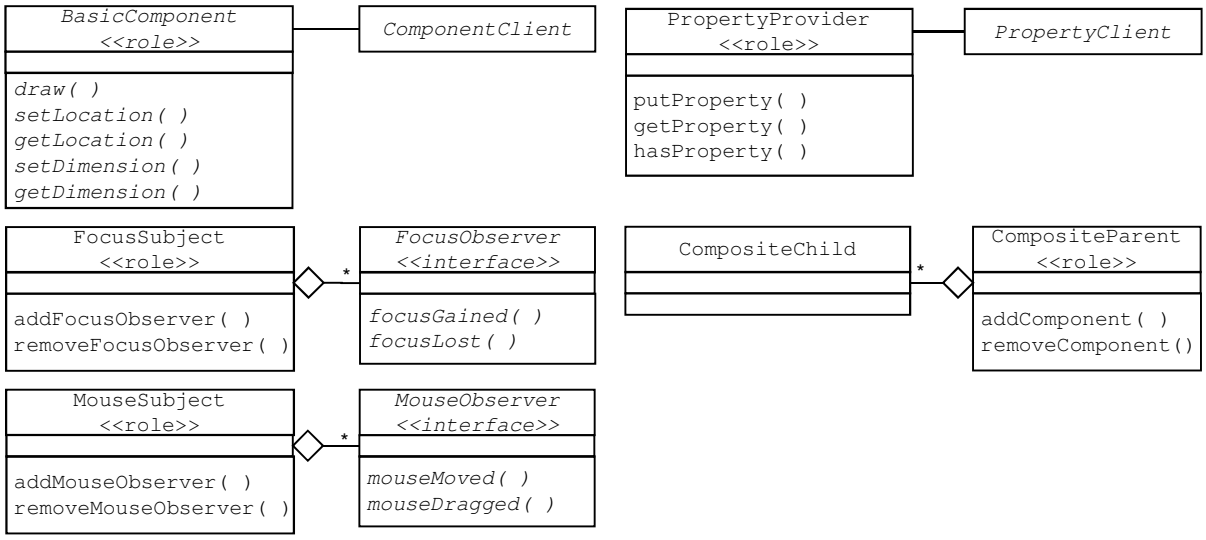

Figure 4.4: Roles created for the Component framework.

4.3 Dynamic Roles Characteristics

Even though we do not use dynamic roles this chapter would not be complete if we did not include a description of their use, so this section presents the characteristics of dynamic roles.

There are several languages that support dynamic roles, and many role models. Everyone chooses a set of role characteristics that suits their needs. A complete description of roles characteristics is therefore lacking if we take a language only. This section tries to present some properties of roles and practical considerations and is largely based on [Gra06].

4.3.1 Classes Playing Roles

A class may have restricted role-playing capabilities ranging from the number of roles and the type of roles it can play. A definite class may not play any role. Examples are final classes in Java. Since playing roles is a mean of extending a class, final classes must not play any role. In some languages a class must have at least one role attached, such as Fibonnaci [AGO95] where a class is a null-object. A class may also play a fixed number of roles even though it is not common. More normally a class may play an undetermined number of roles.

Even if a class may play several roles it may put restrictions on the roles it can play. Playing the same role more than once is prohibited in languages that rely on role type for method dispatching. Other languages like ObjectTeams [Her05] allow a class to play the same role several times but in different contexts. A class can have conjunctive or

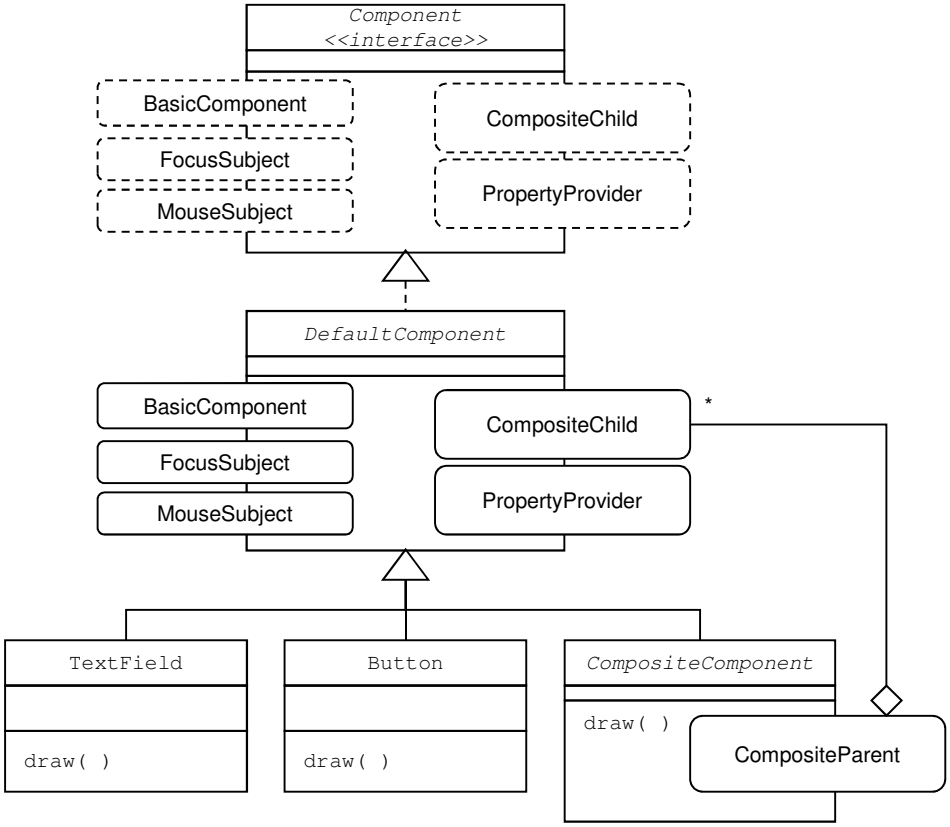

Figure 4.5: The Component hierarchy (excerpt) with roles.

disjunctive attachment of roles. In conjunctive attachment roles are like wrappers that fully cover the intrinsic (attached object). Attaching another role means wrapping the previous role with the new role. In disjunctive attachment a role may be attached directly to the object even if it already plays other roles. Figure 4.7 shows the two kinds of attachments.

Classes conceptually cannot inherit from roles [Kri96]. Roles are typically written for a class, or its subclasses. Abstract roles are those not written for a particular class.

That an object may play several roles is unquestioned, but what about how many objects can a role be attached to? Traditionally roles can be attached to only one object. Multiroles are roles that can be attached to several objects.

Visibility of roles is also important. Normally roles are public because roles usually model object interactions but there can also be private, and protected, roles. Here the notion of private can vary: a role can be private to the object it is attached to or private to the class that defines it, if it's defined inside a class.

4.3.2 Roles Playing Roles

A role may also play other roles. The same thought for classes apply here: roles can play no roles, play a fixed or an unlimited number of roles. Definite roles cannot play another

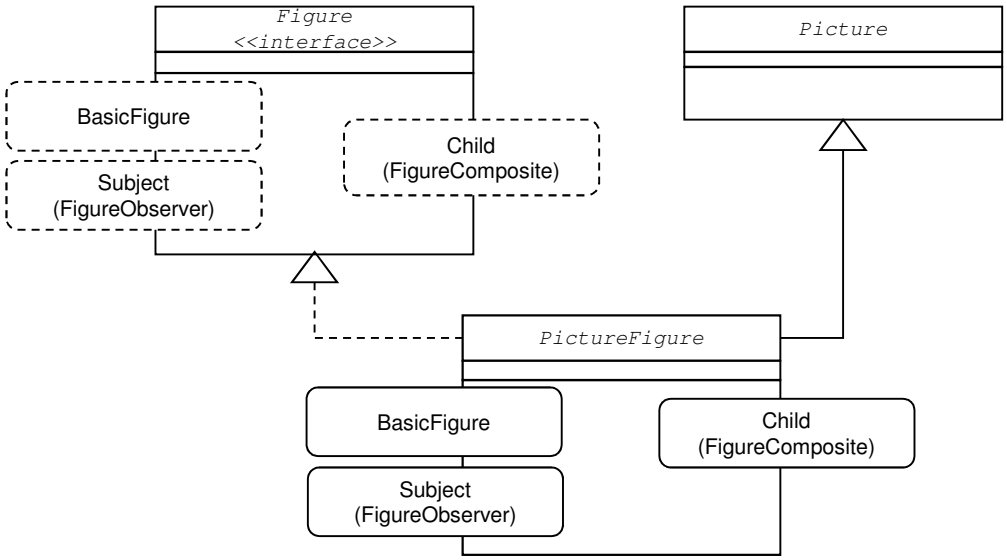

Figure 4.6: Possible solution for the Picture problem. Shows how roles can mimic multiple inheritance.

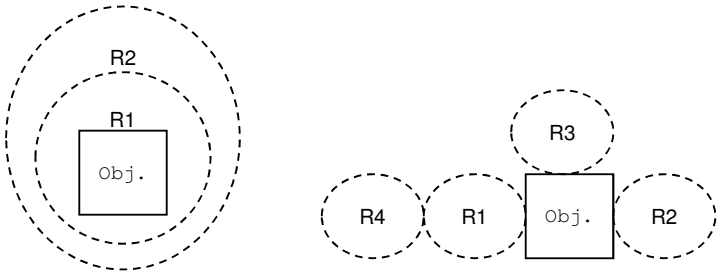

Figure 4.7: Examples of Conjunctive role attachment (left) and Disjunctive role attachment (right)

role. Like classes the type of roles a role can play can vary (see previous section).

Limits on the number of role instances can also be applied. A singleton role is a role with only one instance. A singleton role implies others decisions: is the role a singleton as in the Singleton pattern [GHJV95] or is it shared between objects? It may also indicate that when the role is attached to a new object it is detached from the previous object. All variations are possible. An instance of a role can also be attached only to a single object but several instances of the same role can be attached to the same object (if a class can play multiple times the same role).

4.3.3 Supertypes or Subtypes

Controversy exists whether roles are considered as subtypes or supertypes of its intrinsic. We can see why in the following paradox: at compile time a role may be considered a supertype because its concept is wider than that of its intrinsic (a costumer may be played

by a person or a company) but at run-time the opposite holds (not every person plays the role of costumer). The subtype view is the more consensual view, though.

Another view states that roles are an unrelated type of their intrinsic. This is the view adopted by many role models that are implemented as design patterns [GHJV95]. A role can extend a class, even though they are conceptually different. This is done for code reuse. Roles can extend other roles but some restrictions apply. If a role is an abstract role there are no restrictions. If a role has an intended intrinsic this restricts the subrole as the subrole intrinsic specification cannot be of a reduced type compared to the superrole intrinsic. Here reduced is used in the sense that the type cannot have fewer properties. A subrole may, on the other hand, extend the intrinsic specification.

4.3.4 Defining Properties in Roles

If we allow roles to have fields and methods this will eventually lead to name collisions. This happens if an object has two attached roles with the same methods (or fields). This is bound to happen if the same role is attached more than once to the same object. If this is an intended procedure then bookkeeping of the fields must be ensured. If it is not intended then accidental attachment of the same role has to be prevented. The solution to these problems are language specific and even role specific.

With methods there is a further question: the methods in the role overrides the ones in the objects or vice-versa? The answer may depend on whether the relation between the role and the intrinsic is of a super or sub type nature. It may also depend on the language (C++ allows only the overriding of virtual methods. In Java all methods are virtual and overriddable, except those declared final).

A possible solution is to prevent name collisions. This however renders dynamic attachment of roles impossible if there are abstract roles. It also prevents multiple attachments of the same role. Another solution is to allow renaming of methods.

4.3.5 Method Call

When entity A sends a message to some entity B what happens? The answer depends very much on the role model being used. First it must be known to which entity A is referring to. If A is referencing a role (as in most role languages) then A is calling a method in the role. If A is referencing the object then the method must be dispatched to the role by some mean. This can take a very complicated route because a role may be playing another role.

What happens when several roles define the same method? Some languages call all the methods, others have some mechanism to select one method. The problem with calling

all the methods is the return value. ObjectTeams [Her05] for example leaves the result undefined. Some mechanisms involve selecting the last attached role. Others techniques forces the caller to disambiguate the call by using casts.

There are other considerations such as: may a role call its intrinsic methods or vice versa? Can a role place calls on other roles? What if a role and an intrinsic have the same methods? Note that this situation is different from a method called from an external entity. Some languages define a reference self to contrast with the this reference. The this reference indicates the intrinsic while self references the role, thus disambiguating the call. Some impose a restriction on the intrinsic calling its role's methods. It does assume that the intrinsic knows its roles and may depend on them. In dynamic situations this cannot be allowed as the role may not be there. An alternative way is the intrinsic querying its roles for the presence of a particular one and then call its methods.

To further complicate matters, around methods may be added. An around method is a method that overrides the method in the intrinsic. These introduce more method dispatch problems if different roles with the same around methods are attached to the same object. When the method is called which around method is called? This is a semi identical problem with normal method calling. A further problem is if the around method calls the intercepted method: is the other around method called or is the intercepted method called? The situation in which the around method does not call the intercepted method also presents problems: does the other around method gets called or not? If it is not then the first around method does not only overrides the intended method as it does every other around method. Since roles must not depend on each other, this may break the other role. If it does call the other around method then it in turn may call the intercepted method, which the first around method prevented from being called, thus breaking the first role. This is similar to the composing of aspects problem in aspect-oriented programming [LHBL06].

4.3.6 Role Identity

The question is whether a role has an entity of its own or shares the identity of its intrinsic. Considering that roles are not independent entities the roles should not have an identity of their own. The objects and its roles are referenced as one. If roles can evolve objects they cannot modify its identity, otherwise it ceases to be the same object. To identify the role to be addressed casts are used. This however prevents multiple roles of the same type.

Roles can have identity if it is separated from references. This calls for the use of methods to compare identities. But further distinctions must be made: if two roles are the same and if two roles have the same intrinsic.

4.3.7 Roles Lifecycle and Movement

Roles can be taken on by the object or imposed from the outside. When a role is superimposed it is automatically attached when the object enters a specific context. Either way attaching a role does not necessarily imply memory allocation. Roles can be cached and when a new role is needed old ones can be used. Roles that are not referred to are cached instead of garbage collected. Languages that permit only one instance of the same type can verify if the role is already attached and reuse the same role, instead of creating a new one. Attachment strategy is discussed in section 4.3.1.

Role movements reflect the dynamicity of the real world, but it may be hard to achieve and poses many real problems. The major problems with role movement are the dynamic situations that may arise.

The example in Kristensen and Osterbye [KO96] refers to the role Mayor. When a person is elected Mayor it assumes the role and when another person is elected the role gets transferred. But if we move the role then the person that played the role of mayor does not recall being a mayor. It is possible that other clients knew the person as a mayor and after the move they now know the new mayor and don't remember the other person. There are cases in which only the functionality of the role is important and not who plays the role. In these cases the role may be moved freely without the previous problems (it may have others, though).

Moving a role must also take into account if a role is currently executing or not. If the role is not executing then it may be safe to move it. If roles have identity then clients may have the knowledge that a role is attached to a specific object and moving the role may invalidate that knowledge. The same can be true if a client does not know the specific object but depends on a particular state of that object, when moving a role to another object the state may be changed without the client knowing it. Kniesel [Kni96] presents other reasons to invalidate roles from moving.

If a role is executing then the problems only get bigger. When the role resumes its execution the environment has changed! Some solutions let the role finish its execution and then perform the movement, but that may not always be possible (it may even lead to deadlocks). Solutions for dynamic situations are very difficult to obtain because there are infinite possibilities.

Moving roles must consider the fact that roles may play other roles. Thus when moving a role it makes sense to move along all the roles it plays. But that may be untrue if the target intrinsic has restrictions on the roles it may play. It may play role r1, but may not play role r2. If role r2 is attached to role r1 then the move fails. What to do? Not move role r1 or detach role r2 from r1 and then perform the move? The correct way depends on the concrete case, even though a default behavior can be enforced. Either way new

problems arise. Detaching role r2 may not be possible or if it is detached should it be replaced by another role?

Because of multiple views when moving a role all the references to that role must be updated. A notification method should also be called to prevent the problems mentioned above when a client depends on a specific object or specific object state.

Because of the many problems involved many languages forbid the movement of roles. Once a role has been attached it cannot move. This increases code predictability [BW00].

A similar problem with moving roles is removing roles. The main problem is the past being removed with the role. When a student ceases to be a student it has nevertheless a record of grades, etc. When the role is detached it is as if the object never has been a student. There is also the fact that clients have the references to the role. The school still has references to the student's role. It may depend on the role to know the name of the student. If the role is detached then the school loses this information.

A solution for role removing is role replacement. We could replace the student role with a graduate role that stores the record information. But that is done on a case by case scenario. There are also all the problems with role movement to be dealt with, like removing a role when it is executing and so forth.

Another solution is not to allow the removal of roles. When attached it remains attached and only terminates when the object terminates: it is a life role. Roles may be removed explicitly or implicitly. When explicitly removing a role language support is important. C++ for example offers explicit removal of objects. Java on the other hand does not. So it may be difficult to know when a role is terminated or not. In C++ explicit removal implies memory management that can be very hard with roles and may be hazardous as it may leave dangling pointers. A common way of avoiding it is to mark a role as active or inactive. Roles are removed implicitly when no one is referring them. If an object has references to the roles it plays then the role will never be garbage collected. If soft references are used then the role is removed if no one else is referencing it.

4.3.8 Role Visibility

A class can have inner roles and a role can have inner roles as well. Conceptually there is nothing against it. Making the inner roles visible to the outside is forbidden. An inner role should only be used by the outer class (or role) itself, by a mean of alternating between states, or attached at inner class's objects. A justification for making inner roles public is the fact that particular role should be applied to that class only. Since any role may be written for a specific intrinsic this invalidates this argument, leaving no other argument to make inner roles visible.

A consideration to have in mind is the role interface. Since it will expand the intrinsic interface, how will the various properties that the role defines be seen from outside? Traditional access modifiers are the private, protected and public. We expect that these access modifiers apply to the role as well.

The interface between roles and intrinsics must also be debated. Does a role have access to the intrinsic properties? Does an intrinsic have access to the role properties? How to apply the public, private, protected interfaces here? Some answers depend on the relation between roles and classes. If a role is a supertype of the intrinsic it cannot access intrinsic properties. The intrinsic can access the public and protected ones from the role. If a role is a subtype of the intrinsic then the intrinsic cannot access properties of the role but the role can access the public and protected ones. If roles and classes are unrelated the question of the role-intrinsic interface raises. Can each other only depend on the public interface or can they rely on protected interface or even on private interface? It may be useful to create other access modifiers as roleprotected (or classprotected) [Gra06] that enables a special interface between roles and classes. Of course this can only work if a class knows the roles it may play. If it does not then it won't provide the interface the role may need and won't rely on the role interface.

4.3.9 Exceptions

Several languages deal with exceptions and some have the notion of checked exceptions, meaning that a method that may throw an exception must declare it in the method signature. Roles may override a method thus altering its exceptions signature. In contexts where the role is expected this is not a problem, because the role interface is known. Where a role is not expected then this is not allowed, because that would alter the method signature the client was expecting.

4.3.10 Renaming Properties

A solution to the name conflicts would be to allow the renaming of the methods or fields that the role offers. The role itself can also make an alias of the intrinsic fields. Another reason for renaming is to allow the attachment of a role to an otherwise incompatible class. Few languages support this.

4.4 Summary

In this chapter we discussed the role concept in its several uses throughout the literature. It has been used mainly to describe dynamic situations but it can also be used as a way of

composing classes in a static way.

The dynamic situations offer developers a way to extend an object but when supporting full dynamic situations several problems arise that may render the dynamic approach impossible. Hence a limitation on the attachment/detachment of roles is imposed. Another solution implies the use of contexts where the object gains the roles when it enters that context and they remain active only in the context, thus limiting role interactions and possible incompatibilities.

The static version of roles have been used mainly for modeling and the few role supporting languages do not allow for an advanced class composition, treating roles as little more than interfaces. When faced with the problem of code cloning proposed languages do not provide a full set of features that reduces this replication to a minimum.

Part II

Problem and Solution

Chapter 5

Research Problem and Solution

Code cloning is a problem in many systems, especially large ones. While some clones are intentional and need to be monitored the majority should be removed. To remove clones the available tools and techniques often resort to refactorings [BMD+00, FR99, HKKI04, HKI08, KH00]. But there are still clones that cannot be removed using these tools or techniques.

Class composition also has an impact on code clones as more ways to compose classes provide more means to reduce replicated code. Thus with a better compositional strategy we could prevent the appearance of clones. Roles have been used as a way to expand classes and also as a way of modeling systems. Can roles also be used to compose classes and thus provide more ways to reduce code clones?

In this chapter, several open research issues are raised focusing on clone removal and the benefits that roles can bring both in terms of class composition and clone removal. The research questions and thesis statement are presented and explained, as well as the proposed solution approach. Finally, the research and validation strategies are debated as the baseline to pursue empirical studies.

5.1 Open Issues

From the state-of-the-art review presented in the previous chapters, a number of open research issues arise. An insight of the most relevant ones follows, intended to focus the scope of the work presented in this dissertation:

- **Not all clones can be removed.** Not all clones must be removed either. In this category are included clones for risk reduction purposes or performance issues. But there are still clones that are not intentional and should be removed. These clones are the focus of interest in this thesis. A description of these clones and why they cannot be removed is given in chapter 7.

- **A single decomposition strategy is not enough.** In Object Oriented decomposition, programs are decomposed in terms of objects and classes. But often classes have to deal with concerns that are not directly associated with their main concern. Some of these concerns tend to affect a set of classes in the same or similar way, which leads to replicated code. These concerns are called the crosscutting concerns. The use of a single decomposition strategy is not enough to deal with these crosscutting concerns and this motivated the development of several decomposition strategies. As we have discussed in chapter 3 code clone still exist in spite of the use of these techniques.

- **Roles are used in the design but not in the code.** The work by Riehle [RG98, Rie00] and the OORam method [RWL96] showed how roles can be used to model a system, neatly modeling all the various views we have from an object. However, they did not propose a language that deals specifically with roles and only described how the design could be implemented using traditional classes. This introduces an important gap between modeling and implementation. The use of roles in programming is restricted to dynamic languages as discussed in chapter 4. This type of roles, due to their dynamic nature, does not deal well with the problem of code clones that we are interested in. Also, in these dynamic languages, roles are developed for a specific class or, at the most, classes of a certain type. To deal with code clones we need a static approach to roles and one which enables the roles to adapt to their players and not the other way around as is the case with dynamic role approaches.

5.2 Research Questions

From the aforementioned open research issues, a few research questions revolve around a major question that is considered central to the presented research work: *How to reduce code clones*? Those questions are listed next.

- What are the limitations of existing clone removal refactorings?
- How can we prevent clones to appear in the first place?
- What language limitations may force developers to introduce clones? How can we overcome such limitations?
- How existing decomposition techniques cope with code clones?
- Are roles useful for the implementation phase or are they useful only for modeling?
- How can roles tackle the code cloning problem?

5.3 Research Focus

The research work presented in this dissertation covers subjects from all of the fields and topics described in the earlier chapters (Chapters 2, 3 and 4). Due to the complexity of such areas we do not intend to address all the problems identified in those chapters. From each described area we will be taking a subset that will allow us to reach the thesis goals, that is, the reduction of replicated code.

The major focus of the thesis is on reducing code clones. But as we have seen in chapter 2 not all clones are undesirable and not all clones are casual, some are intentional and serve a well defined purpose. Of course we are not interested in removing these clones too. We are more focused on removing clones that arise from language limitations, especially from composition limitations. To make the focus of the research more clear, table 5.1 shows, for each of the clone factors described in table 2.1 and discussed in section 2.1, the interest level for the research..

Like shown on table 5.1, we intend to focus especially on reducing clones that are due to the lack of composition techniques. For that we will concentrate on the object-oriented technologies, since these are the most used today. From all the described composition we discarded the AOP approach as it strays a bit from pure OO decomposition. We will focus our approach on the role modeling, taking it all the way to the implementation stage. For clarity in the domain areas and focusing of the research, Figure 5.1 depicts the area in the decomposition techniques where the results of the work are focused on and where expected to bring the most contribution.

Category	Subcategory	Factor	Scope	Priority
Development Strategy	Reuse Approach	Simple reuse by copy and paste		
		Forking		
		Design reuse	*	***
		Functionalities/Logic reuse	*	***
	Programming Approach	Generative programming		
		Merging similar systems		
		Delay in restructuring		
Maintenance Benefits	Avoiding	Risk		
		Unwanted design dependencies		
	Ensuring	Robustness		
		Better performance in real time programs		
	Reflecting	Design decisions (e.g., crosscutting)	*	***
Overcoming Underlying Limitations	Language Limitations	Lack of reuse mechanisms	*	***
		Abstraction creates complexity	*	**
		Abstraction is error-prone	*	*
		Significant efforts in making abstractions	*	**
	Programmers' Limitations	Time Limitations		
		Performance by LOC		
		Lack of ownership		
		Lack of knowledge in the domain	*	*
		Difficulty in understanding large systems	*	*
Cloning by Accident	Language Paradigm	Protocols to interact with API and Libraries		
	Programmers Working Style	Programmers mental model	*	*
		Unknowingly implementing the same logic by different programmers	*	**

Table 5.1: Clone Factors that are the focus of the research.

5.4 Thesis Statement

Based on the research challenges presented (sections 5.1 and 5.2) and the state-of-the-art review (Chapters 2, 3 and 4), the author states that:

> *"The use of roles is an effective way of preventing and reducing code replication, which provides better results than traditional approaches, and contributes to enhance the system modularity, while closely following object-oriented principles."*

This statement uses terms whose meaning may not be consensual, and therefore leads to questions that deserve further discussion:

- **What is meant by "*effective way*"?**

  Effective can refer to a good performance, a good design or simply that it does reduce the code replication. When we mean effective we apply it in all these senses. Roles can in fact reduce code replication, so they are effective in that respect, but they also do it without incurring in a performance loss or by introducing significant design changes. Significant changes being described as not having to create new classes (new roles may be created) or changing the classes hierarchies or even the

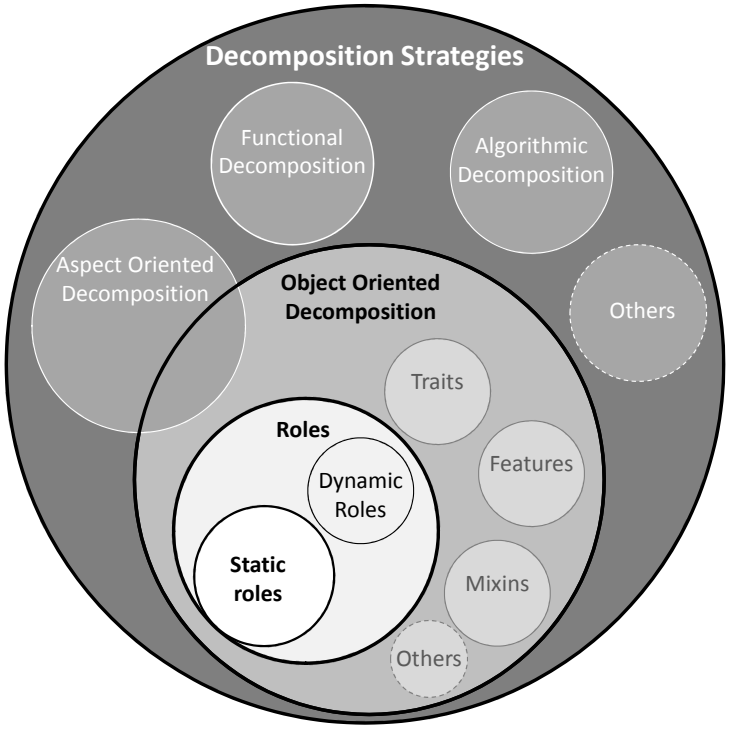

Figure 5.1: Research focus on the Decomposition Methodologies

way it relates to other classes. Thus design changes introduced by roles affect only the classes that contain clones and, in that respect, only the part of the class that has the clone. Class clients should suffer no change.

- **How can we measure "*better results*"?**

  Better results once again may be seen as better performance or even better design. We apply the term here in the sense that roles can remove clones that traditional approaches cannot while preserving most of the basic code structure. As stated previously, both performance and class clients are not affected, so roles provide better results than approaches that need to significantly change the design. So better results refer to a greater number of clones being removed while retaining most of the initial system design.

- **What are the "*traditional approaches*"?**

  Traditional approaches (section 2.5.1) refer mainly to the refactorings used in most clone removal tools available or clone removal proposals. But this term can also be used to some of the class composing techniques that are described in chapter 3. Some class composition approaches do not deal directly with cloned code but deal with crosscutting concerns that ultimately leads to duplicated code. We include in

the terms traditional approaches those approaches that can be used to reduce code clones, like AOP and Traits .

- **What is the meaning of "*enhance the system modularity*"?**

  Code clones are often a by-product of language limitations to cope with crosscutting concerns. We expect our approach to reduce the amount of replicated code that originates from this. Furthermore we will be able to encapsulate such code replications in a role that may be reused. This role reuse is what we mean with the "enhancing the system's modularity" statement. We expect some developed roles to be generic enough to allow us to build a role library.

- **Why is it relevant to follow *object-oriented principles* ?**

  Object-oriented programming is the most used decomposition strategy today. If we want an approach to be adopted by a large number of developers we must not impose a completely different mental model. Despite its limitations OO development has proved efficient and developers are comfortable and efficient in its use. It feels a better choice to introduce small changes to the OO model. This way developers remain in known ground and there is not a paradigm shift. The adherence to the object-oriented model means that our roles must be abstractions from the solution model and not mere artifacts to reduce code. Developed roles must represent a specific concept present in the solution model. Complete replicated code removal could be achieved but that would lead to developing roles that did not have any particular purpose except that of reducing code replication.

The original thesis statement can be decomposed in the following hypothesis, as a more objective means to validate the author's assumptions:

- **H1:** Roles can be used to compose classes.
- **H2:** Roles can be reused.
- **H3:** Roles can be used to remove more code clones than traditional approaches.
- **H4:** Roles can be used to prevent code clones.

5.5 Research Goals

The primary outcomes of this thesis encompass the following aimed contributions to the body of knowledge in software engineering:

1. **Reduce code replication using roles as a composition mechanism;**. The main goal of the proposed thesis is the reduction of code replication. One way to reduce it is to enhance system composition. This allows the same code to be reused by composing it in different ways throughout the system. In our opinion roles are well suited for this task as they offer a good mechanism for code composition, and act at a finer grain level, such as classes or other roles. With roles we can have a smaller module than the class, thus enhancing code reuse. Role researchers have concentrated in dynamic roles or modeling using roles. We will head a different way, by creating generic roles: roles that can be easily tuned to fit a particular class. A major contribution that we make is to build a library of generic roles that may be used as building blocks for more specific roles or classes.

2. **Explore new ideas in role composition**. The focus will be on avoiding code replication but it is also a thesis goal to explore new ideas on role composition. Specifically we will try to simplify the declaration of roles. Many role languages assume that roles are associated with a context so they include contexts as a first class construct. By doing this the language becomes cluttered with many new concepts making OO developers feel uncomfortable. This may be a reason why roles have not yet reached mainstream languages. A new role model may have to be developed or existing role models have to be extended. There is also a limitation in roles languages that we seek to overcome: many role models require the role to declare beforehand its possible players by means of a "played by" clause. This can be done using the name of the player class or even an interface. One reason to use the "played by" clause in the role is to impose a constraint on the classes that may play that role, to ensure that the player has a certain interface (or structure) that the role uses. Our solution will take a novel approach as we want to let classes configure the roles they play and not the roles that define which classes can play them. The inversion of declaration from the role to the class will allow us to provide a renaming mechanism that configures the role as it should be used by the class. The role must, nevertheless, impose some restrictions on the player interface when it needs to communicate with the player, therefore we must provide a way of stating these restrictions but keep it to a minimum.

3. **Enhanced modularity**. With roles we expect to have a higher modularity than with traditional OO approaches. Since roles can be seen as a smaller module than the class then we have an all new level of modularity. Roles can represent concepts that would not be directly allocated to a specific class but would appear in several classes that must perform similar tasks because the way they interact with other

classes. By pinning these concepts to roles we can therefore reuse them whenever we want. Future changes are also limited to the role, instead of having to perform the changes in the various classes.

4. **Propose language extensions and tools**. We propose an extension to the Java programming language for validation purposes. A compiler that supports these extensions has been developed. It must be stressed that easy of use is a central point in the intended work, so building a whole new language would refrain programmers from considering our approach. Extensions to the selected language were also kept to a minimum, without compromising expressiveness.

5.6 Proposed Approach

There are many reasons for the occurrence of clones (see section 2.1), one of them are crosscutting concerns, that is, concerns that a class must deal with that are not its main concern. When several classes deal with the same concern they tend to use similar code. This is more frequent in languages that do not support multiple inheritance. With multiple inheritance we could place the concern in a superclass and all classes would inherit the same behavior. With single inheritance we tend to replicate the common behavior in all classes, if we cannot find a common superclass.

Some clones could be avoided if a language had other composition mechanisms. In particular crosscutting concerns would benefit from the use of a composition mechanism where a class could be composed of several pieces of software. We believe that if we explore the way roles can be used to compose classes we'll find that roles are capable of reducing several clones that traditional clone removal techniques cannot.

There are many definitions of the role concept in the literature (see chapter 4) but we are interested in using roles as components of classes. For that purpose we use the role definition used by Riehle [Rie00], where roles are an observable behavioral aspect of an object. We can use roles to compose classes, meaning that an object's behavior is defined by the composition of all roles it plays.

We propose to use static roles as a basic unit from which we can compose classes. Roles provide the basic behavior for concerns that the classes deal with but are not their main concern. Thus we can better modularize those concerns. Static roles have been used for modeling [RG98, Rie00] but despite their benefits no programming language has appeared that deals with this static nature. To overcome this we will develop a Java extension that supports static roles.

When we started developing our role model the main goal was to enhance code reuse while maintaining the model as close as possible to OO decomposition. We believe that

this approach is more likely to get the acceptance from the OO community than a model that would introduce many new concepts.

One goal of our proposal is to make roles modular and therefore reusable, thus contributing to diminish code replication. If we consider roles as modules then they can be reused and we can build a library of roles. Using modularization [Par72] we can build a set of independent modules meaning we can develop and change each module independently from other modules. This shortens development time and augments comprehensibility because one needs only to deal with a module at a time. Modularization allows the development of libraries thus enhancing code reuse and reducing the amount of code one needs to write. We intend to prove that roles are reusable by developing a library of roles. If such a library can be built it also shows that roles are capable of preventing code clones.

To expand the re-usability of roles replacing the "playedBy" for a list of requirements can be complemented with a renaming mechanism. A name of a method must clearly state the purpose of the method but when creating a full purpose role those names are difficult to achieve. The methods' names for a role are tuned for a particular interaction. In similar interactions, however, where we could reuse the same role those names would be inadequate or just plain misleading. As an example, the Observer pattern [GHJV95] describes an interaction between subjects and observers that is present in many different systems with only minor changes, most notably the names of the methods to register an observer with a subject and the methods used by the subject to notify its observers. A Subject role for a MouseMotionListener instance of the pattern would define methods like addMouseMotionListener, or removeMouseMotionListener. That role could not be reused for an instance of the same pattern but for a KeyListener which uses methods like addKeyListener or removeKeyListener. Because a method name must somehow indicate its functionality, using a generic name like addListener would not be correct as it would reduce the comprehensibility of the code.

5.7 Validation Methodology

A compiler that supports the language extensions was developed. This tool could be overlooked but that would lead the evaluation to be based only on theoretical and argumentative terms. We feel that this is somewhat against the spirit of the thesis, as it focus on a rather practical problem. One reason to choose Java for the extended language is the OpenJDK project, namely its compiler group, which eases the addition of new features to the language.

Since we will focus on code replication, a large amount of code has to be analyzed. We will look for duplicated code in public frameworks and applications. The tool to be used is

the Clone Detection tool CCFinder [KKI02]. Framework analysis serves both the purpose of gaining more experience of code replication within real world code and the purpose of validating the approach as they will be the subject of the case studies.

To validate our approach we will conduct a series of case studies. The case studies include two frameworks and one application. For each framework/application we detect replicated code using clone detection tools. Then we try to develop roles that are suitable for the concerns involved in each detected clone. This will allow us to show to what extent our approach can reduce the replicated code.

In order to focus only on the reduced replicated code and to ensure an unbiased comparison we will not change any class interface or the way each concern is implemented. The point is to compare code reduction and not to compare the framework solution with and without roles, even if this would be useful to express the advantages on the use of roles for modeling. But modeling advantages have already been shown in [RWL96] and [Rie00] so we need not repeat that study. So, if we started to change the framework around roles we could not later pinpoint the advantages of roles in directly reducing replicated code. The amount of replication saved between the original and final versions will be used as a measure of the approach impact.

Further validation is obtained from developing a library of generic roles that may be used to create more specific roles or classes. If such a library can be produced then it means that the approach is indeed capable of creating generic roles and thus to better modularize the system. We expect to use some of these roles in the target frameworks/application. If we succeed we can safely say that due to the reuse of these roles we did not have to reproduce their code in the system thus preventing clones from occurring.

5.8 Summary

The research in the three fields that are covered by this thesis (code clones, composition techniques and roles) has been abundant but some issues still remain. The clone problem is a serious one and existing refactoring techniques are not enough to reduce it in a satisfactory way. In the roles field of work several approaches exist but few deal with the static nature of roles and none proposes a supporting language and how they can be used to reuse code. We propose to merge all the fields and, specifically, we propose to tackle the code clone problem by using roles as a composition mechanism. For that we propose a new role language and use it in a series of case studies that show how it can be used to reduce the clones found within each case study. We also show how roles can be used to provide a better modularization of crosscutting concerns by developing a role library for well known patterns.

Chapter 6

JavaStage

This chapter is dedicated to the presentation of the JavaStage language. To explain the nature of the language and the various options made when developing it we describe the motivations that drove its development. We then proceed to present the JavaStage's syntax and how to use it.

The JavaStage implementation is described and the strengths and weaknesses of the language are described in the light of that implementation. Also presented are some implementation alternatives that were considered during the language development and the reasons for not selecting them. To end the chapter a comparison between JavaStage and other approaches is made.

6.1 Development

One of the forces that drove the implementation of JavaStage was Modularity. We intended to build a library of roles to prove that roles could be reused. For that purpose Modularization [Par72] is one of the most important concepts in software development. Breaking a system into modules allows the independent development of each module. This shortens the development time as each team may develop their assigned modules

simultaneously. Independent development also enables the modification of a module, even drastic ones, without any changes to other modules. Another advantage of modularization is comprehensibility because one can study the system one module at a time. There are numerous advantages of modularization like enhanced error tracing and fixing, reduced system compiling time, etc, but the one that we, as developers, treasure most is the high reusability of modules. This allows the development of libraries which in turn reduces the amount of code one must write in order to build a system and with extreme benefits in system reliability (assuming libraries have been thoroughly tested).

A key concept in modularization is encapsulation. When a module is well encapsulated changes in that module do not affect any other module. A module has an interface and an implementation. The implementation is the way the module is built. The interface defines how clients interact with the module. Since this is what clients see and use it should not change much along the module's life-cycle as clients must be aware of the changes and in turn change their implementation accordingly.

Modules interact with each other but some modules are more tightly connected than others. The scaling property of modules allows the building of a module using several other modules. The inner modules interactions are more intense than outer modules interactions. These interactions may require a specialized interface that the outside modules don't need, and should not know of. To cope with this, most languages declare different levels of access to the modules members. In traditional OO programming languages there are, at least, 3 levels of access: private, protected and public.

6.1.1 Roles as modules

We will contribute to overcome the composition problems of existing OO languages by introducing a new and smaller module than the class: the role. Our view of roles is somewhat different from others as we focus more on the static nature of roles, as used by [Rie00], rather than on its dynamic nature as seen in PowerJava [BSI07] and ObjectTeams [Her05]. We also tried to stick as close to the OO model as possible and the syntax additions at the minimum. We, therefore, dismissed the ObjectTeams' Context concept and the PowerJava's Institution concept as they introduce greater complexity to the model. We believe that the small changes we introduced in the OO decomposition will be better accepted by the developer community than a completely new concept. The small learning curve aims also to be an advantage towards that acceptation.

For roles to be considered as modules they must provide an interface, ensure encapsulation and have to be developed independently from other modules. Providing an interface is straightforward. Ensuring encapsulation and independent development raises a few issues. One must consider the fact that a role only makes sense when "played" by a class. Does

this mean that the class playing the role has access to the role members and vice-versa? If this was so then the role could not be developed independently from the classes that play that role, because any change in the role implementation could cause changes in the class. The same holds if we grant the role with access to the class members. Then changes in the class may force changes in the role. From this discussion we can see that roles and classes have to be independent and rely solely on interfaces. Because roles and classes have a special relationship, even closer than a superclass-subclass relationship, it may be the case that we need a special role-class interface. For that we could use yet another access level, or redefine the meaning of the protected level to include the role-class relationship. We will discuss our option when we present our role model.

6.1.2 Extending the reuse of roles

Traditionally roles declare which classes can play them, whether by name or by interface, using a "played by" clause or similar. We believe that declaring predefined players is a great limitation in the reusability of roles. The same role used for a class could be reused for another class were not for the fact that the role developer did not foresee all its possible uses. One can argue that a role only makes sense in an interaction between classes and thus restrict the player classes to the ones involved in that interaction. However the same role could be reused in another, similar, interaction but with different players. If the "played by" clause specified an interface instead of a class it would be possible for the role to be played by many different classes but even this is not enough. A hindrance to this reuse is the name of the methods that are specific to an interaction. As an example, the Observer pattern [GHJV95] describes an interaction between subjects and observers that is present in many different systems with only minor changes, most notably the names of the methods to register an observer with a subject and the methods used by the subject to notify its observers. A Subject role for a MouseMotionListener instance of the pattern would define methods like addMouseMotionListener, or removeMouseMotionListener. That role could not be reused for an instance of the same pattern but for a KeyListener which uses methods like addKeyListener or removeKeyListener. Because a method name must somehow indicate its functionality, using a generic name like addListener would not be correct as it would reduce the comprehensibility of the code. Another major drawback is that it would limit the class to play only one subject role. Considering this, we believe that a renaming mechanism must be used in order to expand the reusability of the role to several situations. Of course some restrictions must apply, because a class that plays a role must ensure a specific interface, but that interface should be configurable, at least in what respects to method names.

Some languages [TUI07] use a "rename" clause that allows player classes to rename

method names. If the role interface is big then this task is tedious and error prone.

There is also the problem of the role calling the player methods. Again method naming is important. In the Subject role each subject has a method that calls the observer's update method. In the Java AWT-like implementation of the pattern such method is not called update but several methods like mousePressed, mouseReleased, etc., are used. The "rename" clause is not usable here because the number of methods that get called varies between instances of the subject role.

We need a mechanism that allows fast renaming for both role methods and methods that are called by the role.

To summarize, for roles to be fully reusable then they must:

- provide an interface;
- ensure encapsulation;
- have to be developed independently from its players;
- provide a method renaming mechanism that enables the role to be played by any class that fulfills some requisites.

6.1.3 Removing the playedBy

Existing role supporting languages use a playedBy clause or similar. That clause is placed on the role definition and it restricts the objects that can have that role attached. It means that only objects of that class, or its subclasses, or, if interfaces are used, its implementators, may have a role instance attached to it. We argue that this restricts the use of the role that otherwise could be reused in another context or with other, not yet developed classes. We believe that other forms of restricting which classes can be used may be useful, and we will present our solution.

The use of the playedBy clearly indicates that a role is developed for a set of classes in mind, that is, first we have the classes and then we build the roles. One can argue that in dynamic situations, where we want to extend the behavior of existing classes, this is acceptable, because we know which classes we want to expand. On the other hand the same role could be used for extending other classes that were later introduced. We will explain our point of view using some examples.

Considering the example from the figure 6.1, adapted from [IE11], where Point is a class that represents a point in a plane in a Cartesian coordinate system. Location is a role that is played by Point and provides a view of Point in the plane as physical locations on a map of the world. The syntax used is similar to the one used in the JavaStage. The performer keyword represents the intrinsic of the role, which we often refer to as the player.

```
class Point {
  int x;
  int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}

role Location playedBy Point {
  String getCountry() {
    int x = performer.x;
    int y = performer.y;
    String country = "PT"; // placeholder for
                           // computation converting a point in
                           // the plane to the name of a country
    return country;
  }
}
```

Figure 6.1: Example of a simple role.

What we can see from the role's code is that the role assumes knowledge of its intrinsic, like the use of the x and y fields. This means that any modifications to the fields in the class will lead to changes in the role as well. Since we want to enhance role reuse we must impose some constraints on the role-class interaction. Roles should be limited to use only the class interface as any other client. For this situation the Point class should provide the getX( ) and getY() methods. In the example the class does not have them, but in a well developed class those methods should be there in the first place, so we could use them.

What we can also see from the code is that the role is developed specifically for the Point class and cannot be used for any other class, except subclasses of Point. This is limiting as we could reuse the role for another class like PolarPoint which represents a coordinate in a polar coordinate space. This class instead of an x and y values stores a r and beta values. The r represents a distance (the distance from the pole of the plane) and beta an angle (the angle that a ray from the pole must have to reach the point). Figure 6.2 shows that new class and a role developed for that class. In this example we have already assumed that all class-role interaction is done via interfaces and that the PolarPoint has a getX and getY method to convert from polar to Cartesian coordinates.

What strikes immediately from the code is that the code for PolarLocation and Location is exactly the same, assuming we used the getter methods in the Location role. Nevertheless we cannot use a single role for both classes because of the playedBy clause. Since neither Point is a subclass of the other then we cannot use a single one to refer to them both. Making one a subclass of the other is not an option as this would violate the "is a" principle of inheritance. The only solution using the playedBy clause to prevent this duplicated

```
class PolarPoint {
  int r;
  double beta;
  PolarPoint(int r, double b) {
    this.r = r;
    beta = b;
  }
  int getX() { return r * Math.cos( beta ); }
  int getY() { return r * Math.sin( beta ); }
}

role PolarLocation playedBy PolarPoint {
  String getCountry() {
    int x = performer.getX();
    int y = performer.getY();
    String country = "PT"; // placeholder for
                     // computation converting a point in
                     // the plane to the name of a country
    return country;
  }
}
```

Figure 6.2: Example of a PolarLocation role identical to the Location role.

code would be to declare an interface Coordinate with the getX and getY methods and make both classes implement that interface. This is not doable because, in a dynamic context, we assume that both classes are already developed and cannot be modified.

We can argue that we could use a getPoint method in the PolarPoint that converted it to a Point and reuse the Location role on that returned point. But, although that solution is possible for this concrete example, it may not be so, and it hardly will, for all cases where classes share some of the behavior (getX, getY methods) but are not convertible to one another.

6.1.4 The need for a Renaming Mechanism

To expand the reusability of roles, replacing the playedBy for a list of requirements can be complemented with a renaming mechanism. A name of a method must clearly state the purpose of the method but when creating a full purpose role those names are difficult to achieve. The methods' names for a role are tuned for a particular interaction. In similar interactions, however, where we could reuse the same role those names would be inadequate or just plain misleading.

We refer to the Observer example given earlier. The Subject role for a MouseMotionListener would define the methods addMouseMotionListener and removeMouseMotionListener. The Subject role for a KeyListener would use methods like addKeyListener and removeKeyListener. Using a name like addListener would reduce the comprehensibility of

the code and limit the class to play only one subject role. The class that plays the Subject role for a KeyListener role could not play the same role for a MouseMotionListener.

Considering the previous example, a renaming mechanism expands the reusability of the role to several situations. Of course some restrictions must apply, because a class that plays a role must ensure a specific interface, but that interface should be configurable, at least in what respects to method names. A "rename" clause that allows player classes to rename methods that is done on a method by method basis is tedious so we need a more expedite way.

There is also the problem of the role calling the intrinsic methods or interacting with other objects. Again method names are important. In the Subject role each subject must notify its observer calling the observer's update method. The java's AWT implementation of the pattern, for the MouseMotionListener, does not use a single update method and uses several methods like mousePressed, mouseReleased, etc. The "rename" clause is not usable here because the number of methods that get called varies between instances of the subject role.

So even if we had a renaming mechanism for the role methods we still would not be able to produce a subject role that could be reused by every class. To accommodate this we need a mechanism that allows fast renaming for both role methods and methods that are called by the role.

To summarize, our guidelines when developing JavaStage, were:

- treat roles as modules, so roles must have an interface, state and behavior;
- roles and classes communicate via interfaces;
- introduce few language extensions;
- drop the playedBy clause.
- provide a method renaming mechanism that enables the role to be played by any class that fulfills some requisites while adapting its interface to each class demands

6.2 Syntax

We tried to introduce as few extensions to the language as possible and we only introduced 5 new keywords, which are responsible for the definition of a role, the requirements list, declaring that a class plays a role and representing the player type and object [BA13c]. We also introduced a renaming mechanism that makes use of a special character: #. The JavaStage syntax is presented in figure 6.3.

```
type_decl ::= (role_decl | class_decl | ... ) ";"
role_decl ::= {class_modifier} "role" identifier
              ["extends" class_name] "{" role_body "}"
role_body ::= ( requires | class_body )
requires  ::= "requires" type "implements"
            ( (type method_decl) | constructor_decl)
            [throws] ";"
class_body ::= ( plays | ... )
plays ::= {access_modifier} "plays" class_name [configs]
          identifier [role_params] ";"
configs ::= "(" config {"," config} ")"
config  ::= identifier "=" identifier
role_params ::= "(" args_list ")"
```

Figure 6.3: The extension of java syntax in JavaStage.

6.2.1 Declaring Roles

The syntax to declare a role is similar to that of a class. Roles can declare fields and methods like a class. When we want a class to play a specific role we use the plays directive. We must state that in most role models it is the role that states which classes plays them. As discussed in section 6.1.3, this is restricting the reuse of the role, so in JavaStage it is the class that states the roles it plays.

Role members have all the visibility control available to classes. We extended the protected level to include the role-class relationship. A protected role member is accessible to its players and subroles. A protected class member is also accessible to roles. In JavaStage roles provide an interface, have an implementation and provide encapsulation. Roles and classes are thus completely independent modules and may be independently developed.

In figure 6.4 we can see how the PropertyProvider and FocusSubject roles described for the Components framework in section 4.2.3 could be implemented.

6.2.2 Playing Roles

To play a role the class uses a plays directive and gives the role an identity, as shown in figure 6.5. To refer to the role the class uses its identity.

When a class plays a role all the non private methods of the role are added to the class interface. Thus a class can be seen either as being composed of several roles or as an undivided entity. This means that for clients of the class DefaultFigure the representations of the class in Figure 3.6 and Figure 4.3 are equivalent.

A class can reduce the visibility of the exported role members. If a class uses protected

```
public role PropertyProvider {
  private HashMap<String,Object> properties = new HashMap<String,Object>();
  public Object getProperty( String name ){
    return properties.get( name );
  }
  public void putProperty( String name, Object value ){
    properties.put( name, value );
  }
  public boolean hasProperty( String name ){
    return properties.containsKey( name );
  }
}

public role FocusSubject {
  private Vector<FocusObserver> observers = new Vector<FocusObserver>();

  public void addFocusObserver( FocusObserver obs ){
    observers.add( obs );
  }
  public void removeFocusObserver( FocusObserver obs ){
        observers.remove( obs );
  }
  protected void fireFocusGained( FocusEvent e ){
      for( FocusObserver obs : observers )
         obs.focusGained( e );
  }
}
```

Figure 6.4: Definition of the PropertyProvider and FocusSubject role (first version).

```
public class DefaultComponent implements Component {
  plays PropertyProvider propertyProvider;
  plays FocusSubject focusSbj;
  plays MouseSubject mouseSbj;
  plays BasicComponent basicComp;

  public void draw( Graphics g ){
    Border b=(Border)propertyProvider.getProperty("Border");
    if( b != null ){
      b.draw( g );
    }
    //...
  }
}
```

Figure 6.5: Definition of the DefaultComponent class (first version).

in the plays clause then all the public role methods are exported as protected. This way a class can use roles to provide an interface for their subclasses only. A class cannot change a single role member visibility.

6.2.3 Stating Role Requirements

A role does not know who their players might be but may need to exchange information with its player, so it must require the player to have a particular interface. We do that using a requirements list. The list can include required methods from the player but also required methods from objects which the role interacts with. In the list the role states the method's owner and the method's signature. The requires statement has the following syntax:

```
requires supplier implements methodSignature;
```

To indicate that the owner is the player we use the Performer keyword. Performer is used within a role as a placeholder for the player's type. Besides being used in the requirements list, it also enables roles to declare fields and parameters of the type of the player.

With this requirement list we could develop a generic subject role. The subject role can define the observer type with a generic and the concrete type of the observer is defined by the player class. The role must require that its observers implement an update method. This generic subject is shown in figure 6.6. Note that the observer type does not need to implement a specific interface. When the player class defines the concrete observer type the compiler verifies if that type conforms to the role requirement list using structural dependency rather than type dependency.

6.2.4 Playing the Same Role More Than Once

A class can play a role more than once as long as there are differences between role instances. For example the GenericSubject role could be played more than once, as long as the observer type is changed between instances. If the figure was a subject of a FigureObserver and a FigureHandleObserver it could play both roles using

```
plays GenericSubject<FigureObserver> figureSbj;
plays GenericSubject<FigureHandleObserver> figHandlerSbj;
```

6.2.5 Renaming Role Methods

Consider the PropertyProvider role. It assumes that a property is identified by a name and that name is a String. It would be more reusable if it used generics for the property type.

```
public role GenericSubject<ObserverType> {
  requires ObserverType implements void update();

  private Vector<ObserverType> observers = new Vector<ObserverType>();

  public void addObserver( ObserverType o ){
    observers.add( o );
  }

  public void removeObserver( ObserverType o ){
    observers.remove( o );
  }

  protected void fireChanged( ){
    for( ObserverType o : observers )
      o.update();
  }
}
```

Figure 6.6: A generic subject role requiring its observers to implement an update method (first version).

We can also use a generic type to specify the value type, of type Object. After a closer look, the property provider is in fact a map that maps keys to values. We could reuse a map implementation if we inherited from a Map class, but that would be conceptually wrong. Our class is not a map: it plays the role of a property map. The right way is to develop a role that is a Mapper. The only thing that prevents this are methods names. A more general Mapper cannot use names like getProperty or hasProperty, because these are only associated with properties.

We propose a renaming method that allows an easy configuration of methods' names. Each name may have three parts: a configurable one and two fixed. Both fixed parts are optional thus leaving the name of a method to be fully configurable by the class. The configurable part is bounded by # as shown next.

```
fixed#configurable#fixed
```

The configuration of the name is done by the class that plays the role in the plays clause, using the syntax:

```
plays roleType( configurable = nameToUse ) roleId;
```

We can then build our Mapper role using this renaming strategy. For the PropertyProvider we would use the following plays clause to configure the Mapper:

```
plays Mapper<String,Object> (Thing = Property)  propertyProvider;
```

The DefaultComponent class retains its original behavior but we now have a role that is more generic and reusable than the one we started with. In fact the role is able to be reused in all situations that the PropertyProvider would be used and many more. We

```
role Mapper<KeyType, ValueType> {
  private Map<KeyType,ValueType> map;

  Mapper( ) {
    map = new HashMap<KeyType,ValueType>();
  }

  Mapper( Map<KeyType, ValueType> map) {
    this.map = map;
  }

  ValueType get#Thing#( KeyType name ){
    return map.get( name );
  }

  void put#Thing#( KeyType name, ValueType value ){
    map.put( name, value );
  }

  boolean has#Thing#( KeyType name ){
    return map.containsKey( name );
  }
}
```

Figure 6.7: Definition of the Mapper role, that replaces the PropertyProvider role, with configurable methods (second version).

used the same approach to the CompositeParent and came up with a Container role (see a first version in figure 6.10). We must state the fact that reusing the Mapper role for the PropertyProvider role is an implementation choice and does not invalidate the modeling design or even the documentation.

6.2.6 Providing Multiple Versions of a Method

It's possible to declare several versions of a method using multiple definitions of the configurable name. This way, methods with the same structure are defined only once. For this feature to be used we must use a configurable called method inside a configurable role method. We must name the called method after the method it is called from. This is done using a dot name, where the configuration name before the dot is the configuration name of the outer method.

```
void role#Method#( ){
    performer.called#Method.inner#( );
}
```

This way the compiler knows that both methods are to be used together and can check if one configuration name has the same number of configurations of the other and it also checks that they are defined sequentially in the plays clause.

```
public role GenericSubject<ObserverType, EventType> {
  requires ObserverType implements void #Fire.update#( EventType e );

  public void add#Observer#( ObserverType o ){
    observers.add( o );
  }

  public void remove#Observer#( ObserverType o ){
    observers.remove( o );
  }

  protected void fire#Fire#( EventType e){
    for( ObserverType o : observers)
      o.#Fire.update#( e );
  }
}
```

Figure 6.8: Definition of the generic subject role (second version) now with configurable methods.

We could then build our generic subject role using this renaming strategy, so it no longer relies on an update method. We can also configure the addObserver method so we can use an appropriate name. We also added an event parameter to the update method. Our enhanced version of the role is shown in figure 6.8.

We can now use the generic subject role to be used either as a focus subject or as a mouse subject, providing all the notification methods. The focus subject would be configured like:

```
plays GenericSubject<FocusObserver,FocusEvent>
  ( Fire = FocusGained, Fire.update = focusGained,
    Fire = FocusLost,   Fire.update = focusLost,
    Observer = FocusObserver ) focusSbj;
```

6.2.7 Making Use of Naming Conventions

Another feature of our renaming strategy is the class directive. When class is used as a configurable part it will be replaced by the name of the player class. This is useful in inheritance hierarchies because we just need to place the plays clause in the superclass and each subclass gets a renamed method. It does imply that calls will rely on naming conventions.

One such case is the Visitor pattern. This pattern defines two roles: the Element and the Visitor. The Visitor declares a visit method for each Element. Each Element has an accept method with a Visitor as an argument that calls the corresponding method of the Visitor.

Visitor's methods usually follow a naming convention in the form of visitElementType.

```
role VisitorElement<VisitorType> {
  requires VisitorType implements void visit#visitor.class#( Performer t );

  void accept#visitor#( VisitorType v ){
    v.visit#visitor.class#( performer );
  }
}

class DefaultFigure {
   plays VisitorElement<FigureVisitor>( visitor = Visitor ) visit;
   // ... rest of class code
}

class LineFigure extends DefaultFigure {
    // no Visitor pattern code
}

interface FigureVisitor {
    void visitLineFigure( LineFigure f );
    void visitTextFigure( TextFigure f );
    //...
}
```

Figure 6.9: The VisitorElement role, a class Figure that plays the role, a subclass from the Figure hierarchy and the Visitor interface.

We used this property in our VisitorElement role, as shown in figure 6.9. The example shows it being used in a Figure hierarchy with figures as Elements. It also shows that Figure subclasses do not have any pattern code, because they will get an acceptVisitor method that calls the correct visit method.

6.2.8 Roles Playing Roles or Inheriting from Roles

Roles can play roles but can also inherit from roles. A role cannot inherit from a class and vice-versa. When a role inherits from a role that has configurable methods it cannot define them. When a role plays another role it must define all its configurable methods, but can have its own configurable methods.

For example, managing observers is a part of a more general purpose concern that is related to containers. We can say that the subject role is an observer container and develop a generic container role and make the subject inherit from the container. We can therefore reuse the container role already mentioned for the CompositeParent. We can also develop a FocusSubject that plays the GenericSubject role (see figure 6.10).

```
public role GenericContainer<ThingType> {
  private Vector<ThingType> ins = new Vector<ThingType>();

  public void add#Thing#(ThingType t){
    ins.add(t);
  }

  public void remove#Thing#(ThingType t){
    ins.remove(t);
  }

  protected Vector<ThingType> get#Thing#s(){
    return ins;
  }
}

role GenericSubject<ObserverType,EventType> extends
         GenericContainer<ObserverType>{
 requires ObserverType implements void #Fire.update#( EventType e );

  protected void fire#Fire#( EventType e ){
    for( ObserverType o : get#Thing#s( ) )
      o.#Fire.update#( e );
  }
}

public role FigureSubject {
  plays GenericSubject<FocusObserver,FocusEvent>
   ( Fire= FigureChanged, Fire.update= figureChanged,
     Fire= FigureMoved,   Fire.update = figureMoved,
     Fire= FigureRemoved, Fire.update= figureRemoved,
     Thing = FigureObserver ) figureSbj;

}
```

Figure 6.10: Roles extending roles and roles playing roles.

```
public role GenericContainer<ThingType> {
 private List<ThingType> ins;

 public GenericContainer( ){
   ins = new Vector<ThingType>();
 }
 public GenericContainer( List<ThingType> container ){
   ins = container;
 }
 // ...
}

public class CompositeComponent extends DefaultComponent {
  plays GenericContainer<Component>( Thing = Component )
          components( new ArrayList() );
  // ...
}
```

Figure 6.11: Final version of the Container role now supporting constructors. The CompositeComponent plays the GenericContainer role configuring it to use an ArrayList as the container.

6.2.9 Role Constructors

We may need to parametrize roles. In our container role we may want the container to be an ArrayList instead of a Vector. We do that allowing roles to have constructors, as shown in figure 6.11. We support role constructors but we do not allow role instantiation. To initialize a role we can use the plays clause like (also see figure 6.11 for an example)

```
plays role( params );
```

If the role needs to be initialized by the player using fields from the player's own constructor(s) this is also supported. For this JavaStage uses an analogous mechanism used by Java to call the superclass constructor. In Java to call the superclass constructor we place a super() call in the first line of the subclass constructor. In JavaStage we use the role identity to represent the role constructor. It must follow the call of super, if it is present. When playing several roles the calling order of the roles constructors is irrelevant, as long as they are contiguous, like in

```
SomeClass( Type1 param1, Type2 param2 ){
    super( param1, param2 );
    roleID1( param1 );
    roleID2( param1, param2, 10);
}
```

6.2.10 Conflict resolution

The methods defined in the class always take precedence over the methods defined in the roles. Role methods override the class inherited methods. Conflicts may arise when a class plays roles that have methods with the same signature. When conflicts arise the compiler issues a warning and the developer can handle the conflict by redefining that method in the class and calling the intended method. This is not mandatory because the compiler uses, by default, the method of the first role declared, following the plays clause order. This may seem like a fragile rule, but we believe that for most situations it will be enough. We argue that, even if a conflicting method is later added to a role, the compiler does issue a warning so the class developer becomes aware of the situation and can solve it. The important part is that the class composer can solve the situation as he wishes and not as imposed by the role or superclass' developers.

6.2.11 The self problem and delegation

Roles can be seen as a form of delegation (see also the implementation section) where the class delegates the work on the role. While this view is possible roles are not really delegation in its pure form. When a class calls a role method it passes the control to the role, so it is forwarding the call to the role. In the role the this reference will always refer to the role instance and not to the player instance. To refer to the player the role uses the performer keyword instead.

In languages without delegation, when we want to use callback methods (i.e. call methods of the delegator from the delegate) we must pass the delegator as a parameter or otherwise make the delegate know the delegator. The role can call methods on the player using performer, so the implementation of delegation using roles does not need to pass the delegator to use callback methods.

An advantage of roles over delegation, at least in Java like languages, is that we do not need to write the delegation methods. This reduces significantly code replication if the delegate class is used often or if it has a large interface. On the other hand we cannot remove methods from the interface. Another advantage JavaStage offers is the renaming mechanism where we can define the interface and are not limited by the delegate interface.

When using delegation the class can always use the delegate instance as a parameter or a return type in methods. With roles we cannot use this as roles do not define a type. On the other hand there is a special interface between the player and the role that does not exist between delegate and delegator. A player can access protected members of the role and vice-versa, while the delegator can only use the public members of the delegate.

6.3 Implementation

In this section we briefly describe how we implemented our version of roles and the design decisions we made along the way. We do not claim that the presented implementation is the best implementation possible, and we are aware of some of its limitations. Nevertheless we feel that the proposed implementation is powerful enough to reach its goals and to present developers with a valid development tool.

Supporting our role strategy could be done in various ways. We opted for a version using inner classes. This decision not only supports all our options for the approach but also ensures that no performance penalty is introduced in the final code while maintaining the final code executable in existing virtual machines.

When a class plays a role, the role code is copied to the class as an inner class. There are no conflicts with other class names as we use names not allowed by the compiler. Figure 6.12 shows an excerpt of how a class playing the FigureSubject configured as indicated in figure 6.10 would look like after roles were added.

The use of the # character in the inner class name guarantees that there isn't a name clash between synthetic classes and developer's code. The use of the identity in the class name guarantees that no conflicts may arise when playing the same role twice. For example, the name of the class for the FigureObserver role is GenericSubject#figureSbj and for the FigureHandleObserver role would be GenericSubject#figHandlerSbj.

We can see from figure 6.12, that a role is used as an object in the class. This allowed roles to have constructors. Despite this no one can directly instantiate a role, as roles are not meant to have instances.

Role methods are copied to the class interface and call the corresponding method on the role object (see addFigureObserver or fireFigureChanged). This may be seen as introducing a redirection but it is easily solved by inlining the code. No performance loss is introduced with our version of roles.

A role that is configurable is always copied to the player class and is not compiled to bytecodes. Doing so would need to change the Java bytecodes and we wanted to minimize changes to the language. Also it ensures that a class compiled with JavaStage can be run on every existing virtual machine.

Those roles that have no configuration or requirements are compiled to bytecodes. A player class does not get those roles as an inner class, just a role object and a copy of the non-private methods.

```
public class AbstractFigureRaw implements Figure {
  public void moveBy(int dx, int dy) {
    figureSbj.fireFigureChanged();
  }

  private class GenericContainer#figureSbj {
    private java.util.Vector<FigureObserver> ins =
              new java.util.Vector<FigureObserver>();

    public void addFigureObserver( FigureObserver t){
      ins.add( t );
    }

    protected java.util.Vector<FigureObserver> getFigureObservers(){
        return ins;
    }
  }

  private class GenericSubject#figureSbj
                    extends GenericContainer#figureSbj {
    protected void fireFigureChanged( ){
       for( FigureObserver o : getFigureObservers() )
                o.figureChanged( );
    }
    // ... other fires
  }

  private GenericSubject#figureSbj figureSbj  =
             new GenericSubject#figureSbj();

  public void addFigureObserver( FigureObserver t ){
    figureSbj.addFigureObserver( t );
  }

  protected void fireFigureChanged( ){
    figureSbj.fireFigureChanged( );
  }

  protected java.util.Vector<FigureObserver> getFigureObservers(){
    return figureSbj.getFigureObservers( );
  }
}
```

Figure 6.12: Excerpt of how an AbstractFigureRaw playing the GenericSubject role class would look

6.3.1 Role Identity

A recurrent question is role identity. Some argue that roles have no identity [Kri95, Kri96] because they are not independent entities. The object and its roles are referenced as one. Others argue that roles have an identity, different from its intrinsic [Odb94, Tru04]. With role identity classes can have multiple instances of the same role and distinguish each one. There are also those to whom roles share the same identity with its intrinsic and also have one that distinguishes it from other roles in the intrinsic [SSSM95].

Our roles have an identity associated with the player. The identity is given by the player in the plays clause. Whenever the player access role members it uses this identity. If the role is public then its identity is accessed just like any class member. This allows clients, which know the roles the class plays, to select the role they want. Please note that while this is possible we consider that a class should not declare their roles as public, for the same reasons it should not expose fields as public.

The major reason to provide roles with an identity is to add state to roles. Since each role field must be accessed using the role identity there never is a name conflict between fields of different roles.

Role identity is also used to distinguish between roles when resolving a conflicting method. We use the identity to specify which role we want to access. This is better than using class names, because we can change the role hierarchy and still be able to maintain the code unchanged. It also helps to decide between role methods and superclass methods, where super would refer to the superclass and the identity to the role.

6.3.2 The plays Clause

Should not the plays clause be considered equivalent to the extends or implements clauses and be placed accordingly? After all it does have an impact in the class interface. There are in fact several reasons for not doing so. One is the role identity which, purposely, resembles an object declaration (see the implementation section). Yet another reason is the naming configuration, which would clutter that declaration. A final reason is role initialization, as roles may have non default constructors. It would be awkward to do these configurations in an implements-like declaration.

6.3.3 Role Inheritance vs Role Playing Roles

Roles and interfaces are somewhat related [Ste01] so we could let role types have the polymorphic behavior interfaces have. We could write code that would work with any class that plays that role. Our method renaming strategy, however, forbids this because

the actual interface a role provides is configured by the class and not by the role itself. Roles that do not use the renaming strategy could in fact define an interface.

In our approach, where roles can inherit from other roles, making roles as types is a simple step and would clearly distinguish when we would make a role inherit from another role or simply play another role. That distinction is clear when a role has configurable methods. If we want to configure the role we must play it, when we want to extend the role without configuring any of the configurable methods we extend it. When a role has no configurable methods inheriting from it or playing it is a semantic choice only. We still haven't made a decision on whether there are advantages in considering roles as defining a type. We will defer that decision to future work.

6.3.4 Aliases vs Method Renaming

Traits use aliases to rename a method within a class to solve conflicts. Because we solve those conflicts using the role identity there is no need to use aliases. Our renaming mechanism objective is to enhance the reusability of roles. With it we can:

- configure a role, with meaningful method names, in the context of the player class;
- have fast renaming of several methods;
- provide multiple versions of methods.

6.3.5 Requirements Listing

Requirements in most role languages are made by a playedBy clause that states which classes can play it. Since our roles are intended to be played by any class this would restrict role reusability. The role must, nevertheless, impose some restrictions on the player interface when it needs to communicate with it. In our case those restrictions are imposed via the requirement list. The inversion of declaration from the role to the class allowed us to provide a renaming mechanism that configures the role as it should be used. This statement also allows us to impose restrictions not only in the player itself but also on other classes that are part of the interaction. For example, in our subject role, we can configure which interface the observer must implement.

6.4 Limitations

As every language JavaStage has its limitations. We describe here the most relevant ones.

6.4.1 Source Code Must be Available

Some roles in JavaStage cannot be compiled to bytecodes, due mainly to three reasons:

- requiments list;
- configurable methods;
- multiple versions methods.

One of the obstacles is the requirement list. To play a specific role the player class must adhere to a specific interface dictated by the requirements list. But the required interface is nameless. We could work around the naming problem if we used a synthetic interface for each type from which the roles requires a method. This way the role could be compiled, even thought it would increase the system's size.

But there is a further problem with requirements: they may contain configurable methods. In such case the correct interface is known only when compiling the class and is not available when compiling the role. To cope with this we would need to embed that information in the bytecodes of the role. We do not want to change the bytecodes specifications, so JavaStage compiled applications can be executed in every JVM, so we cannot compile the role code and it must be available.

The third problem is related to the second in which using multiple versions methods the required interface is known only when compiling the player class. Again the only solution would be to change the bytecode configuration. But multiple versions raises another question: when creating the corresponding inner class the role methods are copied and then modified to fit the player class configuration. For ease of compiler development this copy requires the source code to be present as it is done based on the AST of the role.

For these reasons the roles that have configurable methods or requirements are not compiled and their source code must be made available. This is an analogous situation with the use of templates in C++, where template code is provided in the form of source code.

6.4.2 No static public Variables

JavaStage uses inner classes as its implementation strategy and Java does not support inner classes to have static fields unless the inner class is also static. To overcome this we declared the static private variables in the player class. To cope with playing the same roles multiple times we changed the name of the static variable to include the role identity, as we have done to the inner class name (see section 6.3).

The same solution cannot be applied to the public static fields. Since they are public clients can assess them but when a class plays the same role more than once the static

fields retain the same name and a conflict arises. Clients accessing the same field would need to know which roles the class plays and use their identity to disambiguate between the fields. Played roles are not always known by clients, and we consider bad practice to do so, because it violates encapsulation principles. Forcing class developers to make their roles public to work around this problem did not seem a good enough reason to break encapsulation, so we opted not to support public static fields.

6.5 Implementation Alternatives

When developing JavaStage we had to choose between several ways of implementing the infrastructure to support JavaStage role features. Of the several options, we will describe two that we consider could have been selected.

6.5.1 Using Reflection

One of the first attempts to develop JavaStage recurred to the use of reflection. The intent was to provide every role with a compiled file (a .class file). Reflection was used to implement the renaming mechanism and supporting the multiple version method.

In this version each role configurable method ”real” name was the one with the # in it. The methods were only really renamed in the class. This required no great effort to implement. But it could only be applied to the role methods directly. If the role method called a configurable from the player or from another participant this solution did not work.

When a role method called a player configurable method the actual call was made using reflection. Each role object had a lookup table for each configurable method it called. Each entry in the table is the method object associated with each configurable method. In the role code it would call the method based on the table index supplied when the role method was called. When creating the role object the class filled that lookup table with the configurations that suited its needs. Each class method then called the role method, using as a parameter the index of the method to call in the lookup table.

This solution involves modification of the role methods in order to introduce the extra parameters needed. These parameters were the object making the call (the this reference in the class) and the index of the methods to call. These modifications were transparent to the programmer but introduced a complexity in the compiler and a number of indirections in the code (the class code calls the role code that, via reflection, then called a class method) that were not straightforward. When making roles playing other roles this would create a web of indirection with very negative effects on performance.

Inheritance also complicated the solution because a role that inherited from another role would need to manage the complete lookup table. So in spite of working this alternative introduced a series of problems that eventually lead it to be discarded.

The main reason for abandoning this implementation alternative was performance. Using reflection has a negative impact on performance. It also had negative effects on the memory usage because of the lookup table. Another requisite that it needed was a performer variable for the role to maintain the reference to its player. In the inner class version we used the performer is simply the outer class' this reference.

Further reasons included the requirement lists. For a complete compilation we would need to code the requirements list and that would need to change the virtual machine class loader. Our solution would therefore not be usable in every existing JVM and that was another major drawback of this solution.

6.5.2 Roles as Standalone Classes

Another alternative to using inner classes was to use roles as standalone classes. We could use the Chai [SD05], a Java extension to implement traits, way of implementing traits. They used traits as standalone classes in their implementation. Traits also can impose requirements on the class that uses them. This is done creating a synthetic interface from the role requirements. When a class uses a trait it automatically implements that interface. Chai also allows the renaming of traits methods by the class, but that is done only in the class, the trait method is not really renamed, only the class method that calls the trait methods gets renamed.

In our case we would need to capture several interfaces because our roles can make requirements not only to the player class but also to other parties. We would need those parties to implement a synthetic interface too, and that may be impossible if all we have from those classes is a compiled class file.

For a complete compilation we would need to code the requirements list and that would need to change the virtual machine class loader. Our solution would therefore not be usable in every existing JVM and that is the main reason we dismissed this alternative.

We would also have to create a performer var to store the player object within the role and that would introduce a size overhead, if insignificant. Another problem with the standalone class would be the access levels. A protected level implies that the class can access role fields and vice-versa, but with standalone classes, and both classes being unrelated this level would be impossible to implement in a straightforward manner.

6.6 Comparison with Other Approaches

In this section we compare JavaStage with related approaches. The order in which the they appear depends on the proximity we feel they are to our approach. In this regard the Traits approach is the one we feel is the closest to ours so it is the first and we also dedicated it the more time and effort. The roles approaches for example, and even though they use roles, are not quite related to ours as they use dynamic roles and intend to extend objects while we use static roles and aim at composing classes.

6.6.1 Traits

Traits, as discussed in section 3.4, can be seen as a set of methods that provide common behavior. When a class uses a trait its methods are added to the class. The class also provides glue code to compose the several traits. Traits cannot store state. State is maintained by the class that uses the trait. For comparing roles and traits [BA13a] we follow a few key points that both approaches must deal with and describe how each handled the situation.

Unit of composition

In roles the unit of composition is the role while in traits it is the trait.

Inheritance

Roles and traits are targeted for single inheritance languages so there is no multiple inheritance support. Roles can play other roles and traits can use other traits. Both approaches also support a class using the same unit several times. In a class, to access the features of the superclass both approaches use the super keyword. In a role, however, the super keyword refers to the super role, as roles can inherit from other roles. In a trait it refers to the superclass of the composing class.

State Support

Roles can have state and it does not cause any conflict because to access role state the class must use the role identity thus no conflicts arise. Traits do not support state. Proposals to solve this introduced a significant complexity to the trait model and encapsulation problems [CBDM09]. When modeling a concept we, often, need to express state. For example, to model a container we need a structure for storage. Forcing the composing class to supply that structure is rather breaking the container's encapsulation.

Conflict Resolution

Both approaches follow the same rules for method overriding. The class overrides methods from roles/traits and roles/traits override the class inherited methods. Conflicts may arise when methods with the same signature are provided by more than one unit. In traits the conflict must be resolved explicitly while in roles the method of the first played role is used (there is a compiler warning). In both cases it is the class composer that decides which method to use. In traits he can choose to exclude some methods so there is no conflict or he can redefine the method and use aliases to refer to each of the conflicting methods. In roles there is no exclusion and the class composer must redefine the conflicting method if he wishes to override the rule of using the method of the first role.

Composition Order

The order in which traits are composed is symmetric so order of composition is irrelevant. The same applies to roles when there are no conflicting methods. When there are conflicting methods the order of the plays will dictate which method is used. This, however, is not mandatory as discussed in the previous topic.

Method Renaming vs Aliases

There is a fundamental difference between aliases in traits and method renaming in the roles. The traits aliases are used only by the class for distinguishing conflicting methods, the class interface is not affected. In roles the renaming affects the class interface. This means that a class may be able to tailor its interface to suit its needs and not be limited by the role interface. The renaming mechanism of roles also allows renaming several methods in one go, while aliases in traits are made one by one. Roles renaming scheme can provide multiple versions of a method. Traits aliases can be applied to any method, while on roles only the configurable methods can be configured.

Flat and Composite view

Both approaches support a flat view of the class as well as a composite view. Thus a class can be seen as a set of methods, the flat view, or as being composed of several units of composition, the composite view. The class interface in both views is exactly the same. The main difference between the two is that a trait method is seen just like a class method, and a role method is always a role method and each reference to other methods will always refer to role methods. For example, suppose a trait that defines the methods foo and bar, where bar calls foo. If the class overrides the foo method then the trait bar method will call the foo method on the class not on the trait. The same situation is handled differently

by roles. If the method bar of the role is called then it will call the foo method on the role and not on the class. For a role method to call a class method it must do it explicitly using the perfomer keyword.

Visibility control

Traits have no visibility control. Freezable traits [DWBN07] compensate this by allowing classes to freeze/unfreeze methods, i.e., declare a method as private (freeze) or making it public (defrost). But there is no way to express access constraints between class and trait. For example, fields should be accessed directly only by the owner's code. Traits do no support this. Roles on the other hand support all Java access levels, so a specific interface between role and class is possible.

Stating Requirements

The use of generic types is a useful feature in most languages, especially for dealing with object collections. Traits can require methods from the class that uses them, but cannot impose restriction on generic types it interacts with. The requires statement of roles indicates the method signature and which type it is required from. This allows roles not only to require methods from the class but also from other collaborators types.

6.6.2 Aspect-Oriented Programming

Aspect-Oriented Programming as used in AspectJ [KHH+01] is another approach that tries to modularize crosscutting concerns as discussed in section 3.3. In [KHH+01] the authors define pointcuts to identify points in the executing program that may trigger a different execution path and advices that indicate the new execution path. While the modularization of crosscutting concerns is the flagship of AOP several authors disagree [Ste06, Prz11]. The effects of pointcuts and advices, especially when several aspects have similar pointcuts, may be unpredictable. A particular problem is the fragile pointcut [KS04]. This problem arises when simple changes made to a method code make a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the necessary advice. Thus simple changes in the class code can have unsought effects [KAB07].

The obliviousness feature [FF00] means that a class is aspect unaware so aspects can be plugged or unplugged as needed. This somewhat resembles dynamic roles and explain why aspects are used in dynamic role languages. But it introduces problems in comprehensibility [GSS+06]. To fully understand the system we must not only know the classes but also have to know the aspects that affect each class. This is a major drawback

when maintaining a system, since the dependencies are not always explicit and there is not an explicit interface between both parts.

With our approach all dependencies are explicit and the system comprehensibility is increased when compared to the OO version [RG98]. We do not, however, have the obliviousness of AOP because the class knows and is aware of the roles it plays. Any changes to the class code are innocuous to the role, as long as the contract between them stays the same. As a final point we do not believe that our approach can replace AOP. We believe that for modeling static concerns our approach is more suitable while AOP is better suited for pluggable and unpluggable concerns. A case study comparing the two approaches would be useful to ascertain this point.

We can compare the AOP implementation of the Subject pattern proposed by Hannemann and Kiczales in [HK02] with ours. Even though it is rather unfair to compare just this pattern, it can help to pinpoint some key differences. In the AspectJ Observer the aspect uses an update method only, while in ours we can use as many update methods as we wish, without effort, due to the multiple version feature. The method in AspectJ is also limited to the name update, because AspectJ does not allow method renaming. In our version the name of the update method is configurable. In AspectJ the storage of the observers is made by a different entity that uses a map with the subject as a key to retrieve all of its observers, while in our version the observers are stored directly in the role. The granularity of the versions is also different. In AspectJ the pointcut are methods that must be explicitly defined, and later changes can render this ineffective. On the other hand the class code is completely free of the aspect code. In our version the class code must call the firing methods of the role whenever the event occurs. This gives the class developer a finer degree of control but the class code must acknowledge the role.

From the discussion in section 3.3.3 we can see that there is still code replication between the several aspects presented there, namely in storing and managing children/observers. We dealt with those by declaring a Container role that could be used in both cases.

6.6.3 Other Composition Techniques

Feature-Oriented Programming

FOP relies on a step-wise refinement of applications by adding new features or refining existing ones. To compose a system we just state which features it has. The composition is made automatically with tool support, like AHEAD [BSR04]. This is a more powerful technique than JavaStage. AHEAD uses several tools for composing the code and extra files for configuring the composition step. JavaStage is a programming language that statically composes classes using only source code. From the point of view of a programmer,

JavaStage is more simple to work with and tool free (except the compiler). But JavaStage can also be used together with AHEAD and be seen as a complementary tool to the FOP technique. Thus the use of AHEAD does not exclude the use of Roles.

AHEAD can be used to compose classes and later we can construct several refinements to that class. This is where we can use JavaStage together with AHEAD. Refinements may be similar between some classes and, in AHEAD, we would duplicate that code in all refinements. With roles we just place the refinement code in a role and then all refinements just play the role, thus preventing code replication. To avoid this duplication we could use Mixins [SB02], but roles offer more ways of configuration and do not have mixins limitations like a linear composition order.

Multiple Dimension Separation of Concerns

In MDSOC concerns are placed in hyperslices that are composed together in hypermodules following a set of composing rules. Hyperslices may be used by many hypermodules. Hypermodules may be reused and can contain other hypermodules. Like FOP/AHEAD this is a much more advanced tool than JavaStage. And just like FOP, JavaStage can be used within MDSOC as a complementary tool.

For example, in the figure framework, a figure could be decomposed in several hyperslices, one for each concern. We could have the properties hyperslice, the observer hyperslice, etc. The Figure hypermodule would compose the several hyperslices. Extensions to the application can be made by adding new hyperslices and composing them in a new hypermodule. For example, if we wish to save the figures in an SVG or any other format we need to add a hyperslice for each format defining how each figure should be saved in that format.

Nevertheless code replication is still found between hyperslices. Several hyperslices that could implement observers would have code replication. We could reuse our GenericSubject role in those hyperslices. In MDSOC we cannot develop a full purpose Subject role like we did in JavaStage as it does not support multiple method versions and method renaming. MDSOC does not exclude the use of roles as they can add yet another form of composition to its tools. MDSOC also relies heavily on tools and configuration files, which can be overkill, while our approach is code based only.

Package Templates

Package Templates (PT) [KMPS09] combine the use of packages and templates. Classes defined in these packages are directly available when the package is instantiated. When instantiated, classes can be tailored to the use context by getting additions, renaming elements and type parameters can be given actual types. This tailoring is similar to ours

as we also support renaming and type parameters. PT may also impose restrictions on type parameters using a constraints declaration that resembles our requirement list.

Classes in a PT can be merged with classes from other PT and can be used more than once in the same merging operation (like roles can be used multiple times). To avoid multiple inheritance problems PT imposes a series of restrictions on inheritance inside a PT. These, however, can be overcome by simple workarounds.

Name clashes are solved via renaming because both fields and methods can be renamed. The renaming cannot be used on the constraints, which means that intermediate classes may be needed to provide these constraints and/or methods added to classes just to meet that requirement.

The main differences between PT and roles are: PT relies on inheritance for the merging and roles rely on inner classes and forwarding methods, so JavaStage does not need the PT restriction on inheritance and still allows multiple role playing; JavaStage allows the renaming of required methods, so it does not need intermediate classes. In PT a class may be the result of the merge of several classes. To refer to one of those classes PT uses the construct super(MergedClassName) while we use the role identity. The use of the class name is more fragile than the identity, because the identity is given by the class composer and the class name is given by the PT and not the instantiator.

Caesar

Caesar [MO03] uses aspect technology to modularize crosscutting concerns and enhance the reuse of aspects leading to a greater reduction of repeated code. Caesar uses an Aspect Collaboration Interface that decouples aspects binding and implementations by defining them in a separated module. Caesar does not allow method renaming.

We can compare our Subject role with the subject role in [MO03]: our role subject has fully configurable methods names while in Caesar all subjects must have a addObserver method. In Caesar if we want a class to be a subject for two different actions (e.g. MouseListener and MouseMotionListener) we must define a binding class for each action, while with our approach we can do the configuration in the class alone. Lastly in Caesar observers are limited to a single notification method, named update. This limits the observer pattern to report only a change, whereas in ours we can define several notification methods each with a meaningful name.

Jiazzi

Jiazzi [MFH01] is based on Units [FF98] and aims at building systems out of reusable components integrated with the language. Jiazzi has two types of units: Atoms (composed of java classes) and Compounds (composed of atoms or other compounds). Jiazzi supports

the addition of features to classes without editing their source code, something that JavaStage cannot do and it is not designed for.

Nevertheless, roles can be used to specify the features to be added. Furthermore, a role could be used to add the same behavior for different classes in the same unit (used as a way of emulating multiple inheritance within a unit), or for the same class but in different units whenever those units are not composable together.

Open Classes

Open classes as used by MultiJava [CLCM00, CMLC06] allow external methods to be added to a class, without changing the class. They also support multi method dispatching. They are used to extend a class interface like our approach but from a different perspective: we focus on constructing the class and open classes on adding methods to existing classes.

The approaches are complementary because roles can be used to compose the class and MultiJava can later add new methods. Roles could also be used to add those new methods. Also MultiJava cannot redefine class methods, and while role methods do not redefine the class's defined methods they can redefine the class's inherited methods.

6.6.4 Role Related Approaches

Chernuchin and Dittrich [CLD05] compared five approaches for role support in OO languages. They were multiple inheritance, the role object pattern, interface inheritance, object teams and roles as components of classes. They used criteria such as encapsulation, dependency, dynamicity, identity sharing and the ability to play the same role multiple times. Roles as components of classes compared fairly well and the only drawback, aside dynamicity, was the fact that there were no tools that supported it. With JavaStage that drawback is eliminated.

Chernuchin and Dittrich [CD05] use the notion of natural types and role types that we followed. They also described ways to deal with role dependencies which we did not consider as it would introduce extra complexity to the role language. Their role model is similar to ours in that it is an extension of the object model. Even though they suggest programming constructs to support their approach no role language has emerged.

Template Classes

VanHilst and Notkin in [VN96] proposed to use roles in the C++ language. They did not extend the language to introduce role support, instead they proposed the use of template classes to implement role like behavior. A role is implemented as a template class and

defines their collaborators via other template types. To play the role a class inherits from the template role, defining each collaborator type, so roles are a supertype of the player.

Since roles are template classes they can play other roles and inherit from other roles as in our approach. To support multiple role playing they use intermediate steps to avoid name clashes and other multiple inheritance problems.

Our roles do not rely on inheritance. Our plays declaration also solves name clashes without the use of intermediate steps. C++ does not support method renaming, like JavaStage does, thus a role must provide generic methods, which can make it impossible to use the same role several times. The C++ template engine is very powerful but its use makes their approach limited to the C++ language. Our approach seems to require fewer changes. For example, it would be easier to apply our approach in the C++ language (which supports inner classes) than supporting templates in Java.

6.6.5 Dynamic Role Approaches

This subsection is dedicated to programming languages that support roles, but on their dynamic nature. Our approach focus is on static class composition, so a direct comparison between them and our approach is not feasible.

ObjectTeams

ObjectTeams [Her05] is an extension to Java that uses dynamic roles. They introduce the notion of team. A team represents a context in which several classes collaborate to achieve a common goal. Roles are implemented as inner classes of a team. When an object is used by the team is gets attached one of the defined team roles. Whenever used inside the team the object has that role attached, whenever used outside the team the role is not considered (a process named as lifting/lowering). A role from a team cannot be reused by another, unrelated, team. Roles are limited to be played by a specific type, because of the playedBy directive.

EpsilonJ

EpsilonJ [TUI07] is another java dynamic role extension that, like Object Teams, uses aspect technology. In EpsilonJ roles are also defined as inner classes of a context. Roles are assigned to an object via a bind directive. EpsilonJ uses a requires directive similar to ours, but, unlike ours, affects only the player. It also offers a replacing directive to rename method names but that is done on an object by object basis when binding the role to the object, and does not allow block renaming. It also applies only to the names of the methods the role offers.

PowerJava

PowerJava [BSI07] also supports dynamic roles. In PowerJava roles always belong to a so called institution. When an object wants to interact with that institution it must assume one of the roles the institution offers. To access specific roles of an object castings are needed. Roles are written for a particular institution, so we cannot reuse roles between unrelated institutions.

6.6.6 Approaches using Class Extensibility

These approaches deal with the extensibility of classes. Our work's goal is not to extend existing classes but to reuse code that otherwise would be replicated in several classes. However these techniques reduce the amount of code that still is duplicated so we included them. These techniques can be combined with ours so we consider them as complementary and not as alternatives.

Classboxes

In Classboxes [BDN05] classes are defined within a kind of module, or unit of scoping. In each classbox we can define classes but can also import classes from other classboxes or refine other classes. Refinements may consist in adding or redefining new behavior or state from an imported class. Since these refinements are only visible within the classbox or classboxes that import from it, existing clients of the refined class are not affected. Roles could be used here as a way to introduce refinements. Supposing that some, related or unrelated, classes need a refinement with the behavior common to all, we can use a role to model that behavior and the refinement would be each class playing that role.

Virtual classes

Virtual classes [EOC06] are used by Caeser. A class can define nested classes. Nested classes are the virtual classes and they can be redefined by subclasses of the enclosing class. Each virtual class has therefore an enclosing object - the object of the container class or one of its subclasses. At run time, the inner class to use depends on the type of the outer object. Like with Classboxes we can use roles to add the new behavior to virtual classes.

6.7 Summary

Implementing a language extension is not an easy task, especially when we want to enhance its code reuse mechanisms and introducing new modules. But following the modularization

guidelines we were able to develop the role extension, JavaStage, that offers a powerful renaming mechanism. JavaStage also supports a role inheritance hierarchy, making roles more flexible and independent of classes.

JavaStage innovates in the manner in which it allows roles to require other participants to implement a configurable interface, at the same time allowing the player to define that concrete interface. This will be a key factor to implement a role library.

The way roles are implemented within JavaStage had a great focus on performance issues and memory usage, creating the smallest overhead possible. While the way they were implemented may not be the best possible implementation we believe that it provides a very good outcome in terms of those factors.

Chapter 7

Removing Clones

In this chapter we recall the clone types mentioned in chapter 2 and from those clones we identify clones that are not removable using current clone removing techniques. We show how roles, and JavaStage, can be used to remove them. We present four refactorings that may be used for that purpose.

7.1 Unresolved Clones

The case studies, described in chapter 9, were used as a way to gain more insights in code cloning. We used that study to identify clones that could not be removed by traditional refactorings, or cases where the traditional solution could be improved by using roles [BA13b]. We organized these clones into 4 categories:

- Identical clones in classes with different superclasses;
- Clones that have identical structures but use different, unrelated, types;
- Clones with the same structure and types but using different methods;
- Clones with the same structure that use different types and method names.

We will discuss each of these categories in the following subsections.

7.1.1 Identical Clones in Classes with Different Superclasses

We detected clones with identical code but that belonged to classes with different superclasses. An example, from the JHotDraw framework, is shown in figure 7.1. These clones could be removed using the Extract Superclass if affected classes did not have a superclass. This solution is not an option in Java, or other single inheritance languages, because we cannot create a common superclass.

To solve these clones we could use Extract Class refactoring. The downside of this particular refactoring is that it forces affected classes to create delegate methods to the new class. In automated and semi-automated refactoring tools this work is done automatically, but without these tools the work is tedious and error prone. Furthermore, when reusing the class in future situations those delegation methods need to be made again, thus leading to replicated code, nonetheless. Our solution using roles seems to be better both in reuse and code saving as we will explain later.

```
public class DrawApplet extends JApplet implements PaletteListener /*...*/{

  void paletteUserSelected(PaletteButton button) {
    ToolButton toolButton = (ToolButton) button;
    setTool (toolButton.tool(), toolButton.name() );
    setSelected( toolButton );
  }

  public void paletteUserOver(PaletteButton button, boolean inside) {
   if (inside) {
     showStatus( button.name() );
   }
   else if (fSelectedToolButton != null) {
     showStatus( fSelectedToolButton.name() );
   }
  }
  // ... rest of class code
}

public  class DrawApplication extends JFrame
   implements PaletteListener /* ... */ {

  /* this class has the paletteUserSelected and
   * paletteUserOver methods identical to the
   * DrawApplet class                          */
}
```

Figure 7.1: Example of a clone in classes that have different superclasses.

7.1.2 Clones That Have Identical Structure but Use Different, Unrelated, Types

We have found, especially in the Spring Framework, some clones that have the exact same structure, but use different types. If the different types have a common supertype or one is the supertype of the other, we can use that type throughout the code. If they are not related we can create a common supertype and use it. This uniforms the type between the clone code, transforming it to the same kind of clone clones as in the previous subsection. However, this may not be done for a number of reasons: the types may be unrelated and we cannot create a common type, either because they already have a supertype or because they are unrelated and such a common supertype would be conceptually wrong. Even if the types have a supertype the characteristics of the code may require each code to use its specific type. If we cannot substitute the types with a common type we cannot refactor using any of the available refactorings.

Spring has several classes that deal with portlets and servlets. Both have different class hierarchies but are used in much the same way. Figure 7.2 shows an excerpt of such a case. In the example all methods call the getRequest method. But the getRequest method returns a PortletRequest object in the PortletWebRequest class and an HttpServletRequest in the ServletWebRequest class. The request classes are not related so we cannot return a common type. We cannot even create a common type because these classes are not part of the Spring framework and have different sources as they come from different frameworks.

7.1.3 Clones With the Same Structure and Types But Using Different Methods

Another type of clones that we cannot remove are clones that have the same basic structure, use the same types, but each uses different methods. The name differences span from the clone methods' names themselves to the methods called inside the clone code. To resolve these clones we would need to refactor methods' names in order to uniform their names, but that is not always possible or desirable.

In figure 7.3 we show an example of such a clone taken from the Spring framework. In the example the classes have to either expose or ignore methods as MBean operations. Their code is pretty much the same, except for methods names.

```
public class PortletWebRequest  extends PortletRequestAttributes
                                implements NativeWebRequest {
  public Locale getLocale() {
    return getRequest().getLocale();
  }
  public String getContextPath() {
    return getRequest().getContextPath();
  }
  public String getRemoteUser() {
    return getRequest().getRemoteUser();
  }
}

public class ServletWebRequest extends ServletRequestAttributes
                               implements NativeWebRequest {

  public Locale getLocale() {
    return getRequest().getLocale();
  }
  public String getContextPath() {
    return getRequest().getContextPath();
  }
  public String getRemoteUser() {
    return getRequest().getRemoteUser();
  }
}
```

Figure 7.2: Example of a clone in classes that use different types.

7.1.4 Clones With the Same Structure That Use Different Types and Method Names

These Type II clones have the same structure but the types, methods' names, and called methods' names are all different. Such clones cannot be resolved using neither technique in the refactoring catalog.

An example can be found in JHotDraw, where Tool and Command objects must inform their listeners of their actions. An excerpt of these classes is shown in figure 7.4.

```
class MethodNameBasedMBeanInfoAssembler
           extends AbstractConfigurableMBeanInfoAssembler {

  private Set<String> managedMethods;
  private Map<String, Set<String>> methodMappings;

  void setManagedMethods(String[] methodNames) {
     managedMethods = new HashSet/*...*/;
  }
  void setMethodMappings(Properties mappings) {
    methodMappings = new HashMap</*...*/>();
    /* ... */
  }
  boolean includeReadAttribute( Method method, String beanKey) {
    return isMatch(method, beanKey);
  }
  boolean includeWriteAttribute( Method method, String beanKey) {
    return isMatch(method, beanKey);
  }
}

class MethodExclusionMBeanInfoAssembler
           extends AbstractConfigurableMBeanInfoAssembler {

 Set<String> ignoredMethods;
 Map<String, Set<String>> ignoredMethodMappings;

 void setIgnoredMethods(String[] ignoredMtdNames){
    ignoredMethods = new HashSet/*...*/;
 }
 void setIgnoredMethodMappings(Propertiesmappings){
   ignoredMethodMappings = new HashMap</*..*/>();
   /* ... */
 }
 boolean includeReadAttribute( Method method, String beanKey) {
   return isNotIgnored(method, beanKey);
 }
 boolean includeWriteAttribute( Method method, String beanKey) {
    return isNotIgnored(method, beanKey);
 }
}
```

Figure 7.3: Example of a clone that uses different methods but same types.

```
abstract class AbstractCommand implements Command {

 public void fireCommandExecutedEvent() {
   Iterator iter = myRegisteredListeners.iterator();
   while (iter.hasNext()) {
         ((CommandListener)iter.next()).commandExecuted(
               new EventObject(myObservedCommand));
  }
 }
 public void fireCommandExecutableEvent() {
   Iterator iter = myRegisteredListeners.iterator();
   while (iter.hasNext()) {
         ((CommandListener)iter.next()).commandExecutable(
               new EventObject(myObservedCommand));
  }
}

abstract class AbstractTool implements Tool {

  public void fireToolUsableEvent() {
    Iterator iter = myRegisteredListeners.iterator();
    while (iter.hasNext()) {
          ((ToolListener)iter.next()).toolUsable(
               new EventObject(myObservedTool));
   }
 }
 public void fireToolActivatedEvent() {
    Iterator iter = myRegisteredListeners.iterator();
    while( iter.hasNext() ) {
          ((ToolListener)iter.next()).toolActivated(
               new EventObject(myObservedTool));
  }
 }
}
```

Figure 7.4: Example of a clone that has different types and method names but similar structures.

7.2 Clone Removal Role Refactorings

To remove the clones identified in the previous section we sketched a refactoring for each category. We named our refactorings as: Extract Role; Extract Role Changing Types, Extract Roles with Configurable Methods; Extract Role with Types and Methods. They use similar steps but we opted to define all instead of one with several options. This way there is a one-to-one mapping between clone categories and refactorings.

7.2.1 Extract Role

You have a class that has the same code as another class and it deals with a concern that is not the class main concern.

Create a new role and move the relevant fields and methods from the old classes into the new role and make the classes play the role.

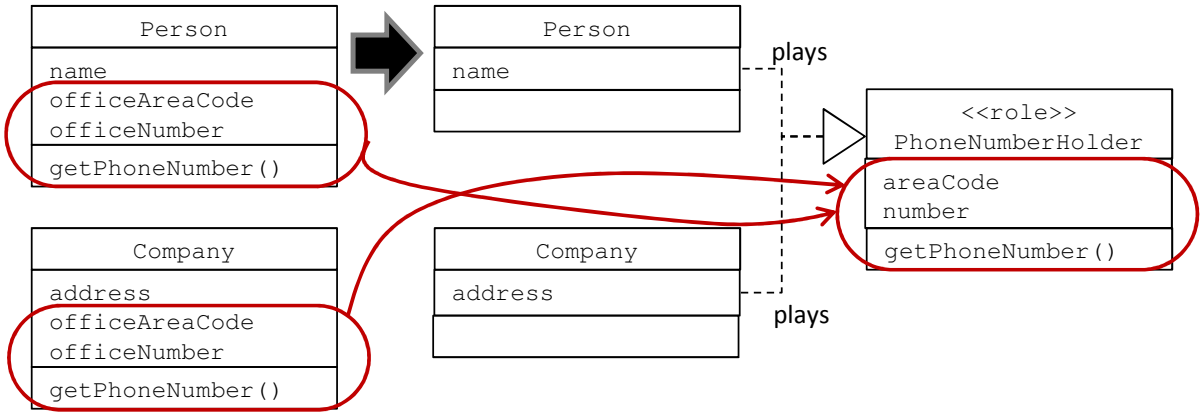

Figure 7.5: Extract role

Motivation

This refactoring can be applied to the clones of category "Identical clones in classes with different superclasses".

A class should represent a specific concept that has crisp conceptual boundaries. Nevertheless the classes grow over time and get added responsibilities that are hardly related to the initial concept. But the added responsibilities sometimes are not enough to create a class on their own.

To retain the class original concept as clear as possible the class code should deal only with that concept. You need to consider what the extra code is and, if possible, group the several added responsibilities in a few coherent sets, as independent of each other as possible.

Mechanics

- Group the responsibilities you want to separate in a coherent set.
- Create a role with a name that indicates the concern it deals with.
- Make the class play the role.
- Use MOVE FIELD on each field you wish to move.
  - try to generalize the role, sometimes the names must change as they are tuned for the class's intent and not the role's intent.
- Use MOVE METHOD on each method you wish to move.
  - try to generalize the role, sometimes the names must change as they are tuned for the class's intent and not the role's intent, if this is the case then use EXTRACT ROLE WITH CONFIGURABLE METHODS.
- If the moved methods call methods of the class add a requires statement for each called method.
  - again try to generalize the role, if the names of the called methods are generic enough then use them, otherwise try to prepare them for renaming using EXTRACT ROLE WITH CONFIGURABLE METHODS.
- Compile and test.
  - If the moved method does not call other methods to be moved compile and test after each method move.

Example

Consider the classes presented in figure 7.1. They have the same code that responds to events when the user of the JHotDraw application selects a button on the tool palette or simply hovers above it. In this case we can separate this code from the rest of the class code, because it is fairly independent of the rest of the class code. It just keeps track of which button is currently selected tool and does not affect the rest of the class behavior.

We start by building a role for this concern and make both classes play the role.

```
role DefPaletteListener {
}

class DrawApplet extends JApplet implements PaletteListener, /*... */ {
    plays DefPaletteListener palListener;
```

```
}

class DrawApplication extends JFrame implements PaletteListener, /*...*/ {
    plays DefPaletteListener palListener;
}
```

Now would be the time to move any fields to the role. In this case, however, the code to be moved does not use any fields.

The next step is to move the methods. In this case we have to move two methods. The first to move, `palleteUserSelected`, calls the `setTool` and `setSelected` of the class so we need to put them in the requirements list when we make the move. The names of the methods are mandatory as they are part of the PalleteListener interface, and we are very unlikely to encounter the same code related to other types of listeners. This means that there is no need to configure method names.

```
role DefPaletteListener {
   requires Performer implements void setTool(Tool t, String name);
   requires Performer implements void setSelected(ToolButton button);

   void paletteUserSelected(PaletteButton button) {
      ToolButton toolButton = (ToolButton) button;
      performer.setTool(toolButton.tool(), toolButton.name());
      performer.setSelected(toolButton);
   }
}
```

Now we can move the other method. It too calls methods on the class so, again, we need to place them on a requires list. The final role would be

```
role DefPaletteListener {
   requires Performer implements void setTool(Tool t, String name);
   requires Performer implements void setSelected(ToolButton button);
   requires Performer implements void showStatus(String msg);
   requires Performer implements ToolButton getSelectedButton();

   void paletteUserSelected(PaletteButton button) {
      ToolButton toolButton = (ToolButton) button;
      setTool(toolButton.tool(), toolButton.name());
      setSelected(toolButton);
   }

   public void paletteUserOver(
      PaletteButton paletteBt, boolean inside) {
      ToolButton selected = performer.getSelectedButton();
      if (inside) {
         performer.showStatus( paletteBt.name() );
```

```
        }
        else if (selected != null) {
            performer.showStatus( selected.name() );
        }
    }
}
```

We suggest that most clones that could be removed with Extract Class should be solved using Extract Role instead. This way classes do not need to create delegate methods and all they have to do is to play the role. The use of delegating methods will cause code clones, which can be significant if there are many methods. In the case of roles the only repeated code is the play clause. The Extract Class may also impose the creation of a class that represents only a partial concept. This is conceptually wrong. The role, according to the definitions we use, is supposed to be a partial concept so it is conceptually a better solution.

The decision to use Extract Role or Extract Class is made considering the code nature. If the code represents a standalone concept it should be put into a class, if it represents a partial concept it should be put into a role. According to Kapser and Godfrey [KG06b], when refactoring a clone concerns such as stability, code ownership and design clarity need to be considered. We believe that roles fare better in all these criteria than the use of an extracted class, in those cases where the concept is just a partial one.

In our example the code reflects only a partial concept, that of an entity that keeps track of user's actions over a tool palette, so does not present a full class, it represents a behavior that a class must have but which does not define it. In this case a role is better suited than a class, so we opted for the use of Extract Role.

We also argue that Extract Role can be used instead of Extract Superclass. In this case the use of delegation methods is not required so the decision is purely a conceptual one. In single inheritance languages we should not use inheritance as a simple code reuse mechanism. If the concern involved represents just a small set of the class behavior should the class inherit from another just to reuse this code? Inheritance defines what a class is and the class should not be defined by a small subset of its behavior, unless it is the main concern of that class. We believe that a role solution is conceptually better as it says only that a class has that behavior not that the class is defined by it. It also leaves the class free to inherit from another class and the role to be reused by another class that already has a superclass, providing a better reuse scenario.

Again, the decision to use Extract Role or Extract Class depends on the nature of the concern. If the concern is better modeled by a class then Extract Superclass should be used. If the concept is better modeled by a role then Extract Role should be used.

7.2.2 Extract Role Changing Types

You have a class with the same code as another but differ in the types they use and the similar code is not the main concern of the classes.

Create a new role and move the relevant fields and methods from the old classes to the new role, using generics for the different types, and make the classes play the role-

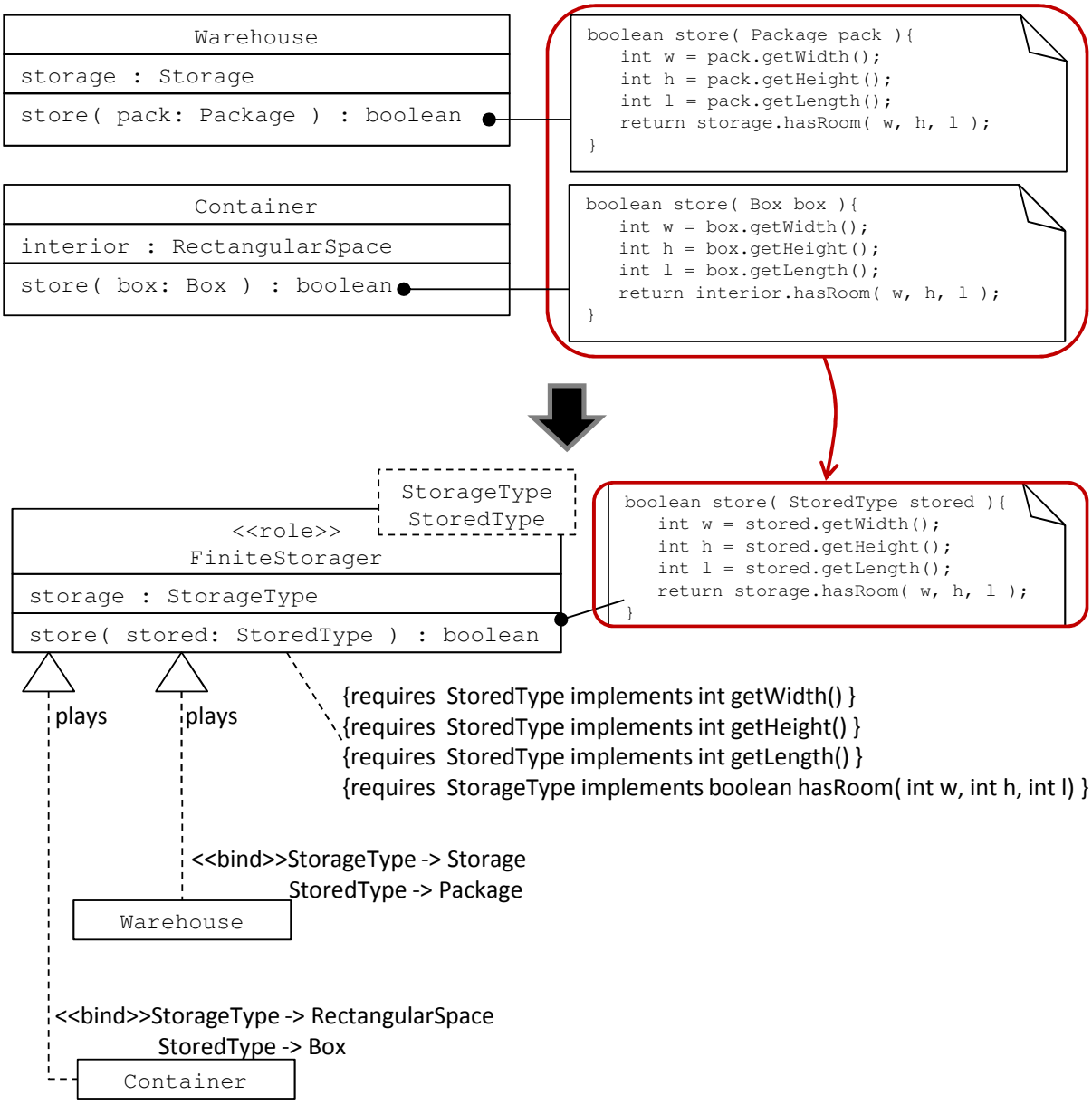

Figure 7.6: Extract Roles Changing Types

Motivation

We apply this refactoring to the clones that have identical structure but use different, unrelated, types.

Sometime the classes get responsibilities added that are generic enough to be used in several other situations. However the types they use are different in each situation. It can be that the types are related. In such cases you can use the most general type and then try to remove the clone using Extract Role. If a common type cannot be used then apply this refactoring. You have to capture that general responsibilities and try to factor out the points of possible generalization, namely types. Then check to see if the whole makes a clear concept on its own.

Mechanics

- Group the responsibilities you want to separate in a coherent set.
- Create a role with a name that indicates the concern it deals with.
- Make the class play the role.
- Identify the different types used by the code that cannot be replaced by a common type.
- For each identified type mark it to be replaced by a generic
- Use Move Field on each field you wish to move.
  - Check if the field name is adequate for all the uses, and rename it otherwise.
- If the moved field is of a marked type replace the type with the corresponding generic.
- Update the plays clause to state the concrete type for the generic.
- Use Move Method on each method you wish to move.
  - try to generalize the role, sometimes the names must change as they are tuned for the class's intent and not the role's intent, if this is the case then use Extract Role with Types and Methods.
- If the moved method uses a marked type substitute it for the corresponding generic.
- If the moved methods call methods from the class add a requires statement for each called method.
  - try to generalize the role, sometimes the names must change as they are tuned for the class's intent and not the role's intent, if this is the case then use Extract Role with Types and Methods.

- If the moved methods call methods from the generic types add a requires statement for each called method.
  - again, try to generalize the role, if the names of the called methods are generic enough then use them, otherwise try to prepare them for renaming using EXTRACT ROLE WITH TYPES AND METHODS.
- Update the plays clause to state the concrete type for the generic after each new generic added.
- Compile and test.
  - If the moved method does not call other methods to be moved compile and test after each move.

Example

The classes of figure 7.2 use different types to represent a request. We can make a role specify the type of the request and thus remove the clone. The first steps are identical to the EXTRACT ROLE. We create a role, in this case a LetRequest role and make the classes play the role.

```
public role LetRequest {
}

public class PortletWebRequest /* ... */ {
    plays LetRequest letRequest;
}

public class ServletWebRequest /* ... */ {
    plays LetRequest letRequest;
}
```

When moving the `getLocale` method it calls the `getRequest` method, which goes into the requirements list, in the class which returns a request object of the concrete type used by each class. For example if we wanted the role to address the PortletWebRequest we would write the role as

```
public role LetRequest {
  requires Performer implements PortletRequest getRequest();

  public Locale getLocale() {
    return performer.getRequest().getLocale();
  }
}
```

This solution works for the PortletWebRequest but not for the ServletWebRequest. We need to replace the PortletRequest type with a generic. Since we are calling the `getLocale` method on this type we must place it in the requirements list. The role would look like this

```
public role LetRequest<RequestType> {
  requires Performer implements RequestType getRequest();
  requires RequestType implements Locale getLocale();

  public Locale getLocale() {
    return performer.getRequest().getLocale();
  }
}
```

The classes would have their plays updated to

```
public class PortletWebRequest /* ... */ {
    plays LetRequest<PortletRequest> letRequest;
}

public class ServletWebRequest /* ... */ {
    plays LetRequest<HttpServletRequest> letRequest;
}
```

After moving all the methods the role would be

```
public role LetRequest<RequestType> {
   requires Performer implements RequestType getRequest();
   requires RequestType implements Locale getLocale();
   requires RequestType implements String getContextPath();
   requires RequestType implements String getRemoteUser();
   public Locale getLocale() {
      return performer.getRequest().getLocale();
   }
   public String getContextPath() {
      return performer.getRequest().getContextPath();
   }
   public String getRemoteUser() {
      return performer.getRequest().getRemoteUser();
   }
}
```

We used the RequestType type to represent the type of the request inside the role. We can also see that the code calls a number of methods on the request object. Every method is put into the requirements list. .

7.2.3 Extract Role with Configurable Methods

You have classes with the same code but they differ on the names of some methods and the similar code is not the main concern of the classes.

Create a new role and move the relevant fields and methods from the old classes to the new role, using the renaming mechanism to allow for different method's names, and make the classes play and configure the role

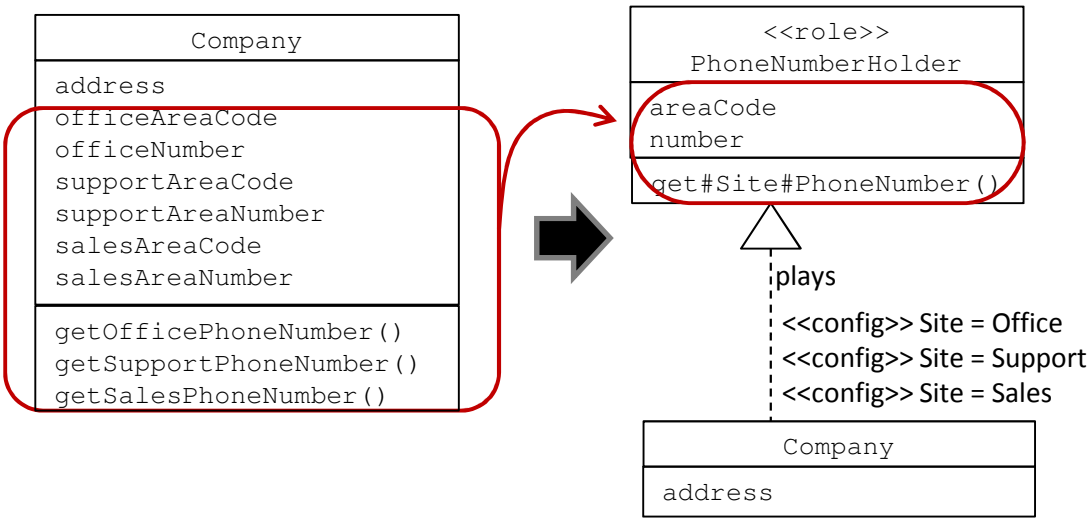

Figure 7.7: Extract Role with Configurable Methods

Motivation

This refactoring is to be used with clones with the same structure and types but using different methods. To remove these clones we need to use the renaming mechanism of JavaStage.

A class may have responsibilities that are generic enough to be used elsewhere. The problem with this reuse is that the name of the methods are different between its several uses. If general purpose names can be used then consider using Extract Role. If general purpose names cannot be used because the names are instance specific and must reflect the context in which they are used then this refactoring can help. Since the name of the methods is what contextualizes the code we can make it generic by allowing those names to be configured.

Mechanics

- Group the responsibilities you want to separate in a coherent set.
- Create a role with a name that indicates the concern it deals with.

- Make the class play the role.
- Use MOVE FIELD on each field you wish to move.
  - Check if the field name is adequate for all the uses, and rename it otherwise.
- Identify the methods whose names should be configurable
- Use MOVE METHOD on each method you wish to move.
- If the moved method is to be made configurable or calls methods that can be configurable use the renaming mechanism to mark them as configurable, taking advantage of naming patterns to reduce the amount of configurations to use.
- If the moved methods call methods of the class add a requires statement for each called method.
- If the role uses a configurable method of another type add it to the requires list as well.
- Update the plays clause on each class configuring the methods.
- Compile and test.

Example

Consider the classes in figure 7.3, they have to either expose or ignore methods as MBean operations. Their code is pretty much the same, except for methods names.

After creating the role and making the classes play the role we move the fields to the role, renaming them so they are contextualized to the role code and not to a class code.

```
public role MethodBeanInfoAssembler {
   private Set<String> methods;
   private Map<String, Set<String>> methodMappings;
}
```

We then identify the methods to be configured. These are the class methods `setXXXMethods`, `setYYYMappings`. The configurable methods also include the called methods `isZZZ`. We then move each method at a time.

Beginning with the `setXXXMethod` the role would be declared as:

```
public role MethodBeanInfoAssembler {

   private Set<String> methods;
   private Map<String, Set<String>> methodMappings;
```

```
   void set#Action#Methods(String[] methodNames) {
      methods = new HashSet/*...*/;
   }
}
```

and the classes would have a plays clause like

```
class MethodNameBasedMBeanInfoAssembler /*...*/ {
   plays MethodBeanInfoAssembler(  Action = Managed ) infoAssembler;
}

class MethodExclusionMBeanInfoAssembler /*...*/ {
   plays MethodBeanInfoAssembler(  Action = Ignore ) infoAssembler;
}
```

Finally we move all the methods and update the plays clause to do the configuration of the role in each class. The final code would be

```
public role MethodBeanInfoAssembler {
   requires Performer implements
                  boolean is#Include#(Method method, String beanKey);

   private Set<String> methods;
   private Map<String, Set<String>> methodMappings;

   void set#Action#Methods(String[] methodNames) {
      methods = new HashSet/*...*/;
   }

   void set#ActionMethod#Mappings( Properties mappings){
      methodMappings = new HashMap</*...*/>();
      /* ... */
   }

   boolean includeReadAttribute( Method method, String beanKey) {
      return performer.is#Include#(method, beanKey);
   }

   boolean includeWriteAttribute( Method method, String beanKey) {
      return performer.is#Include#(method, beanKey);
   }
}

class MethodNameBasedMBeanInfoAssembler /*...*/ {
   plays MethodBeanInfoAssembler(
            Action = Managed,
            ActionMethod = Method,
```

```
            Include = Match
          ) infoAssembler;
}

class MethodExclusionMBeanInfoAssembler /*...*/ {
   plays MethodBeanInfoAssembler(
            Action = Ignore,
            ActionMethod = IgnoredMethod,
            Include = NotIgnored
          ) infoAssembler;
}
```

Using this refactoring we could remove the clone identified in figure 7.3. Each class only retained the method that determines if the method is to be managed or ignored, all the code, even the fields, moved into the role.

7.2.4 Extract Role with Types and Methods

You have classes with similar code but that code differs on the types and methods names and the similar code is not the class main concern.

Create a new role and move the relevant fields and methods from the old classes to the new role, using the renaming mechanism to allow for different method's names, using generics for the different types, and make the classes play and configure the role

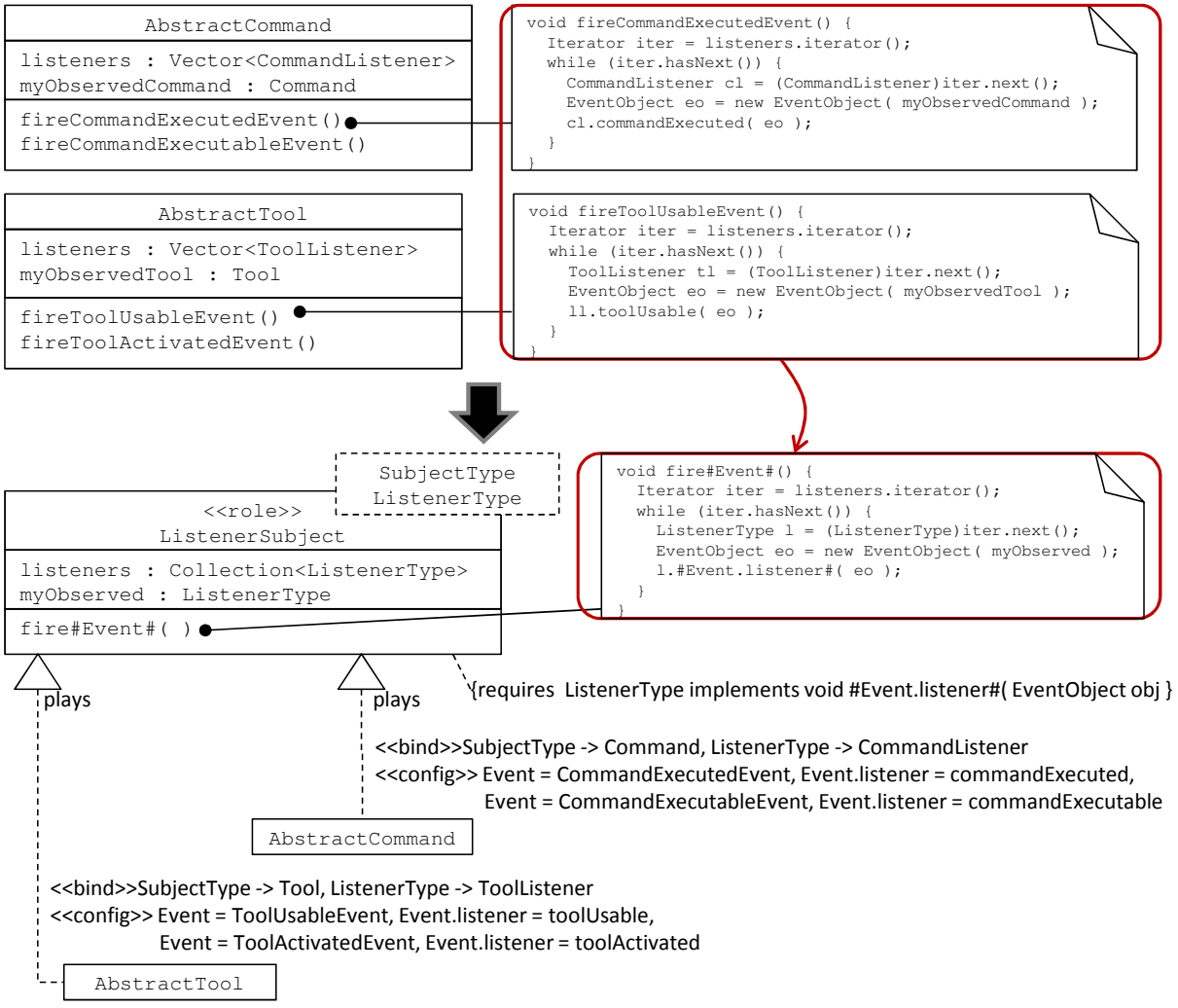

Figure 7.8: Extract Role with Types and Methods

Motivation

We apply this refactoring to the clones that have identical structure but use different, unrelated, types and methods.

Sometime classes have responsibilities that are generic enough to be used in several other situations except that they use different types in each situation and the names

of the methods are explicit to that particular case. You have to capture that general responsibilities and try to factor out the points of possible generalization, namely types and methods names. Then check to see if the whole makes a clear concept on its own.

Mechanics

- Group the responsibilities you want to separate in a coherent set.
- Create a role with a name that indicates the concern it deals with.
- Make the class play the role.
- Identify the different types used by the code that cannot be replaced by a common type.
- For each identified type mark it to be replaced by a generic
- Use Move Field on each field you wish to move.
  - Check if the field name is adequate for all the uses, and rename it otherwise.
- If the moved field is of a marked type replace the type with the corresponding generic.
- Update the plays clause to state the concrete type for the generic.
- Identify the methods whose names should be configurable
- Use Move Method on each method you wish to move.
- If the moved method is to be made configurable or calls methods that can be configurable use the renaming mechanism to mark them as configurable, taking advantage of naming patterns to reduce the amount of configurations to use.
- If the moved methods call methods of the class add a requires statement for each called method.
- If the role uses a configurable method of another type add it to the requires list as well.
- Update the plays clause on each class configuring the methods.
- Compile and test.

Example

The AbstractCommand class from figure 7.4, informs its listeners when the command finishes its execution and when it becomes executable, among other actions: The AbstractTool informs its listeners whenever the tool becomes usable or has been activated. This is the same behavior but using different types and method names. Furthermore we can see that this behavior can be generalized for other uses as well.

So we begin by creating a role that represents an Observer, or a Listener. The types of the listeners, CommandListener and ToolListener, are different so we create a generic to represent this type and create a storage for them in the role. An EventObject, in each fire method, is used to represent the source of the event. This is defined as being an Object so we do not need a Generic for this Type. However each class stores a reference to the observed tool/command, so we need to move that reference to the role and thus created a type for the observed: the SubjectType.

```
role ListenerSubject<SubjectType, ListenerType> {
   Collection<ListenerType> myRegisteredListeners;
   SubjectType myObserved;
}
```

Now we need to move the methods. To maximize its reuse some methods need to be configured. Those methods are the fireXXX methods and the methods called on the listeners. Thus we can use a fire#Event#( ) method for the role and, for the listeners a #Event.listener#(EventObject o) method.

The methods are essentially the same so we can use the multiple version feature of JavaStage to develop just one method.

```
role ListenerSubject<SubjectType, ListenerType> {
   Collection<ListenerType> listeners;
   SubjectType myObserved;

   public void fire#Event#() {
      Iterator iter = listeners.iterator();
      while (iter.hasNext()) {
         ((ListenerType)iter.next()).#Event.listener#(
                                 new EventObject( myObserved ));
      }
   }
}
```

Since the role calls methods on the listeners we need to place those methods in a requirement list.

```
role ListenerSubject<SubjectType, ListenerType> {
  requires ListenerType implements void #Event.listener#( EventObject obj);
```

```
  // ...
}
```

Now all we need to do is update the plays clause to reflect the usage of this role and its configuration.

```
abstract class AbstractCommand implements Command {
   plays ListenerSubject<Command,CommandListener>(
     Event = CommandExecutedEvent, Event.listener = commandExecuted,
     Event = CommandExecutableEvent, Event.listener = commandExecutable
   ) commandListener;
}

abstract class AbstractTool implements Tool {
   plays ListenerSubject<Tool,ToolListener>(
     Event = ToolUsableEvent, Event.listener = toolUsable,
     Event = ToolActivatedEvent, Event.listener = toolActivated
   ) toolListener;
}
```

For this refactoring we use all JavaStage features and a combination of the previous refactorings. With this we can achieve a greater level of reusability. For example we could derive from this role a full purpose role that could be used for many more kinds of events.

There are also features used in the role that, for simplicity, are not explicitly shown in the code. One such feature is the role constructor where we can define the type of storage used for the listeners.

7.3 Summary

This chapter presented a series of clones that could not be refactored and why they could not. It also presented four refactoring involving roles that can remove the mentioned clones. The four refactorings were duly explained along with examples of how they can be applied to the clones found in the target systems of the case studies.

Part III

Validation

Chapter 8

Towards a Role Library

In order to prove the validity of our approach we developed a compiler for the JavaStage specifications, based on the OpenJDK compiler. The ultimate goal is to develop a library of roles with this tool. This will prove that roles can enhance modularity as a role may be reused in contexts where classes cannot.

To build such a library we started by analyzing the 23 GoF patterns [GHJV95]. Design Patterns are a good starting point because they provide solutions to recurring problems in software development. So it may be possible to build roles that capture the basic structure of the pattern so we can reuse that basic structure instead of replicating it whenever a pattern is applied. Another advantage of using patterns as a starting point is that they are used in many frameworks and so represent a lot of real world code. If we can create roles for these patterns, it means our approach is likely to have an impact on many of today frameworks and applications.

8.1 Roles in Design Patterns

Each pattern defines a number of participants that collaborate with each other to carry out their responsibilities. Some participants in these patterns can be seen as roles while others cannot. This distinction is made in [HK02] by considering the roles superimposed or defining, even though their concept of roles differs somewhat from ours.

A defining role is a role that completely defines the class, that is, the class has no other concern besides its participation in the pattern. Such an example is the State hierarchy in

the State pattern. In this pattern there is an object - the Context - that alters its behavior when its internal state changes. The pattern proposes the State, which is a class (or an interface) that defines the behavior associated with a particular Context. Each possible state is then implemented by a subclass (or a class implementing the interface). The Context object has a state object into which it delegates state-specific requests. When the Context object changes state it actually changes the state object. Each state subclass has no other concern than performing the actions the object forwards to it when in that state.

A superimposed role is a role that is assigned to classes that have other concerns outside their participation in the pattern. In the Chain of Responsibility pattern, for example, the Handler role is superimposed in every participant that is a link in the chain. It has to either handle a request or forward it to its successor. This behavior is not the main concern of the class but is has to perform these actions just because it is part of a Chain of Responsibility.

For each pattern we took the roles played by each participant and focused on similar code between instances of the pattern to find reusable code [BA11]. A goal we expected to reach with our approach is a better modularization in some of the patterns.

For those patterns for which we developed a role we also implemented a sample scenario that illustrated its use, but those samples will not be discussed here, except the sample for the Observer pattern that will be used as an indicator of role-player independence. The results presented here show that it is possible to build reusable roles that are independent of their players.

We will explore, for each pattern, similarities between its instances in order to discover replicated code and if that code can be placed inside a role. We will also explore ways to use roles in each pattern.

Abstract Factory

The Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern relies mainly on inheritance where every factory inherits from an abstract class or implements an interface.

Even though the code for each factory is basically the same, creating and returning a product, the products they create are very different. Because the return types and how to create the products differ so greatly it is rather difficult to obtain a generic role for this pattern. Nevertheless roles may be useful for those cases in which multiple inheritance may be used.

Builder

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.

Each building process is case specific and therefore no generic role could be developed.

Factory Method

The Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets classes defer instantiation to subclasses.

The normal implementation of this pattern is clearly instance dependent and provides no common code between instances. There is, however, a variation whose purpose is to connect parallel class hierarchies. In this variation each class belonging to a hierarchy delegates some responsibilities to a corresponding class belonging to another hierarchy. For this variation of the Factory Method pattern there is one similarity between instances that we can explore: each class has a method that creates the corresponding object. What we did was to put the creation of the product in a creator class. The creator class provides methods that create the required product, one method for each product. Then the classes just have to call the corresponding method in the creator. One advantage of this solution is the modularization of the pattern in which the correspondence between classes is made in a separate class rather than on a class by class basis. Future additions and changes are made in this class only. Because the creation process is now delegated to a single class this means that, as an extra advantage, we can change the creator dynamically.

We developed a role that allows the specification of the factory method that creates the object of the corresponding class. The method uses the class directive in the renaming feature which allows the plays reference to be made only in the top class of the hierarchy. On the other hand the use of the class directive implies that the creator must rely on method naming conventions. This is, however, a small price to pay for the extra modularity gained.

In a sample implementation we simulated a Figure hierarchy in which each figure has a specific manipulator. The Figure class plays the FactoryMethod role where it defines that the product created is of the type FigureManipulator and the creator is a ManipulatorCreator object. Only the Figure class has the plays declaration, as stated above, but each subclass has it's own createManipulator method that redirects the call to the ManipulatorCreator. This is the class responsible for the creation of the correct Manipulator for each subclass. As said above, if we wish to change the way manipulators are created or even which manipulator is created for each subclass we do it in this class alone and we need not to change the Figure subclasses. Figure 8.1 illustrates this example.

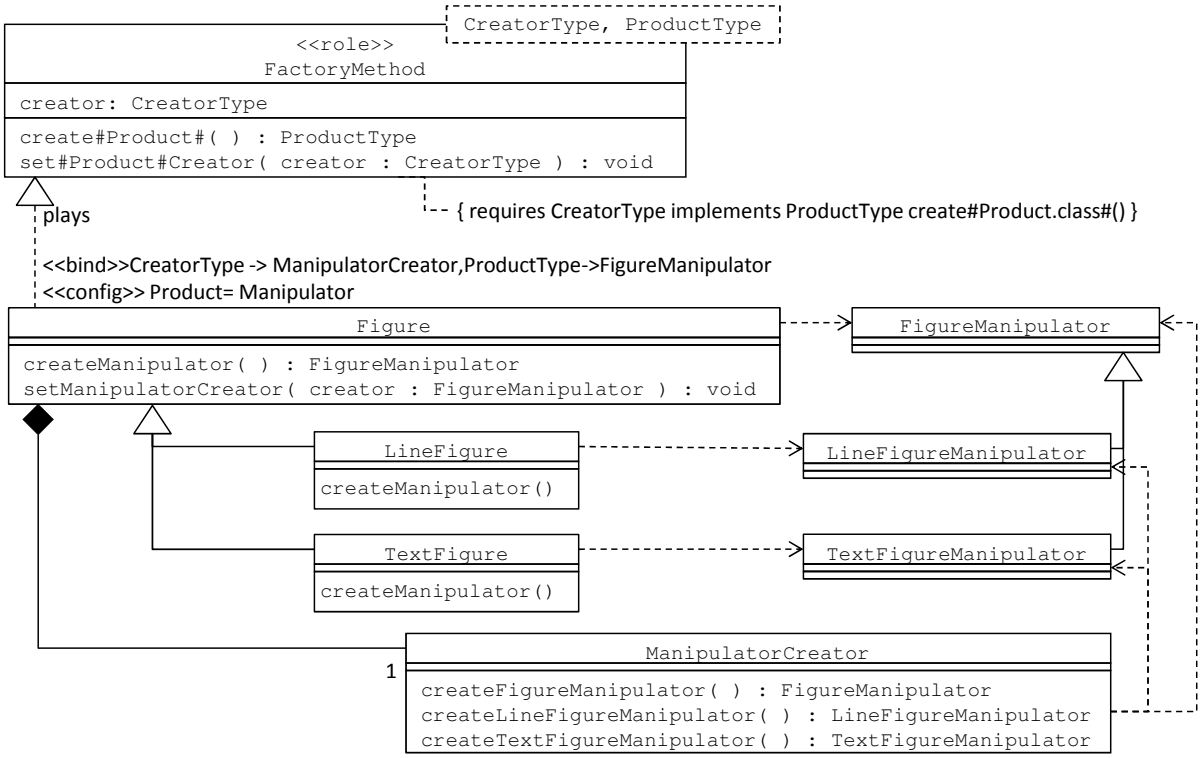

Figure 8.1: The use of the FactoryMethod role to relate a Figure subclass to the corresponding FigureManipulator.

Prototype

The Prototype pattern specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype. This pattern relies on the prototype class to have a clone method that produces an identical copy of the object.

While every class has its own mechanisms for cloning its objects it may not be sufficient because of the deep copy vs shallow copy problem. The clone method may do just a shallow copy where a deep copy is needed, or vice-versa. If the client, when it really matters, could choose how the copy is made it would be more practical. For this we developed a role that moves the creation of the copy to another class. This mechanism is similar to the one used in our solution to the FactoryMethod pattern role. This means that the new class is responsible for creating the copies of all classes that may be used as prototypes and thus may choose how to make the copy.

For demonstration purposes we took the Figure hierarchy again and made Figure play the Prototype role. We also created a FigureCloner class which is the responsible for the clone creation. With this solution we can change the way the clones are created (deep or shallow copy) without changing the hierarchy classes. This allows for a more modular approach. We can also change the way clones are created, dynamically, by changing the cloner.

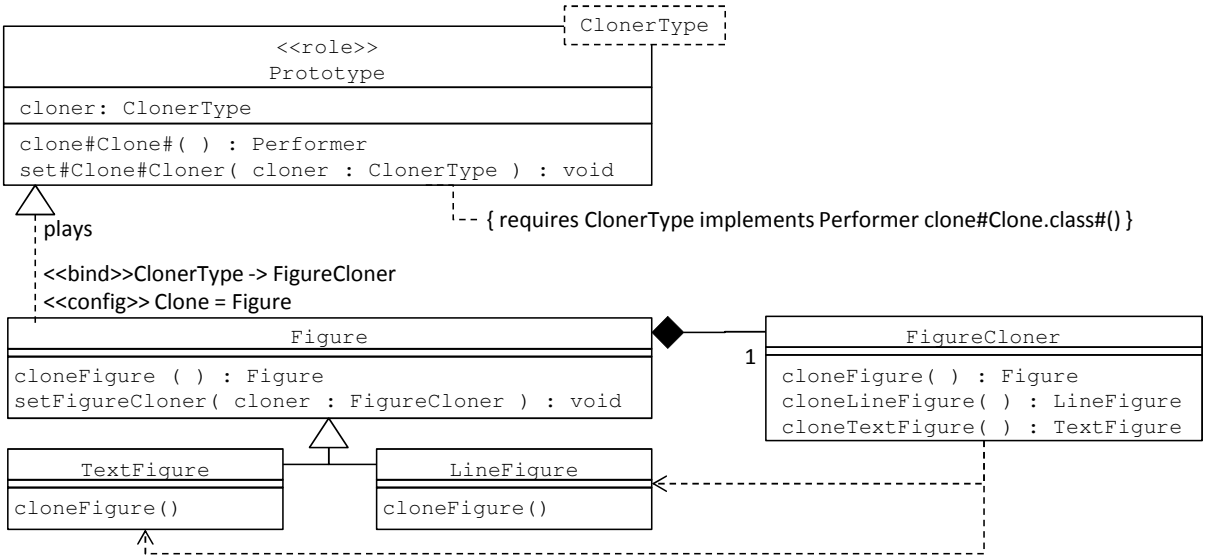

Figure 8.2: The use of the Prototype role to create clones for a figure hierarchy.

Singleton

The Singleton pattern's intent is to ensure that a class has only one instance, and provide a global point of access to it. It defines only one role: the singleton role. The purpose of the role is to ensure that only one instance is created, maintain that single instance and providing access to it.

The suggested implementation of this pattern involves the use of a static reference and the static method that provides access to the instance. It also needs the class constructor to be declared private. The variations between different instances of this pattern are:

- type of the singleton
- name of the access method

In our singleton role the name of the access method is configured using the renaming feature. The type of the singleton is the class that plays the role (Performer) and the role creates a new instance via `new Performer()`. The programmer still has to make the constructor private, though.

Adapter

The Adapter pattern converts the interface of a class into another that the client expects. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

The code in this pattern depends on the nature of the adaptee and no specific role could be developed. The single similarity that we found was that some methods are forwarded

directly to the adaptee. But each method may have multiple parameters and return types that it is impossible to create an all purpose forward method. This is a pattern in which the multiple inheritance may be useful, so instance specific roles can be developed as a way to emulate multiple inheritance.

Bridge

The Bridge pattern decouples an abstraction from its implementation so the two can vary independently. To do this the pattern defines an interface for the abstraction and another interface for the implementation. The way they are connected is unique to each pattern instance. The sole similarity between several implementations is the field that the interface class has to reference the concrete implementer. We thought that this was not enough to provide a role for this pattern.

Composite

The Composite pattern composes objects into tree structures to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

Each composite must maintain a collection of child components and implement the operations defined by the component hierarchy. There are common operations for all composite instances that are related to maintaining the collection of children (like addChild, removeChild, ...). The operations defined by the component hierarchy are instance dependent and are not suitable for generalization, even thought most of their implementations is the traversal of the children collection and performing the corresponding operation on each child. A map function, popular within functional programming, would be useful here.

For this pattern we did not develop a particular role, but we reused the Container role, which takes care of children management in the composite role. This role allows the definition of the management methods via the renaming mechanism

Decorator

The Decorator pattern attaches additional responsibilities to an object dynamically. To achieve this the decorated component is enclosed in another object that adds the required functionality. This pattern is based on an inheritance hierarchy in which the component and the decorators share the same interface. This allows for the use of roles as a multiple inheritance both for roles and components.

If there are many components then we can define a role that contains the default behavior of the components and another role that defines the default behavior for the decorators.

This way they only share the same interface while having a default implementation for each.

Facade

The Facade pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Because this pattern implementation depends exclusively on the system it provides an interface for there is no similar code between instances of the pattern. There was no role that could be developed for this pattern.

Flyweight

Flyweight uses sharing to support a large number of fine-grained objects efficiently. It depends on small sharable objects that the client manages and on a factory of flyweights that creates, manages and ensures the sharing of the flyweights.

The concrete flyweights and their interface are instance dependents and are not open to reusability between instances. The same is not true for the flyweight factory as most share the same behavior: verify if the required flyweight already exists and, if so, return it or, otherwise, create it, store, and then return it.

We developed a role for the flyweight factory that relies on a map to manage the flyweights and also supplies the management method. The flyweight creation method is the only instance specific method the factory needs so we require the player to supply such a method. The types of the flyweights are defined using generics and the methods names are configured using the rename strategy.

In a sample implementation (see figure 8.3) we created a Glyph factory where the glyph's key is a character. There are two types of glyphs: alphanumeric, for letters and numbers, and symbol glyphs, for other characters. The factory takes the character and returns the corresponding glyph.

Proxy

Proxy provides a surrogate or placeholder for another object to control access to it. The real subject is placed inside one object, the proxy, which controls access to it. Some, simple, operations are dealt by the proxy itself, while others, more complicated, are forwarded to the subject and handled by it.

Which methods are forwarded or handled by the proxy are instance dependent as is the creation of the subject, but the forward mechanism and checking if the subject is already created or accessible is similar between instances. We therefore developed a proxy role

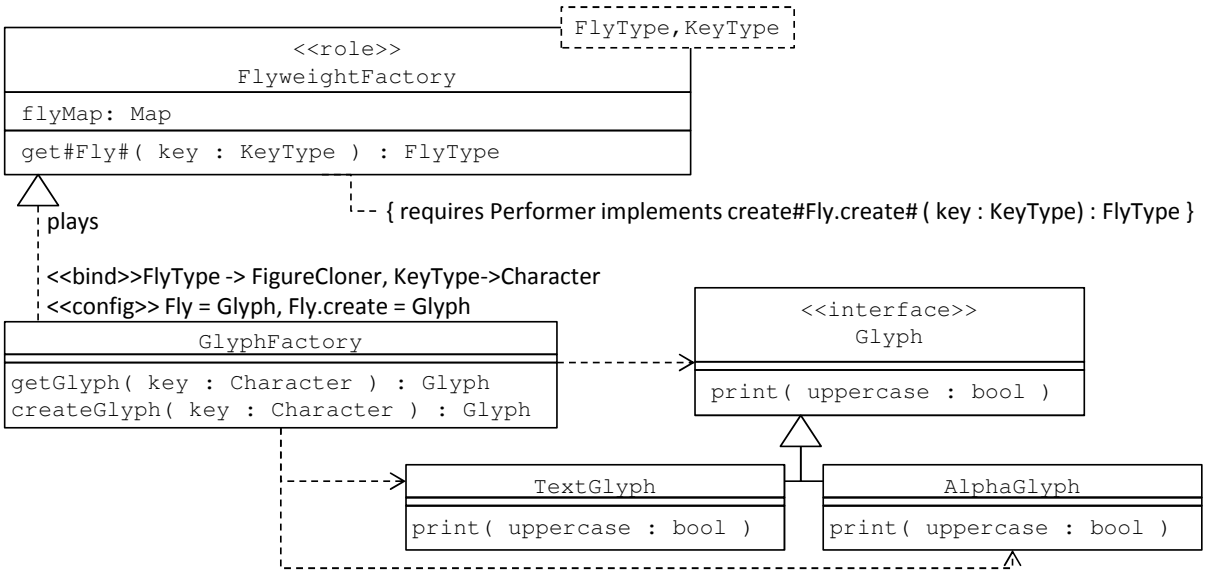

Figure 8.3: Flyweigth Factory example

that provides this similar behavior. It stores the reference to the subject and provides the method to check the existence of the subject and triggers its creation otherwise.

Chain of Responsibility

The purpose of the Chain of Responsibility pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. The receiving object passes the request along the chain until an object handles it.

The implementation of this pattern, in Java, often involves the use of a reference to the successor and the code to handle or pass the request. The specific code for each instance relates to how the request is handled and how each handler determines if it can or cannot handle the request. Apart from that, the variations between different instances of this pattern are:

- type of the handler
- type of the request
- name of the request method
- name of the method that verifies if the request can be handled
- name of the method that does the handling

This pattern has some variations that we observed. There are some implementations that require no request information to be passed, that is, the request method has no

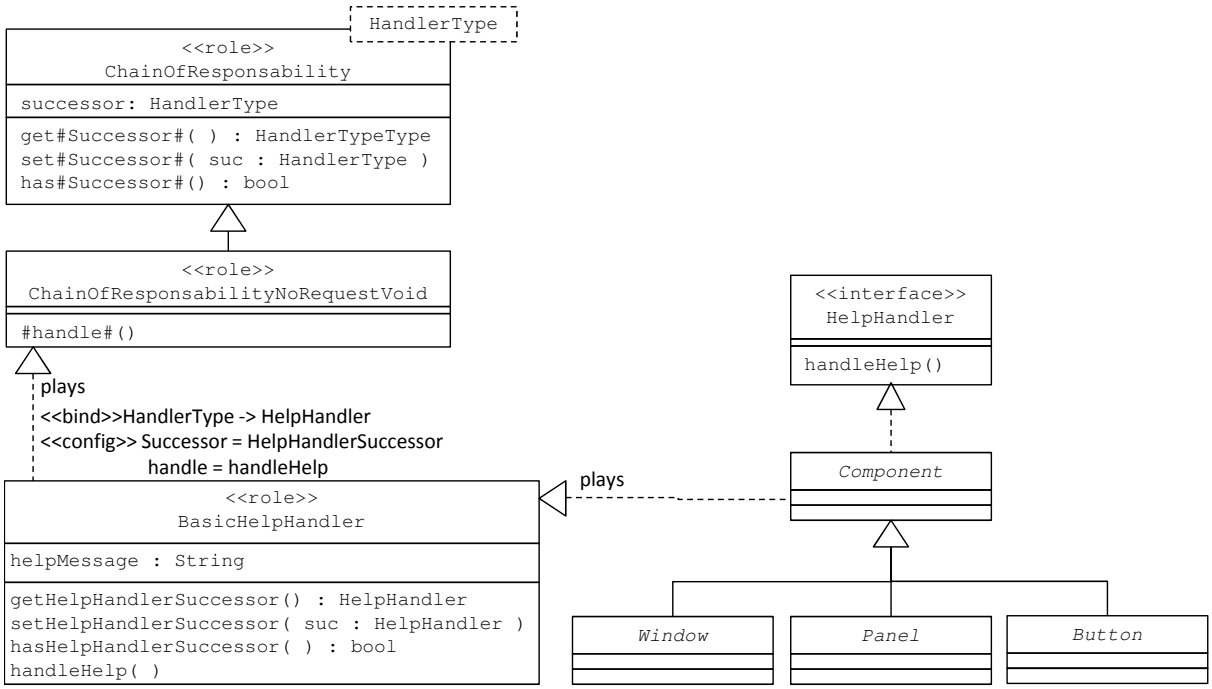

Figure 8.4: Using a role to play another, more generic role, in an example Chain of Responsibility implementation.

parameter. There are also implementations in which the request method returns a value. To accommodate these variations we developed a role for each.

In our roles the names of the methods are configured using the renaming feature. The types of the handlers, the request (in those roles that use it) and return type (where in use) are defined by generics. The roles also define the get and set methods for the successor, which are also configurable via renaming.

In our example we needed a help handler for graphical components. These components form an inheritance hierarchy that is similar to a real component hierarchy. Since the main concern for the components is not the handling of help messages and because they are already part of an inheritance path we developed a role for the help handling part. The role we developed for this example, BasicHelpHandler, plays the ChainOfResponsabilityNoRequestVoid role that is part of our role library. This particular role implements a chain of responsibility in which the request methods return no value and has no request parameter and extends the ChainOfResponsability role. The BasicHelpHandler implements basic help methods such as the use of a help message and defines the request handling methods' names. The component superclass then plays the BasicHelpHandler role, enabling all components to handle help requests. Because of the multiple inheritance emulation capabilities of roles even non components can play this role, or components that are not directly subclasses of component may play this role.

It is worth to mention that the BasicHelpHandler role could be reused in several other

applications because it is in no way associated with the component hierarchy.

Command

The purpose of the Command pattern is to encapsulate a request as an object, thereby letting developers parameterize clients with different requests, queue or log requests, and support undoable operations. Commands are usually organized in a inheritance hierarchy. Due to the specific nature of each command hierarchy we could not provide a role for this pattern.

Interpreter

The intent of the Interpreter pattern is to, given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. This is done by creating a class for each grammar rule of the language. As each language has its own grammar therefore each class is unique as is the way to interpret the grammar. Hence the difficulty to find similarities between instances of this pattern that lead to no roles being developed.

Iterator

The purpose of the Iterator pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This is done by creating a common interface for the iterators while each aggregate is responsible for the creation of a specialized iterator. Since all that is common between iterators is their interface there is no role developed for this pattern.

Mediator

The Mediator pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. The pattern suggests the use of a mediator class, also called the director, that the several colleague classes communicate with when they have changed state. The mediator class is the responsible for the interaction between the colleagues. Since each instance requires a different director and the needed interactions are also instance dependent we found no common ground to build a role. The communication with the director, however, is usually done with the observer pattern for which we developed a generic role, so our role library would be used in this pattern as well.

Memento

Memento captures and externalizes an object's internal state so that the object can be restored to this state later, without violating encapsulation. The internal state, or enough information to compute that state, of the originator object is stored in a separate class, the memento, that is passed to a caretaker. If later on the originator state must be reset the caretaker passes the memento object back to the originator. The caretaker does not operate on the memento, only the originator does. Because the internal state of an object is known only to the object it is impossible to develop a generic role to represent a memento. The way the memento is stored and passed to the originator is also highly dependent on the actual use of the pattern so no appropriate role was developed.

Observer

The observer pattern's intent is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. For that purpose it defines two roles: the subject and the observer. The subject must notify all the observers upon a change in its state therefore it must know its observers. The observer must implement the updating interface and respond to the notifications.

This pattern has several implementation alternatives, in particular for the updating mechanism. Besides the update method name there is the question of its parameters. It may have none to several. Some usual parameters are the subject of the update, some info to indicate which state item has been changed, and the new value for that item. In the Java's AWT approach, a widely used and well known example, the update mechanism makes use of an Event which aggregates several information about the updating, including the previously referred information. It is therefore possible to pass several items of information in one single parameter. Furthermore there is not a single update method but several methods, each implying a specific change (such as: mouseMoved, mousePressed, ...). For our approach we will follow Java's AWT observer style not only because it is well known but also because it is adaptable to many different scenarios.

In this pattern there are things that are instance specific such as the actions taken by the observer when an update method is called or deciding whether or not to trigger the update mechanism. Using the event approach the nature of the event is also instance specific. The several update methods names are also specific to an instance. But there are several questions that are common to all instances such as: there is a collection of observers per each subject, the methods to manage those observers and the notifying methods.

Our subject role makes it possible for the programmer to specify which Container to use via the constructor. With the name renaming feature it also allows to specify

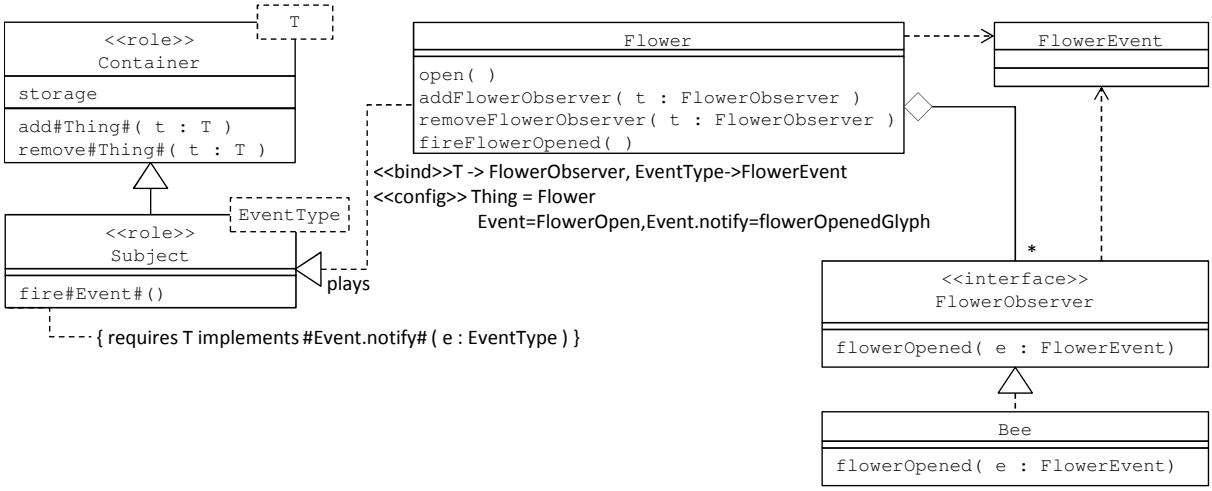

Figure 8.5: Sample implementation of a subject role.

the names of the update methods as well as the event firing methods. The observers management methods are also specified by using the renaming feature. It must be pointed that the management of the observers is done using our Container role. The observers and event types are defined using generics. We thus built a role that fulfills all the needs for a subject role. Its use it illustrated in figure 8.5.

We did not implement an observer role because all observers are implementation specific. For the same reason we did not implement an event role. Another aspect that we did not include is the firing of the events, once again because that is implementation specific. Nevertheless our role implements all the firing methods, the programmer needs only to call them at the correct site.

State

The State pattern's allows an object to alter its behavior when its internal state changes.

There are almost no variations in this pattern because each instance is specially made for the task at hand. The only possible similarity between several instances is the state change mechanism, which can be made in one method. We therefore developed a role that is responsible for the state transitions and keeping the current state. The state change method terminates the actual state before updating to, and starting, the new state. The name of the state change method and the state terminating and starting methods are configured by the rename feature, while the state class is configured by generics.

Strategy

The purpose of the Strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from

clients that use it. The pattern proposes an inheritance hierarchy of algorithms with each subclass implementing its version of an algorithm. In most strategy patterns each strategy class has only one method: the one that implements the intended strategy. There is no similar code between instances of the pattern, apart the fact that some instances store the currently selected strategy, so no generic role could be produced. The mechanisms for selecting and using the strategy are also instance specific so no role for the context was produced.

Template Method

Template Method defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Methods lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Template methods are mostly defined in superclasses with the general algorithm based on method calling. The subclasses specify the behavior of the several methods required by the algorithm. This is clearly a case in which inheritance is the optimal solution. As the template method structure is dictated by the concrete application there is no role available for this pattern.

Visitor

The Visitor pattern's intent is to represent an operation to be performed on the elements of an object structure. Visitor allows the definitions of a new operation without changing the classes of the elements on which it operates. This pattern defines two roles: the element and the visitor. The element must provide an accept method that takes a visitor as an argument. The visitor must declare a visit method for each element type. The variations between different instances of this pattern are:

- type of the element classes;
- type of the visitor;
- name of the accept method;
- name of the visit methods;
- what to do in each visit method.

For this pattern we implemented an element role that permits the specification of the accept method name and where the type of the visitor is configured using generics. The visit methods names are also configurable via the renaming mechanism where we used the class construct. The element classes are typically part of an inheritance tree, meaning that they all (or most of them) have a common superclass. Thus the declaration of the element

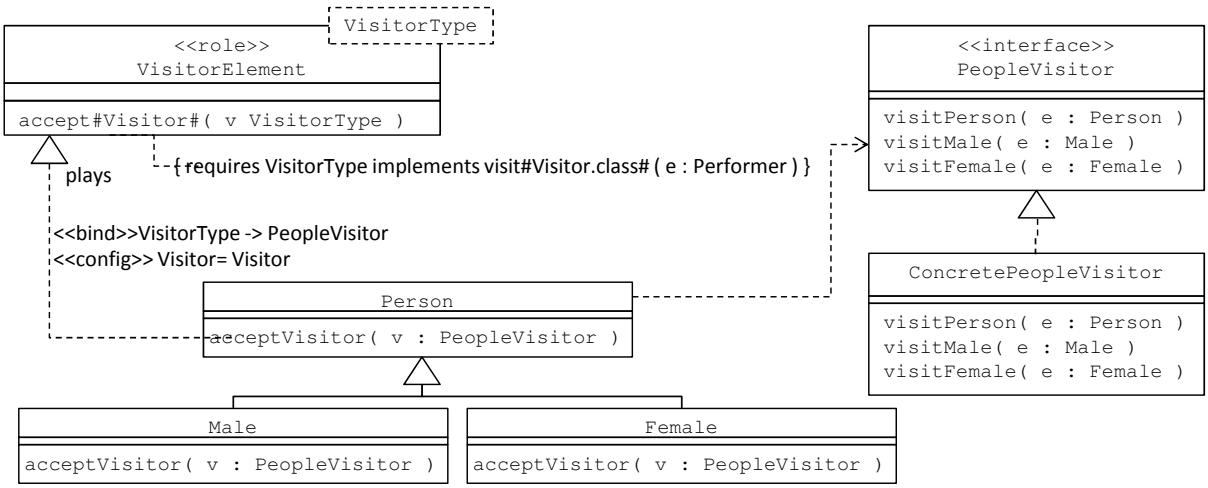

Figure 8.6: An implementation of the visitor pattern with roles.

role may be done in the superclasse only. If the element classes are not related then each must play the element role. In this case it would be better to create a specific role. This role would play the generic role fully configured for this particular instance, much like what we did in our sample implementation of the chain of responsibility pattern. We did not create a visitor role because they are instance specific and have no common code.

In our implementation of a visitor pattern instance, shown in figure 8.6, we created a simple hierarchy of Person with two subclasses: Male and Female. The Person superclass plays the VisitorElement role. Since this role uses the .class configuration every subclass will have the acceptVisitor method configured to call the correct visit method of the PeopleVisitor without having to implement it.

Discussion

Table 8.1 resumes the previous discussion. The table presents, for each pattern, the similarities found between instances and the name of the role developed, if any. For each role we stated the reusability of that role, its additional advantages and limitations. Every role that we developed has the advantage of reducing the code programmers need to write to implement a pattern and so we did not include it in the table, to save space, and named that column "additional advantages".

In table 8.2 we reproduce the information from our roles stating which methods are required and who must supply them and which methods they supply to the intrinsic. From the table we can see that every supplied method uses the renaming mechanism, showing that it would be difficult to achieve role reuse without it. The table also shows that the required methods are not all related to the performer but also to the participants in the collaboration. In fact, some roles do not depend on the Performer at all. The playedBy

			Developed role properties	
Pattern	Similarities between instances	Role name	Additional advantages	Limitations
Abstract factory	Same basic structure but differs greatly in return types and product creation methods			
Builder	None			
Factory Method	None in the general case. When connecting parallel hierarchies each subclass creates a specific product, and the creation is done by a single method	Creator	More modular. Allows a dynamic creator. Subclasses without pattern code	For connecting parallel class hierarchies only
Prototype	Every class must have a clone method	Prototype	More modular. Allows dynamic cloner. Subclasses without pattern code	
Singleton	Static reference. Static method. Private constructor	Singleton		
Adapter	Some methods just forward the request to the adaptee			
Bridge	Reference to the implementer			
Composite	Child management Some methods just traverse the children	reuses Container	child management already implemented	
Decorator	Decorator forwards requests to the decorated			
Facade	None			
Flyweight	Flyweight management	FlyweightFactory		Flyweight management only
Proxy	Proxy forwards requests to the real object Proxy creates the real object	Proxy		Proxy management only
Chain of Responsibility	Each handler forwards the request to the successor	Handler		
Command	None			
Interpreter	None			
Iterator	None			
Mediator	Usually uses the observer pattern			
Memento	None			
Observer	Observer bookkeeping Fire events call an update method in the observer	Subject		
State	Current state field When changing state the old and new states usually perform maintenance tasks	State		State management only
Strategy	Some implementations store the actual strategy			
Template Method	None			
Visitor	Each class calls a corresponding method on the visitor	VisitorElement	More modular Subclasses without pattern code	

Table 8.1: Summary of the roles developed for the GoF patterns

Pattern	Supplied/ Required	Relevant Methods
Singleton	Performer	Default constructor
	Supplied	`Performer get#instance#( )`
Observer	ObserverType	`void #Event.notify#( EventType event )`
	Supplied	`void fire#Event#( EventType e )`
Visitor	VisitorType	`void visit#visitor.class#( Performer t)`
	Supplied	`void accept#visitor#( VisitorType v )`
State	StateType	`void #state.end#( )`
		`void #state.start#( )`
	Supplied	`StateType change#state#( StateType s )`
Proxy	Performer	`ProxyType create#proxy.create#( )`
	Supplied	`ProxyType get#proxy#( )`
		`boolean has#proxy#( )`
Prototype	ClonerType	`Performer clone#clone.class#( Performer proto )`
	Supplied	`Performer clone#clone#( )`
		`void set#clone#Cloner( ClonerType c )`
Flyweight	Performer	`FlyType create#fly.create#(KeyType key)`
	Supplied	`FlyType get#fly#( KeyType key )`
Factory Method	CreatorType	`ProductType create#product.class#( )`
	Supplied	`ProductType create#product#( )`
		`void set#product#Creator(CreatorType c)`
Chain Of Reponsability	Performer	`boolean #handle.canDo#(RequestType req)`
		`void #handle.do#( RequestType req )`
	HandlerType	`void #handle.pass#( RequestType req )`
Composite (Container)	Supplied	Please see Fig. 4

Table 8.2: The developed roles summary description

clause cannot state this dependency from the other participants as it focus only on the intrinsic's class. Our requires list is of a more general use and conveys more information than the "playedBy" clause.

From our study there are a few patterns that do not gain from the use of roles, namely: Builder, Facade, Iterator, Mediator, Memento, Strategy and Template Method. These roles are quite instance specific and the classes built for their implementation are usually case specific and are not reusable outside that pattern. An Iterator, for example, is made for a particular collection and cannot be reused for another type of collection, even if it is of a similar type.

There are a few patterns that could benefit from the use of roles like a way to emulate multiple inheritance and to provide a default implementation to some operations done in a class inheritance hierarchy. These are the Abstract Factory and specially Decorator. This is also valid for the State if states share some common code.

We also found some similar code between instances that we could not isolate and put in a generic role. This was the case of patterns that forwarded method calls, like Adapter, Decorator, Proxy and Chain of Responsibility. The variations were not supported by

```
public role FlowerSubject {
  plays Subject<FlowerObserver,FlowerEvent>(
    Thing=FlowerObserver,
    Event=Open,  Event.notify=flowerOpened ) sbj;
  }
}

public class Flower {
   plays FlowerSubject flwrSubject;
   private boolean opened = false;

   public void open(){
     opened = true;
     fireOpen( new FlowerEvent( this ) );
   }
}
```

Figure 8.7: The FlowerSubject role and the Flower class from our subject role sample.

roles because they were in the methods return type and parameters types and number. Arranging support for such variations would make the role heavily configurable and just the configuration alone would be more complex than to write the code in the first place.

We developed roles for a total of 10 patterns out of 23, which is a good outcome, especially because every role has a high reusability factor. We believe that our Subject role, for example, will be useful for a large number of Observer instances. There are also additional advantages in some roles, like a better modularity, in the Factory Method and Prototype. Other roles are limited in their actions, like Factory Method, which addresses a particular variation, but are nevertheless highly reusable for that purpose.

To test if roles and player are truly independent we produced a dependency structure matrix (DSD) for each sample of the developed roles. For the Observer role sample we developed a Flower subject that must notify its observers when it opens. For that we developed a FlowerObserver that has the flowerOpened(FlowerEvent e) method. As an observer we developed a Bee class that, when notified, will print a message that the bee is seeing an open flower. The Flower class plays the FlowerSubject role, which is the Subject role configured to this particular scenario. Figure 8.7 shows the code for both role and class. The code for the bee, observer interface and the flower event are not shown for simplicity. The FlowerSubject role is not really necessary as the Flower could configure the Subject role directly but it is a good programming practice to do so.

From that sample, we obtained the DSM of Figure 8.8. Here we can find that there is no dependency between the Subject role and the Flower class and that the FlowerSubject depends only on the Subject role and not vice-versa. That would hold even if we did not declare the FlowerSubject role and used the direct configuration as discussed before. If we group the classes into modules as shown in the figure we can see that the module where

Name		1	2	3	4	5	6	7	8
EventType	1								
ObserverType	2	1							
Subject	3	1	1						
FlowerEvent	4							1	
FlowerObserver	5				1				
FlowerSubject	6	1			1	1			
Flower	7				1		1		
Bee	8				1	1		1	

Figure 8.8: Dependency Structure Matrix for the Observer role sample.

the role is included does not depend on any other module. It shows that the flower module is dependent from the role module via the role. It also shows that the Flower module does not depend on its concrete observers, as expected from the observer pattern. The Subject role is therefore independent of its players. We may also add that we also used that same role in the JHotDraw Framework.

8.2 Summary

In this chapter we showed that it is possible to write a library of reusable roles. We did this by developing roles for the GoF design patterns. We were not able to develop roles for each pattern, mainly because patterns instances shared no common code, i. e., their implementation is specific to a particular problem. Nevertheless we did develop roles for those patterns that have similar structures or behaviors between instances.

With the developed roles we were able to use them in sample implementations of the respective pattern. Even though the sample implementations are simple and focused on the pattern they provided a way to test each role effectively. They also enabled us to test the roles and players independence by constructing and analyzing a dependency structure matrix. These tests proved that roles are independent of their players.

This study also showed that the role feature of renaming methods is a crucial one in developing generic roles. This feature makes the use of roles to be easily configured for each specific instance of a pattern, so methods names are adequate in the context of that specific instance. The feature of requiring methods from other collaborators and not just from the player also plays a very important part in this process, as it enables the role to require a suitable method name from their collaborators and not a generic method, enhancing the role adaptation for each specific instance of a pattern.

Chapter 9

Case Studies

In order to support our claim that roles can be used to remove code clones we have conducted case studies using three open source systems. For each system we detected, using a clone detection tool, which clones were present. We then analyzed each found clone and its surrounding code to perceive the nature of the concern it dealt with.

We grouped clones that addressed the same concern. This step is crucial to the development of roles for the clones. We intend our roles to represent a concept and to deal with a concrete concern and not just to reduce replicated code. For each concern we tried to develop a role that could handle it in a satisfactory way and to reuse that role in all clone instances.

The number of clones that we were able to reduce with roles is a measure we can use to validate our claim, if it reflects a great number of concerns.

9.1 The Target Systems

For the case studies we used three open source systems: the JHotDraw framework, the OpenJDK compiler and the Spring Framework. We wanted the systems to be from different

fields of usage. This way we can show that roles are applicable in a variety of systems and not confined to a particular use in a specific field. It also enables us to gain more insights on how the code cloning affects different systems and how roles can be used to remove such clones.

JHotDraw is a Java GUI framework for technical and structured Graphics. The JHotDraw framework defines the basic structure for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. JHotDraw was used as a case study in several works by other authors [CMM+05, Deu05, MDMR09] as well as in a previous work of ours [Bar08] making it a natural candidate for this thesis. The version used is 5.4b, one of the most used versions in the mentioned works. The JHotDraw was developed by Erich Gamma (one of the GoF authors) and Thomas Eggenschwiler as a "design exercise" for the application of design patterns. As a "design exercise" its development followed the software development rules with great care.

The OpenJDK compiler (javac) is an open source Java compiler, and other tools, that has support from Oracle. This compiler formed the base from which we started to build our own JavaStage compiler. The author's familiarity with the compiler and the compiler implementing properties that deviated somewhat from the traditional software development techniques made it a good candidate to become a case-study. The use of public variables, for performance reasons, is a good example of such coding techniques that go against the traditional rules of software development. With this case study we expect to gain insights on how roles can cope with systems more focused on performance issues.

Spring is a layered Java application platform for building enterprise solutions. Spring framework provides a powerful and flexible collection of technologies to improve the development of enterprise Java applications and is claimed to be used by millions of developers. It is also a framework that has been used in a number of studies [LLT11, RJ07, MTB11]. Since it is a widely used framework and has been thoroughly tested is should pose an interesting challenge.

All the selected systems are mature so we expect to find few clones that cannot be removed by refactoring alone.

9.2 The Case Study Setup

To detect clones we used CCFinder [KKI02]. CCFinder is a token-based clone detection tool (see 2.4). Due to this nature CCFinder can detect Type I and Type II clones. With a bit of manual work we can use CCFinder to identify Type III clones. It does not detect Type IV clones. CCFinder makes a token sequence from the input code through a lexical

analyzer and applies a, language specific, rule-based transformation to the sequence. The purpose is to transform code portions in a regular form to detect clone code portions that have different syntax but have similar meaning. Another purpose is to filter out code portions with specified structure patterns. After the detection the user has several browsing capabilities: view the metrics of all the clones, filter the clones based on source files or metrics, highlighting the various places in which a specific clone (or clones) is reproduced, etc.

For the clone detection, we used the standard options of CCFinder. Since we are interested only in clones that are not solvable with the traditional refactorings we need to filter those clones that are. One of such refactoring is the Extract Method that usually deals with concerns inside a unique class. So in order to filter out such clones we only considered clones that appeared in, at least, two files. This also filter clones that do not deal with crosscutting concerns as a concern must be present in at least two classes to be considered a crosscutting concern.

We manually inspected all remaining clones to identify its nature and the concern they dealt with. The tool several browsing capabilities made this task easier. All we had to do was select the clone set to inspect and the tool showed the various places where the clone was reproduced. Because the tool does not take into account the names of the identifiers some clones have code that are unrelated (meaning false clones). These clones were removed. We also took into consideration the rest of the class code to gain an insight on the purpose of the class and its main concern as well as how clone concern fits into the class. We did not, however, made a profound analysis of the rest of the class, just enough to assess if the clone addresses specific aspects of the class or if it is really a crosscutting concern. In this phase we also dismissed false clones. We also ignored clones that used deprecated code or code marked for substitution.

Clones were grouped according to the concerns they dealt with. A role must focus on a specific concern, and not on a clone, so this grouping step helped us reasoning which roles we should develop.

In order to reason if the use of roles affected or not the code clone we set a rule of not changing the clients of any class that plays a role. This way the impact of making a class play a role instead of writing the code in the class itself in the system is null. We also have set a rule not to change the way a concern is implemented, even if it meant that their clients would not change. This was done to retain as much as possible the author's initial intent. The enforcement of this rule was, however, less restrict than the first. Minor changes in the implementation would not compromise the way it was supposed to work. With minor changes we mean changing the name of a variable or changing the order of some, non interdependent, instructions.

Another rule was that we only developed roles that conform to our vision of roles and not just to reduce replicated code for the sake of it. We detected some clones that could be removed simply by using a different inheritance hierarchy. We could develop a role that reduced that replicated code, but since changing the inheritance hierarchy was a better solution we did not do so.

The purpose of these rules is to maintain each system as close as possible to the original, with the only difference being the use of roles instead of replicating code. The previous rules can indeed restrict the development of roles but the purpose of the case studies is to assess the impact of roles in reducing code replication and not building improved systems. To prove the validity of roles in developing frameworks we refer to the work of Riehle in [Rie00] where he describes how the JHotDraw framework could be developed using roles. It is not the purpose of this thesis to prove that the use of roles can provide a better system, even if we assume that a system without code clones is better than one with. We only claim that roles can reduce the amount of replicated code. If we started changing the framework then we would lose our reference point and could not pinpoint the advantages of roles regarding the reduction of code clones. Nevertheless, we did find situations in which refactoring the code would bring benefits to the system structure but refrain from doing so.

It must be mentioned that some clones do overlap, i. e., a larger code clone may include a smaller code clone, leading to clone sets overlapping. We grouped the clones in one single entry, according to the concern they dealt with, even if some clones do appear in more than one concern. When a clone contains code related to several concerns we included them in each concern. That will lead to a clone being resolved by more than one role, as we developed a role for each concern. Sometimes a concern also contains several clone sets, not just one. In fact, due to small changes, not on the clone itself, but on the code surrounding it, when a concern was scattered over many classes it usually had several clone sets associated with it. An example of this situation is the case with the "'Creating UndoActivity"' concern, from the JHotDraw case study, that has 14 clones associated with its implementation. In each case study we present a table showing the results of these steps, namely the concerns identified and the clones associated with each concern and the number of classes affected by the concern.

To remove the clones we developed roles using JavaStage and the role refactoring we proposed in Section 7.2. To present the concerns we grouped them into four categories. These categories reflect how the concern was solved using roles and are:

- Purpose Built Roles
- Roles From Library

- Roles Placed in the Library
- Unresolved

Purpose built roles - Into this category we placed those concerns that needed the role to be specifically developed. We expect this category to be the one that contains the largest number of concerns.

Roles From Library - In this category we placed concerns that used roles that are part of the role library we started to build in chapter 8. We expect some concerns to be placed into this category.

Roles Placed in the Library - Roles can also find their way into the library if we consider that the concern is of a general purpose or can be transformed into a general purpose role. We hope to increase our library with these roles.

Unresolved - these are the concerns for which we could not develop a role. Hopefully this will be the category with the least number of concerns.

We will not describe every concern in detail, for that would be too verbose and space consuming. For that reason we will only briefly describe the ones that used roles from the library and the ones that have roles that were placed in the library. The unresolved concerns will be described with a bit more detail and the reasons why they could not be resolved are presented and debated.

We will also show, for each concern, which proposed role refactoring was used (see Section 7.2).

We also counted the lines of code (LOC) used for each concern in the original system and in the system with roles. This way we can determine if the role system is better than the original system or, at least, has fewer lines of code. One can argue that LOC count is not a good measure for the effort when using different languages, but since JavaStage is an extension to Java and the development of a role in JavaStage is very similar to that of a class in plain Java, we can apply it here with relative confidence. The concerns that had more LOC than the original code were concerns with few lines of code where the role requirements and configuration overhead did not overcome the replicated code.

Since roles introduce new statements it is necessary to explain how we counted the LOC number in each case. Each requires statement was counted as one LOC and each play statement was counted as another LOC. The role methods configuration was counted as another LOC. The LOC count of the roles is higher than the classes for this reason. Assume one concern that presents 8 lines of replicated code in each class which could be resolved with a simple role. We would expect this role to have the same 8 LOC. That is not so because we do not count the class declaration as a clone LOC (the class does other things) but count the role declaration as a solution LOC. Roles may also require methods,

and these requirements are counted as LOC. Thus for the 8 LOC clone the role would have 1 more fixed, 1 more for each player and 1 more for each requirement. If the role requires 3 methods and the concern appears in two classes then the clone has 16 LOC and the role solution would count 14 LOC. If one of the required methods is not available in the class then it must be implemented and is counted in the solution LOC but not in the original LOC. If a getter method is required then it adds 2 more lines (method declaration and return statement), and since the role is used in two classes then it adds 4 LOC to the solution role, leaving a final 18 LOC count, greater than the original 16 LOC. This apparently is worse than the original solution but LOC do not account for the modularity and maintenance issues. Removing the clone gives the system a great advantage in modularity terms.

9.3 JHotDraw

9.3.1 JHotDraw Overview

JHotDraw is a Java GUI framework for technical and structured Graphics. JHotDraw was originally developed by Erich Gamma Thomas Eggenschwiler as a "design exercise", a way to demonstrate the use of several of the GoF patterns. JHotDraw derives from other drawing editor frameworks and it is an evolution of the HotDraw framework [Joh92], originally developed in Smalltalk and that gains from the authors' previous experience with ET++, a previous version of a user interface framework written in C++ [WGM89].

The JHotDraw framework defines the basic structure for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework can be customized using inheritance and combining components.

We can say that the JHotDraw framework is structured around four main inheritance hierarchies. These hierarchies reflect the main classes used in the framework. These are the Figures, Views, Tools and Handles. The relations between these hierarchies are depicted in Figure 9.1.

The Figures classes represent all the figures that can be drawn using the target application. This allows each application developed with this framework to be targeted at a specific domain by providing specific figure types as well as figure connectors. Such an example would be an editor for the UML modeling language. For such an editor figures like UMLClassFigure and connectors like UMLInheritanceConnector would be created.

The Views classes provide the various renderings of the figures and how they are shown in the application window. There can be views with zoom, for example, or any other view the application developer need to have, like mini-maps. The view is also where the user interacts with the figures.

The Tool classes represent the various operations the user can perform on the figures or on the application itself. Tools like creating figures, moving, grouping, etc, form the

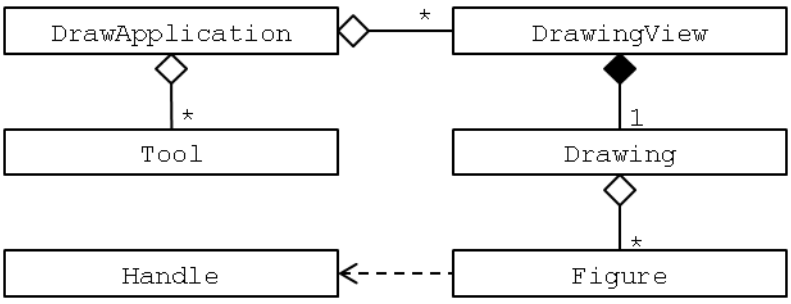

Figure 9.1: Relationships between the main classes of JHotDraw

default tool set of the framework. Each user action may be undone and redone so tools must support this behavior.

The default way of interacting with an already created figure is to use its various handles. Each handle represents a specific action than can be performed to a figure: there are handles for each corner of a figure's bounding box (used to resize the figure), handles to rotate the figure, manipulate its points, etc.

9.3.2 JHotDraw Results

The first result included 271 clone sets. After filtering clone sets with a file span less than 2 they were reduced to 146. These 146 clone sets were then manually inspected. After inspection 41 clones were discarded leaving a final 105 clone sets.

We then proceed to identify the concerns these clone sets dealt with. As already mentioned, sometimes the clone itself did not provide enough information to achieve a conclusion, but examining the surroundings of the clones for a wider vision was enough to detect some crosscutting. The most expressing one is the undo concern: while several clone sets contributed to this concern it was when the surroundings were examined that we found that every class implementing undo had an UndoActivity inner class.

We identified a total of 42 concerns. From those 42 we removed 4 because 1 could be removed by using traditional refactorings, 1 was deprecated code and 2 belonged to identical classes where one will substitute the other which will be deprecated. Table 9.1 shows the identified concerns and how they relate to the number of clones and number of classes. It shows that although most of the concerns have only 1 clone set associated they may have up to 14 clone sets. We can also see that a clone may affect from 2 classes (most cases) up to 24.

After associating each clone with a concern we proceed with the development of roles for each concern, using the refactorings described in section 7.2. For the developed roles we either develop a special purpose role, or have used some role from the library developed in chapter 8. Unfortunately not all concerns were resolved. We show the concerns that were resolved in table 9.2. For these concerns we also present which refactoring was used. Unresolved concerns are shown in table 9.3.

Table 9.2 shows that of the 38 concerns we were able to develop roles for 30, leaving only 8 concerns (see table 9.3) with no available role. The final outcome is better than these numbers indicate as we discuss in the unresolved concerns section.

After the concerns we made the LOC count. The results are shown in table 9.4. We can see that, for the majority of the resolved concerns, roles needed fewer lines of code, a 30% reduction in code size. This seems to indicate a smaller effort when developing a system with roles.

Concern	# Associated Clones	# Affected Classes
Drawing Handles	7	11
Setting up the undo activity before executing a Command	2	8
BringToFront/SendToBack Commands	1	2
Desktop initial configurations	1	2
Undo/Redo Commands	1	2
Handle creation	11	14
Drawing polygons	1	2
Persistence (read/write)	3	6
Changing connection handles	1	2
UndoActivity	13	24
Polygon and PolyLine Handles	3	2
Palette Listener	1	2
DisplayBox persistence	2	5
DisplayBox handling	6	8
Tools and Commands Dispatchers	6	4
Creating UndoActivity	14	18
Handle manipulation starting action	3	5
Figure/Handle and Enumerator	1	2
DesktopListener Subject	2	3
Polygon locator	1	2
Changing connections	3	3
Point is inside Figure	3	6
Finding connectable figure	1	3
Testing command executability	5	7
Floating text holder	2	2
DrawingView Listener Subject	2	4
Setting text in a text Figure	2	2
Enumerator	1	3
Figure Listener that resends notifications	2	3
DrawingView Listener	1	2
Menu enabling	1	2
Version control	1	2
Drawing editor	1	3
Selected button manager	1	2
Text attributes management	2	2
Updating DrawingView Strategy	1	2
Mouse motion handling	1	2
Connection insets computing	1	3

Table 9.1: JHotDraw's identified concerns associated with the corresponding clone sets.

Concern	Used Refactoring
Resolved With Purpose Built Roles	
Drawing Handles	ER
Setting up the undo activity before executing a Command	ER
BringToFront/SendToBack Commands	ER
Undo/Redo Commands	ERTM
Handle creation	ER
Drawing polygons	ER
Changing connection handles	ERTM
Polygon and PolyLine Handles	ERCT
Palette Listener	ER
DisplayBox persistence	ER
DisplayBox handling	ER
DesktopListener Subject	ER
Polygon locator	ERCT
Changing connections	ERCM
Finding connectable figure	ERCM
Testing command executability	ER
Floating text holder	ER
Setting text in a text Figure	ER
Menu enabling	ER
Version control	ER
Selected button manager	ER
Text attributes management	ER
Updating DrawingView Strategy	ER
Connection insets computing	ER
Resolved With Roles From Library	
Tools and Commands Dispatchers	ERTM
DrawingEditor	ERTM
Resolved With Roles Placed in the Library	
Figure/Handle and Enumerator	ERTM
DrawingView Listener Subject	ERTM
Enumerator	ERTM
Figure Listener that resends notifications	ERTM

ER = Extract Role, ERCT = Extract Role Changing Types, ERCM = Extract Role with Configurable Methods, ERTM = Extract Role with Types and Methods

Table 9.2: JHotDraw resolved concerns

Unresolved Concern	Reason
Desktop initial configurations	required too much configuration
Persistence (read/write)	similar but not quite clone code
UndoActivity	Inner classes constructors mainly
Creating UndoActivity	after other roles was just a line of code
Handle manipulation starting action	required too much configuration
Point is inside Figure	just a (rather different) line of code
DrawingView Listener	perfomance issues
Mouse motion handling	

Table 9.3: JHotDraw unresolved concerns

Concern	Original LOC	Roles LOC	Roles/ Original
Drawing Handles	64	40	63%
Setting up the undo activity before executing a Command	56	44	79%
BringToFront/SendToBack Commands	20	12	60%
Handle creation	70	87	124%
Drawing polygons	12	11	92%
Palette Listener	20	17	85%
DisplayBox persistence	35	12	34%
DisplayBox handling	58	29	50%
DesktopListener Subject	63	45	71%
Changing connections	98	53	54%
Finding connectable figure	98	53	54%
Testing command executability	14	14	100%
Floating text holder	47	36	77%
DrawingViewListener Subject	63	26*	41%
Setting text in a text Figure	36	22	61%
Enumerator	33	11*	33%
Figure Listener that resends notifications	35	23*	66%
Menu enabling	20	14	70%
Version control	12	9	75%
Selected button manager	18	12	67%
Text attributes management	206	120	58%
Updating DrawingView Strategy	29	26	90%
Connection insets computing	10	7	70%
Undo/Redo Commands	32	31	97%
Changing connection handles	20	19	95%
Polygon and PolyLine Handles	32	28	88%
Tools and Commands Dispatchers	89	32*	36%
Figure/Handle and Enumerator	33	2*	6%
Polygon locator	13	20	154%
Drawing editor	54	28*	52%

* used role from library

Table 9.4: JHotDraw LOC count

9.3.3 Solved Concerns

In the solved concerns we count 30 concerns. In 2 concerns we were able to remove then using solely roles from our role library (see chapter 8) and from 4 concerns we placed their roles in the library.

From the 24 purpose built concerns we find that 18 (75%) used the Extract Role refactory. This is somewhat expected because when developing a purposely built role the methods and types are specific to the purpose. The other role refactorings were used in cases where the code was similar, like in the "undo/redo command" where they are quite identical in their structure, the only difference being the methods that both call on the undo/redo manager object or on the command being undone/redone. While the UndoCommand class calls methods like popUndo and isUndoable, the RedoCommand class calls popRedo and isRedoable. We used the Extract Role with Types and Methods to capture the structure of both commands in a role that enabled us to configure the type of the manager and the method naming was left to the renaming mechanism.

An example of the use of the Extract Role Changing Types is the "Polygon and PolyLine Handles". This role is responsible for handling the several invokeStart/Step/End methods called during a manipulation of the handles in a polyline or polygon figure. The type of the figure is configurable by the role.

An example of the use of Extract Role with Configurable Methods is the "Changing connections" concern. There is code similar between two connection handles and a tool that handles connections. That code is responsible for two tasks: finding a suitable connection between two figures, tracking which figure is being manipulated by the user and which is currently the target figure, and changing that connection. We assigned those tasks to a single role and therefore the code can be reused by totally different classes like a handle or a tool.

A concern, with significant LOC, in which roles had more LOC than the original was the ”Handle creation” concern. It deals with the creation of handles for each figure. We placed the creation of the handles in a handle creator class that has a method for the handle creation for each class. Since some clones only have similar code we had to reproduce every method in this creator class. That, along with the fact that the original classes have to declare that they play the role and the definition of the role itself lead to more lines of code than the original implementation. But the role has an advantage over the original code: it can dynamically change the handle creator.

The “Polygon Locator” is responsible for returning a point inside a polygon given a point index. It is used in two classes but one of them uses an anonymous class that defines this behavior. Currently JavaStage’s roles cannot be applied to anonymous classes so we had to develop an inner class to play that role and then use it. This introduced a higher

overhead that made the role have more code than the clone.

The Observers

From the concern list we see that there are some instances of the Observer pattern, referred to as Listeners because the authors of the framework followed Java's approach to the Observer pattern. We will use both terms indiscriminately. Clone detection found some subjects and a few listeners. After a code inspection we were able to find for each instance its subjects and its observers. These are the instances of the Observer pattern that were found:

- DrawingView Listener
- Pallete Listener
- Figure Listener
- Tool Listener
- Command Listener
- Drawing Listener
- Desktop Listener

Even though they are all instances of the same pattern they were not implemented in the same way. One reason that explains this is that the framework was developed by several authors each with its own experience and programming habits. We believe that if they had our role library they would have used it and then all listeners would be pretty much the same.

Some of the instances followed Java's Swing style subject that consists in having a single listener list for all kinds of observers' type. The notification methods have to check each observer's type, and cast it appropriately, before calling the corresponding updating methods. Some instances followed the Java's AWT style subjects using a AWTEventMulticaster, by developing an extension to this java class. The extension deals with its specific kind of listeners. Other instances followed the Java's way of using events but implemented a separated list of listeners for each kind of observer type. Other instances used a list for each kind of listener type but the updating methods did not have an event parameter, and some even had two parameters.

The palette listener subject is different from the rest as it cannot add or remove listeners: it uses a single listener that is set when the subject (PaletteButton) is created. It could be argued that it is not an instance of the pattern, but we referred it because the

names used for classes and objects suggested that it was intended to follow the pattern style.

If we changed the code so that each instance was using the same implementation we could use a single role for them all but that would either imply changing the concern implementation (the swing version) or changing their clients (the two parameter version). As mentioned we set a goal of not changing the code except in what refers to the role playing. This lead us to use 3 roles, one for each version of the pattern. One concern - "Tools and Commands Dispatchers" - reused the subject role developed for the GOF pattern already in the library. We used the Subject role for the Tool listener subject and Command listener subject concerns. We did not use it directly in the Tool or Command classes because of the way it was implemented: each class developed an EventListener class that performs the actions associated with the management and the notification of observers. It is this dispatcher inner class that uses our subject role.

One of the observer pattern instances was based on the Java's swing style, namely "DrawingView Listener Subject". We developed a role, SwingSubject, which uses an EventListenerList. When a listener is added to the subject it stores not only the listener but also its class. Because the list can contain any type of listener, the class is used to check if the listener is of the appropriate type. If it is, then the listener is cast to the intended type and the notification method is called, otherwise the listener is ignored.

Comparing this implementation from the one used for our Subject role from the GoF library we only see one advantage that is the fact that a single list can store several types of listeners, while we declare a container for each type of listener. But that comes with the cost of having to store the class along with the listeners for type checking. The single list is good for subjects that have multiple types of listeners like a Swing Component does. In the case of the JHotDraw framework the subjects only have a type of associated listener so our implementation is better suited.

We could use our Subject role wherever we used the SwingSubject, without breaking client code, but we followed the rule of keeping the original implementation as much as possible. A further analysis of the role suggested that it could be generalized. After that generalization the role was no longer tied to the JHotDraw framework and so we decided to put it into the library. It can now be used whenever we want a subject that follows swing subject rules.

The Adapters

In Java, each listener has a defining interface type and most have an associated Adapter class. The Adaptor class defines the default behavior of each listener, which is doing no action at all. Concrete listeners have to either implement the listener interface, and

implement every method defined, or inherit from the Adaptor class, and implement only those which are relevant for their case. If a listener happens to be also a subclass then it is limited to implement the interface since it cannot inherit from the adaptor class.

We developed a solution that resembles the Java way but with roles. Instead of defining an adapter class we developed an adapter role. That role implements the no action default implementation for each method of the listener interface. This way a listener can always play the adapter role, even if it is a subclass, having to implement only the relevant methods. That role was then generalized and became a configurable role where the type of the listeners and events are defined by generics and the method names by our renaming mechanism. This role found its way to our role library.

In JHotDraw we found several instances of the observer pattern that could use this role. For each instance we developed a dedicated version by configuring the generic role for each purpose. We could write a generic role for each instance of the observer pattern, but once again our goal was not to improve the framework, but reduce replicated code using roles. Therefore we developed an Adaptor role for the following concerns:

- DrawingView Listener
- Figure Listener
- Tool Listener
- Command Listener

Other library candidate roles

There are figures that contain other figures, like composite figures or decorator figures. This means that they are subjects of a figure observer pattern, but because they must react to changes in their contained figures they also play the role of figure listeners. Whenever their contained figures change they need to notify their own listeners. This is done simply by forwarding the notification method.

We found this role useful for other situations so we generalized it using generics to configure listeners and event types. With the renaming mechanism we can configure method names. Finally we putted the role - NotificationResender - into the library. For the framework we developed a specific role - FigureChangeResender - that plays and configures the library role to the figure context.

During some operations, we need to traverse a list of figures or handles that we referred to as the "Figure/Handle and Enumerator" concern. We even found a use for a reverse traverse of a list of figures. Each enumerator, even for the reverse, has a code very similar

to the others, so we develop a role for that concern. We even did find it useful enough to put it into the role library.

The developed role has methods to return the next element in a collection and see if it has more elements. It also lets clients define the collection it handles and reset the traversal. It used generics for the types and allows the renaming of the methods that return the next element and whether there are more elements.

9.3.4 Explaining Unresolved Concerns

A surprising result is that for the 2 concerns with the most clone sets and class involved neither technique works. This is due to the nature of the clones. They are clones only in the structure and not on the code itself. The "Creating Undo Activity" concern must create an undo activity object for each of the various tools and commands supported by the framework. Each tool class has an UndoActivity inner class hence the undo activity creation is just a line of code instantiating an object of the respective inner class. Because each inner class constructor has different parameters in number and types, roles could not resolve this concern. UndoActivity concern clones are due to the inner classes, because they all have the same name and constructors with the same structure, even if not equal. Another example of such a concern is the Handle manipulation starting action: code was similar but not quite identical because methods called had different parameters and most code would disappear with refactoring.

Another example is Persistence: because figures must be streamed they have a write and read methods that have similar structures, but not quite identical code. It is enough for two figures to have a single field and store/read that field to have identical code in this concern, even if the fields are unrelated and of different types. Some figures have to write longs while others have to write integers. We have considerably reduced this duplicated code with our DisplayBoxed role, though, since figures have to deal with their display box they need to write/read the display box.

Another unresolved concern is the DrawingView listener. The replicated code is redefining the original method, apparently for performance issues that we failed to understand. It is our belief that if we deleted the method, the code would be equally effective.

One unresolved clone - "Desktop Initial configuration" - dealt with a Desktop's panel initialization, which initializes panel titles and adjusts a scrollPane. Each possible initialization is similar so we could configure a role for every way a scroll pane is configured and then reuse them whenever we created a Desktop. But configuring a role would be more confusing as there would be several roles, one for each configuration, and knowing them would require more effort than to know how to configure the scroll pane.

The others unresolved concerns where a single line in the form of

`return getSomeObject().doSomething()`. Since the first method returns different objects that calls different methods we could develop a role, but role configuration would be harder than writing the code itself.

If we had not considered some of these concerns as clones then roles would count only 4 unresolved clones.

9.3.5 Other Considerations

Because JHotDraw was developed as a design exercise and used as a pedagogical tool the code had several characteristics that facilitated the development of roles. One of those was the field access. Classes never accessed their fields directly but always through the use of getters and setters methods. This helped when we moved the fields to a role. The methods that accessed that field but were not moved to the role needed no modification whatsoever. It also helped when the field was not moved to the role but was accessed by the role. The required methods that were needed were already developed in the class.

9.4 OpenJDK Compiler

9.4.1 OpenJDK Compiler Overview

The OpenJDK [1] is an open source implementation of the Java Platform, Standard Edition and related projects. One of those projects is the Compiler Group comprised of developers involved in the design, implementation, and maintenance of the javac compiler for the Java programming language, and associated components. The purpose of the javac compiler is to read source files written in the Java programming language and compile them into class files. Optionally, the compiler can also process annotations found in source and class files using the Pluggable Annotation Processing API. The compiler is a command-line tool but can also be invoked using the Java Compiler API.

The compiler of JavaStage was developed by adapting the javac compiler code found in the Compiler group [2], version 1.6.0. javac is written in the Java programming language, thus selecting this system as a case study was a natural choice.

The javac compiling process can be divided into three stages, as shown in figure 9.2. Different parts of source files may proceed through the process at different rates, on an "as needed" basis. This process is handled by the JavaCompiler class. We can summarize it as:

- *Parse and Enter* - All the source files specified are read, parsed into syntax trees, and then all externally visible definitions are entered into the compiler's symbol tables. The reading is done by a Scanner that converts the text to tokens. Tokens are then read by the Parser that builds the syntax tree. Each tree is passed to Enter that enters symbols for each definition. Enter starts by entering symbols for all top levels classes, interfaces or enums. Enter has a MemberEnter auxiliary class that processes the symbols of class members.

- *Annotation Processing* - All appropriate annotation processors are called. If any annotation processors generate any new source or class files, the compilation is restarted, until no new files are created.

- *Analyze and Generate* - Finally, the syntax trees created by the parser are analyzed and translated into class files. The work to analyze the trees and generate class files is performed by a series of visitors. During the course of the analysis, references to additional classes may be found. The compiler will check the source and class path for these classes; if they are found on the source path, those files will be compiled as well, although they will not be subject to annotation processing.

[1] http://openjdk.java.net/
[2] http://openjdk.java.net/groups/compiler/

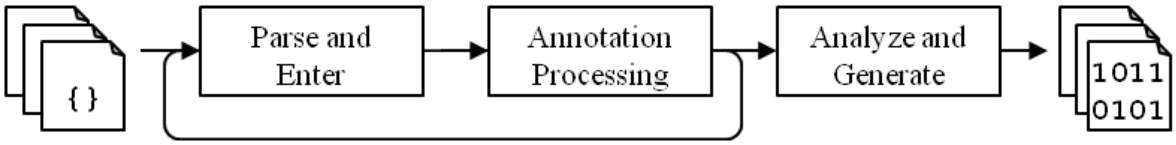

Figure 9.2: javac compiling stages

The principal visitors for the Analyze and Generate stage are:

- *attr* - the top level classes are "attributed", i.e., names, expressions and other elements within the syntax tree are resolved and associated with the corresponding types and symbols. Many semantic errors may be detected here, either by Attr, or by Check.
- *Flow* - responsible for checking definite assignment to variables, and unreachable statements, which may result in additional errors.
- *TransTypes* - deals with generic types by replacing them with code without generics.
- *Lower* - processes "syntactic sugar". This takes care of nested and inner classes, class literals, assertions, foreach loops, and so on.
- *Gen* - generates the bytecodes that are associated with the code attributes of the method trees. If that step is successful, the class is written out by ClassWriter.

The OpenJDK compiler is not limited to the compiler, it also includes several tools like javadoc that generates documentation in HTML format from doc comments in the source code. In this tool there are several classes responsible for generating the html information for methods, fields, classes etc. A subtype of doclet is available for each of these types of information.

9.4.2 OpenJDK Results

Initially 581 clone sets were obtained. After the filtering of the clone sets appearing only in two, or more, classes these were reduced to 148. For an explanation of this great reduction see section 9.4.5. After inspection, 20 clone sets were considered false clones and disregarded. This left a final 128 clone sets to consider and determine the concern they addressed.

We identified a total of 37 concerns, from which we removed 5. Methods associated with one of the removed concerns were listed in the "to do" list to be removed from one of the classes involved. Another concern was an enum duplicated in another package. This duplication is a change in Java from version 1.5 to version 1.6, that moved the enum from one package to the other. Another of the removed concerns was a class duplicated for no

apparent reason, but we suppose one of them is going to be removed. The final 2 concerns could be easily removed by placing a method in one of the available utility classes. The considered 32 concerns are shown in table 9.5. Also shown in the table, for each concern, is the number of clone sets associated with it and the number of classes that deal with that concern.

After associating each clone with a concern we proceed with the development of roles for each concern, using our role related refactorings (see section 7.2). Used roles were developed purposely to each concern but we were also able to place one role in the role library (chapter 8). We show the concerns that were resolved in table 9.6. For these concerns we also present which refactoring was used. Unfortunately, not all concerns were resolved. Unresolved concerns are shown in table 9.7.

Table 9.2 shows that we developed roles for 25 concerns out of the 32 concerns, leaving only 7 concerns (see table 9.3) with no available role. After analyzing the unresolved concerns we can say that the results are quite satisfactory and that the 7 unresolved concerns are, for its majority, not due to roles limitations.

After the concerns we made the LOC count. The results are shown in table 9.8. We can see that for the majority of the resolved concerns roles needed fewer lines of code, a 37% reduction in code size. This seems to indicate a smaller effort when developing the system with roles.

9.4.3 Solved Concerns

In the solved concerns we count 25 concerns. In 1 concern we were able to generalize it and place its role in the role library (see chapter 8). We did not use any role of the library, though.

From table 9.6 we find that two concerns were resolved by a combination of refactorings. This is explained because the classes involved dealt with the same concern but they were not implemented in the exact same way among them. This explains why there are various clone sets associated with this concern. In both cases we developed two different roles, using role inheritance, to accommodate the differences observed between implementations. The "Writing headers" concern, for example had 15 associated clones and affected 5 classes. This means that all classes share the same behavior but they implemented it with variations among them. To fully capture those varieties we developed two roles using the two mentioned techniques and also used the role inheritance for one role to extend the other. It must be mentioned that the ERTM refactory was used because one method had a slight change in the name, so we had to use the renaming mechanism. If roles were developed in the first place we suppose that the name would be the same, and no configuration would be needed, simplifying the role.

Concern	# Associated Clones	# Affected Classes
Writing headers	15	6
Taglet information	3	6
Building fields	6	10
Message formatting and retriever	1	2
Reading formatted chars	3	3
Writing headers for html doclets	3	6
Annotation proxy maker	1	2
Loop visitor	1	2
Returning Objects as types	1	2
Class member	1	2
Writing navigation links	5	4
Storing information for local vars	2	3
Value visitor for annotations	2	2
Converting files to URL and vice versa	2	4
Creating scanners	1	2
Local class info	4	7
Returning annotation descriptions	1	3
Version flags	6	2
Visitor subjects	4	8
Converting types to String	27	3
Printing navigation bar	1	2
Printing error messages	1	2
Converting a list of a type to a list of another type	1	2
Generating file from builder	3	7
Member document implementer	1	2
Returning formal type parameters	1	2
Initiating tree visiting with environment preservation	1	2
Get declared type	1	2
Mnemocodes initializer	1	2
Determining opcode names and lengths	2	2
Building a set of modifiers	9	2
Building access modifiers string	4	4

Table 9.5: OpenJDK compiler's identified concerns associated with the corresponding clone sets.

Concern	Used Refactoring
Resolved With Purpose Built Roles	
Writing headers	ER + ERTM
Taglet information	ER
Building fields	ERTM
Message formatting and retriever	ER
Reading formatted chars	ERCT + ER
Writing headers for html doclets	ERCM
Annotation proxy maker	ER
Loop visitor	ER
Returning Objects as types	ER
Class member	ER
Writing navigation links	ERTM
Storing information for local vars	ER
Value visitor for annotations	ER
Converting files to URL and vice versa	ER
Creating scanners	ERCT
Local class info	ERTM
Returning annotation descriptions	ER
Visitor subjects	ERTM
Printing navigation bar	ERCM
Printing error messages	ERCT
Generating file from builder	ERCT
Member document implementer	ER
Returning formal type parameters	ER
Initiating tree visiting with environment preservation	ERTM
Resolved With Roles Placed in the Library	
Converting a list of a type to a list of another type	ERTM

ER = Extract Role, ERCT = Extract Role Changing Types, ERCM = Extract Role with Configurable Methods, ERTM = Extract Role with Types and Methods

Table 9.6: OpenJDK compiler resolved concerns

Unresolved Concern	Reason
Version flags	code in enums
Converting types to String	flag comparison only
Get declared type	too small code
Mnemocodes initializer	code uses constants defined in different classes
Determining opcode names and lengths	code uses constants defined in different classes
Building a set of modifiers	static and non static methods
Building access modifiers string	consecutive flag checking

Table 9.7: OpenJDK compiler unresolved concerns

Concern	Original LOC	Roles LOC	Roles/ Original
Writing headers	224	125	56%
Taglet information	40	19	48%
Building fields	251	122	49%
Message formatting and retriever	84	63	75%
Reading formatted chars	112	60	54%
Writing headers for html doclets	62	41	66%
Annotation proxy maker	54	34	63%
Loop visitor	16	23	144%
Returning Objects as types	24	17	71%
Class member	28	23	82%
Writing navigation links	191	120	63%
Storing information for local vars	24	18	75%
Value visitor for annotations	64	42	66%
Converting files to URL and vice versa	126	40	32%
Creating scanners	16	13	81%
Local class info	38	29	76%
Returning annotation descriptions	18	18	100%
Visitor subjects	20	6	30%
Printing navigation bar	32	23	72%
Printing error messages	34	32	94%
Generating file from builder	73	71	97%
Returning formal type parameters	12	14	117%
Initiating tree visiting with environment preservation	18	27	150%
Converting a list of a type to a list of another type	10	6*	60%

* used role from library

Table 9.8: OpenJDK compiler LOC count

The Extract Role refactory, was the most used with 14 (56%).concerns This is somewhat expected because when developing a purposely built role the methods and types are specific to the purpose. The somewhat unexpected result was that the second most used refactoring was the Extract Role with Types and Methods, the most complex one, used in 6 concerns (24%). This may be explained by the fact that most of the usage was done in concerns that treated several types the same way. For example, the "Building Fields" concern was responsible for building documentation for the members of a class. Each member has a builder class responsible for creating the documentation of such entities like fields and methods. The code for each class was similar but each class dealt with a different kind of writer, because there was a writer class for each kind of member, and some methods were named differently. To solve these clones we developed a role that enables method configuration, as well as type configuration for the writers.

An example of the use of the Extract Role Changing Types is the "Creating scanners" concern. There are several classes that act as scanners, either to read the Java code or to read the document files. Each Scanner class has an inner factory that creates a scanner of its respective type. Factories differ only in the types of scanners created, so we created a role where the type of the scanner can be configured.

The concern "Writing headers for html doclets" is among the ones that use the Extract Role with Configurable Methods refactory. This is one of the concerns that deals with the writing of the documentation in the html format. To do this each class uses methods like printBottom(), printBodyHtmlEnd(). The class methods are named like printClassUseFooter() or printPackageUseFooter(). Our role has configurable methods allowing the player classes to tailor the methods' names to their needs.

This case study has 3 concerns that needed more code to remove the clone than the code of the clone itself. The reason for this is the same in all the concerns. It was the fact that in javac many fields are used as public so there are no getters and setters methods to access them. This resulted in an overhead for the roles as they needed to access class fields, and thus these methods had to be created, and were counted as a solution LOC. Since roles also required other methods from the class the sum of these requirements and the getters methods code resulted in a LOC number higher than the clone code itself.

Visitors

If there is a pattern extensively used in this case study it is the Visitor pattern. It is used by every pass of the compiler to transverse the syntax tree. It is also used in other situations for other types of trees. The clone detector did not find the use of this pattern for the syntax tree nodes. This occurred despite the fact that there are several classes, one for each language construct like import, class, variable declarations, method declaration,

method call, etc. Each class even has two accept methods that call the corresponding visitor method, one for TreeVisitors and another for simple Visitors.

The reason why the clone detector did not capture this concern was the filtering of the clones that did not span two files. We expected each class to have its how file, but in this case all the node classes were developed inside the same file, as inner classes of the JCTree class, that acted simultaneously as the superclass for all the node classes.

The only visitor pattern detected was the one used in the "Visitor subjects" concern that was used for visiting attributes associated with the nodes. And even in this case not all the subjects were detected. In this case the number of tokens was probably the cause for not detecting these clones. In fact the accept method has a small signature that fails the 12 token minimum used by the clone detection tool. The ones detected also included code unrelated to the concern like constructors, or a get method, thus making it exceed the 12 token limit.

In our role library we developed a Visitor role, so this concern should be in the resolved with roles from the library. Nevertheless we could not reuse our library visitor role. This comes from a limitation of roles but also from Java's Generics. The accept method was defined using generics for the attributes visitor type like this

```
<R,D> R accept(Attribute.Visitor<R,D> visitor, D data);
```

This definition allows each visitor to define the type of the parameter and the type of the result of the visit method. Java's generics do not allow us to have generic type configured with generic types, so we could not develop a role where we declared visitor to be of an unknown type like in

```
role<VisitorType>{
   public <R,D> R accept(VisitorType<R,D> visitor, D data){
      visitor.visit#class#( this, data );
   }
}
```

This situation forces us to develop a specific role for each visitor instance. We did not develop a role for the syntax tree because it was not detected by CCFinder, but that one would be similar to the attributes role, just changing the visitor's type.

Our visitor role also assumes name conventions for the visit methods that are not followed in the OpenJDK compiler. To use the visitor pattern in this case we had to rename the visit methods in order to follow the naming conventions. Since the code is not part of a framework or the compiler's API this change is innocuous to the systems's clients. If roles were to be used right from the start the name conventions required for the visitor pattern would not be a practical problem.

Library candidate roles

From this case study we only encountered a possible candidate for a library role. The "Converting a list of a type to a list of another type" concern is used by its classes to convert a list of one type into a list of another type. The conversion is done on an object by object basis and it ensures that the order of the objects is the same. Since this is a concern that other classes may have we placed this role in the role library. Both initial and final types are configurable with generics and the conversion method uses the renaming mechanism..

9.4.4 Explaining Unresolved Concerns

Some of the non removed clones could be considered as not real clones by some authors, but we considered them nevertheless because they addressed the same concern. Such are the cases of the concerns "Converting types to String" and "Building access modifiers string". In these concerns the replicated code is mostly an if statement followed by an action. For example, in the converting types to strings the code was just a sequence of ifs, one for each type present, and returning the string equivalent of that type, as shown next

```
else if(elmT.equals("byte"))return "jbyteArray";
```

However the returned type names were different between classes (in one class it would be "jbyteArray" and in another it would be "jbyte") and in some cases even the type was different (one using "byte" and another using "B"). This is a case in which the code is not similar, only its structure is, and we could mark it as a false clone.

In the other concern the code was just a series of ifs testing if a given flag is set. An example is:

```
if ((access & ACC_ABSTRACT) !=0) v.addElement("abstract");
```

Depending on the class, the number and flags tested were different. Some tested only access levels, others tested modifiers like static, synchronized, etc.

The "Get declared type" concern contained just two lines of code with an if statement followed by a return or a throws. To address this concern we would need to build a configurable role because it uses different types and would require methods from the player. Since this code was so small the use of such role was disregarded as it would be more complex than to replicate the code.

Some concerns were not resolved due to role limitations. One of such limitations is that roles can not be played by enumerations (enums). Our reasoning for roles is that they represent a concept that is to be played by a class and not just a set of methods that are added to the class. Thus we limited the role playing capability to classes. In this

case study, and in the "Version flags" concern, we found duplicated code inside enums, but roles are not an option for their removal.

Methods having different attributes for each player class are not supported by roles. If a method is static in a role it is static in all its players and if it is not static we cannot make it static in any player. This limitation prevented us to develop roles for the concern "Building a set of modifiers". In this concern a class has the method static and the other uses a non static method.

The "Mnemocodes initializer" and "Determining opcode names and lengths" are related concerns as they affect the same classes and use related code. However we could not remove either. The reason for that is that both classes use a set of constants that have the same name and values, but are defined in different places. Those constants are the opcodes used in the JVM. The reason for replicating the definition of such constants we could not ascertain, but while in a case a class defines its own constants the other inherits them from an interface which is autogenerated and contains constants defined in the interpreter. Since the code used such constants and we cannot be sure that they will stay equally named and valued, one of them being automated, we decided not to develop a role.

9.4.5 Other Considerations

One distinguishing feature of this case study is that the code uses a lot of public fields for, possibly, performance reasons. This affected the way the roles were developed. In some cases we had to rename those public fields to uniform the names. The renaming mechanism can be used to define methods names but it cannot be used to define field names. This refactoring was done because these classes did not make part of the compiler API so it would not compromise any client.

The development of roles also suffered from the lack of proper getters and setter methods. Whenever we moved a field from a class to a role and that field was accessed inside the class, most of the times it did not have a corresponding get or set method and we had to define one. This was a reason why some concerns had more code than the clone. The results could also be better if these methods were already developed, because then we would not count them as solution LOC.

An important consideration in this case study is the high number of clones detected prior to the filtering of clones with the two file span. This filtering reduced 581 clone sets to 148. This may imply that this filtering could have removed important data. It did remove the visitors data from the syntax tree classes. Since these classes also share some common behavior it should be expected that a large number of clones were originating in this file alone.

The development of subclasses as inner classes of their superclass was also used in

other situations, like the symbols classes that are all subclasses of the Symbol class and inner classes as well. This use explains the large number of clones removed when filtering clones only present in two files.

So, did we lose important data by using this filter? The short answer is yes. Does it compromise the case study? Maybe, but some filtering would have to be done because 581 clone sets to be analyzed is just too much data for manual inspection. A reason why we believe that this does not affect the results much is that the inner classes are always subclasses of their outer classes so much of the cloning that could arise inside that hierarchy could be removed by using traditional refactorings.

9.5 Spring Framework

9.5.1 Spring Framework Overview

Spring framework[3] is a layered Java application platform for building enterprise solutions. Spring framework provides a powerful and flexible collection of technologies to improve the development of enterprise Java applications and is claimed to be used by millions of developers. Since it is a widely used framework and has been thoroughly tested is should pose an interesting challenge.

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect-Oriented Programming), Instrumentation, and Test, as shown in Figure 9.3.

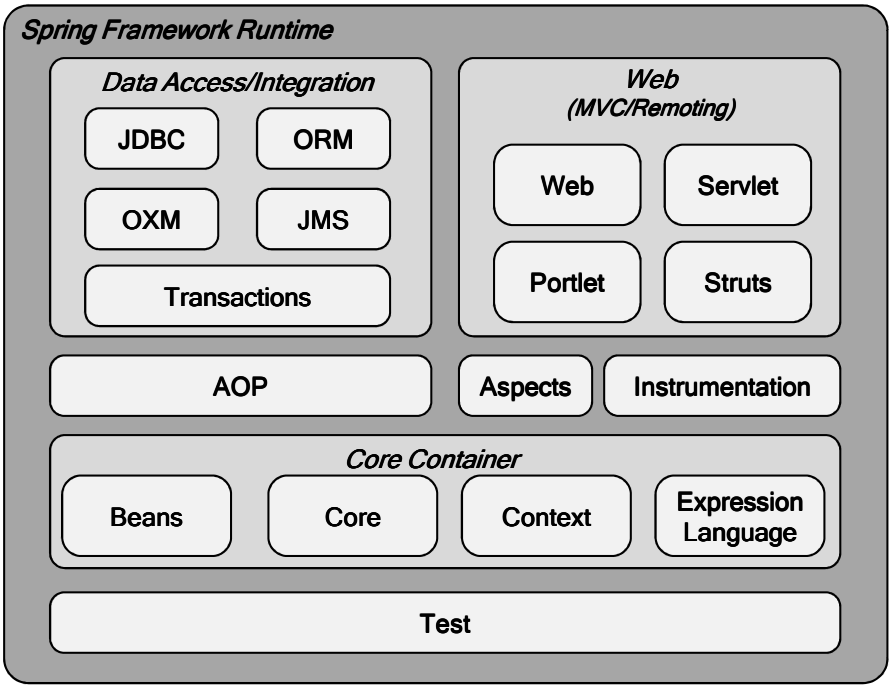

Figure 9.3: Spring Framework overview

The Web layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules. Spring's Web module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the inversion of control container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.

The Web-Servlet module contains Spring's model-view-controller (MVC) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring

[3] http://www.springsource.org/

framework. The Web-Portlet module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of the Web-Servlet module. This mirroring will be responsible for many code clones.

The Core Container consists of the Core, Beans, Context, and Expression Language modules. The Core and Beans modules provide the fundamental parts of the framework, including the inversion of control and dependency injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows the decoupling of the configuration and specification of dependencies from the actual program logic.

Spring's AOP module provides an AOP Alliance-compliant aspect-oriented programming implementation allowing the definition of, for example, method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated. Using source-level metadata functionality, it is possible to incorporate behavioral information into the code, in a manner similar to that of .NET attributes.

The separate Aspects module provides integration with AspectJ. Since JavaStage does not support aspects this source code was ignored in the case study.

The Instrumentation module provides class instrumentation support and classloader implementations to be used in certain application servers.

9.5.2 Spring Framework Results

Even after the filtering of those clones with less that two files there were 416 clones. Due to this great amount of clones and because we wanted to manually inspect all clones, we filtered out the clones with less than 20 tokens. The clones to be inspected were thus reduced to 164 clones. After manual inspection 31 clones were discarded leaving a final 133 clones.

We then began the process of grouping the clones by concerns and identified 89 concerns. From these concerns 5 were discarded because 1 could be resolved with simple changes in the inheritance hierarchy and 2 could be well resolved with normal refactorings and the last 2 included deprecated code. We thus obtained 84 concerns. Table 9.9 shows the identified concerns and how they relate to the number of clones and number of classes. It shows that most of the concerns have only 1 clone set associated, but some have up to 5 clone sets. We can also see that a clone may affect from 2 classes (most cases) up to 5.

After associating each clone with a concern we proceed with the development of roles for each concern, using the refactorings described in section 7.2. For the developed roles we either develop a special purpose role, or have used some role from the library developed in chapter 8. Unfortunately, not all concerns were resolved. We show the concerns that

were resolved in table 9.10. For these concerns we also present which refactoring was used. Unresolved concerns are shown in table 9.11.

Table 9.10 shows that of the 84 concerns we were able to develop roles for 81, leaving only 3 concerns (see table 9.11) with no available role. After the concerns we made the LOC count. The results are shown in table 9.9. We can see that for the majority of the resolved concerns roles needed fewer lines of code, a 29,5% reduction in code size. This seems to indicate a smaller effort when developing a system with roles.

9.5.3 Solved Concerns

This case-study had the most identified concerns, 84, but at the same time was the one with the least unresolved concerns, only 3 unresolved concerns. From all the 81 solved concerns 1 was placed in the library.

From the 80 purposely resolved concerns the most used refactory was the Extract Role which was used in 33 (46,9%) concerns. This was used when the code was identical, for example, in the case of the "Basic trigger behavior". In this concern the SimpleTriggerBean and CronTriggerBean classes are convenience classes to ease a beans-style usage. Each class is a subclass of an already existing bean class (different superclasses of course), that lacks sensible defaults. So they set as defaults the Spring bean name as job name, the Quartz default group as job group, the current time as start time, and indefinite repetition, if not specified. These classes should also register the trigger with the job name and group of a given JobDetail, allowing the scheduler factory bean to automatically register a trigger for the corresponding JobDetail, instead of registering the JobDetail separately. Their code was very similar and could not be reused because of the different class hierarchies each came from. We placed that code in the StdTriggerBean and solved this concern by making both classes play the role.

Extract Role with Types and Extract Role with Types and Methods were also used extensively with 21 (25,9%) and 20 (24,7%), respectively. This is somewhat unexpected since these are the refactories that are the most complex. The Extract Role with Types and Methods is the one we expected to use the least, and we expected it to be used more in the library roles or library candidate roles, because of the high level of configuration it provides. This unexpected usage is due to the portlet and servlet support that Spring has. The way Spring handles each of the two services is very similar. So there is much code that handles servlets that is equal to the code that handles portlets. This code is a significant part of the Spring framework.This alone was responsible for 25 concerns of the 81 resolved concerns (30,9% of resolved concerns). Since much of the handling code differs only in the types used (servlet-portlet, HttpRequest-PorletRequest,

Concern	# Associated Clones	# Affected Classes
Context resource for servlets/portlets 1 2		
Nested Exception	1	2
Link for JPA Dialect	1	2
Data binder for servlets/portlets	1	2
Basic trigger behavior	1	2
Managing methods in MBeans operations	1	2
Adapters for requests	1	2
Mock implementation for multipart requests	1	2
Property accessor for traversing beans	1	2
Abstract handler mapping	2	2
Invocation Handler Adapter	1	2
Metadata bean initializer	1	2
Mapping exceptions class names to view names	2	2
RMI Client Interceptor	2	2
Web context scope for porlets/servlets	1	2
Common multiparts resolver	1	2
Resource Editor	1	2
Dispatcher for servlets/portlets	7	2
Trigger interceptors after some action	3	2
Context aware processor	1	2
Multimapper	1	2
Handler methods invoker	3	2
Lob Creation Synchronization	1	2
Session Bean Manager	1	2
Mock config implementation	1	2
Configurable application context	4	4
Transaction Manager	2	2
String Expression	1	2
HandlerExecutionChain	1	2
Servlet/portlet Framework	1	2
Proxy Bean Factory	2	5
Support for ObjectXXXFailureException	1	2
Alternate implementation of Property Editor	1	2
Multipart Parameter Manager	1	2
Supporting class for closing suppressing invocation handlers	2	4
Simple service exporter	1	2
Portlet/servlet bean	1	2
Connection Factory Manager	1	2
Connection Factory Getter	1	2
Creating instances of container factory beans	2	3
Managing cookies for mock responses	1	2
Managing sets and lists	3	4

Table 9.9: Spring's identified concerns associated with the corresponding clone sets (Part I).

CONCERN	# ASSOCIATED CLONES	# AFFECTED CLASSES
JpaVendorAdapter	1	2
AnnotationMethodHandlerAdapter	1	2
Template Dao Support	1	2
Custom Editor for maps and Collections	2	2
Isolation level setter	1	2
ProxyFactory holder	1	2
Entry with strings	1	2
Dealing with Configuration Property Values	1	3
Test support	3	2
PostProcessInstantiation tasks	1	2
Looking up and logging Jndi templates	1	2
Checking for SQL Script delimiters	1	2
Defining names to use within Metadata providers	1	2
Validating handler for HandlerMappings	1	2
Setting base names for resource bundles message source	1	2
Session and request management for request Attributes	1	2
Character encoding and content management for requests	1	2
Single Connection Factory	1	2
Session holder for mock requests	1	2
Shutdown and destroy executors	1	2
Manipulating outputstream for MockResponses	1	2
AnnotationMethodHandlerExceptionResolver common code	3	2
Web service feature conversion	1	2
Standard ThreadPoolTask Executor	2	2
Static Factories	1	2
Creating Rmi registry	1	2
Hibernate AccessException conversion	1	2
TaskScheduler	3	2
Generating maxValue incrementer as auto incrementer	1	2
Abstract handler exception resolver for Portlets/Servlets	1	2
HttpInvokerRequestExecutor response validator and Gzip Response body	1	2
Deciding parameter name to use in a database call metadata provider	1	2
Model+View for Portlets/Servlets	1	2
Setting concurrent limits	1	2
Parameter manager for mockers	1	2
Invoking methods in invocation handlers	1	2
Convert comma delimited strings	1	2
PortletServlet utils	5	2
Binding information to threads	1	2
Check parameters and headers	3	2
Request parsing	5	2
Locating a constructor/method on a type	2	2

Table 9.9: Spring's identified concerns associated with the corresponding clone sets (Part II).

Concern	Used Refactoring
Resolved With Purpose Built Roles	
Context resource for servlets/portlets	ERTM
Nested Exception	ER
Link for JPA Dialect	ERCT
Data binder for servlets/portlets	ERCT
Basic trigger behavior	ER
Managing methods in MBeans operations	ERCM
Adapters for requests	ERTM
Mock implementation for multipart requests	ER
Property accessor for traversing beans	ERTM
Abstract handler mapping	ERCT
Invocation Handler Adapter	ER
Metadata bean initializer	ER
Mapping exceptions class names to view names	ERCT
RMI Client Interceptor	ER
Web context scope for porlets/servlets	ERCT
Common multiparts resolver	ERCT
Resource Editor	ER
Dispatcher for servlets/portlets	ERTM
Trigger interceptors after some action	ERTM
Context aware processor	ERTM
Handler methods invoker	ERCT
Lob Creation Synchronization	ER
Session Bean Manager	ER
Mock config implementation	ER
Configurable application context	ERTM
Transaction Manager	ERTM
String Expression	ER
HandlerExecutionChain	ERCT
Servlet/portlet Framework	ERTM
Proxy Bean Factory	ER
Support for ObjectXXXFailureException	ER
Alternate implementation of Property Editor	ER
Multipart Parameter Manager	ER
Supporting class for closing suppressing invocation handlers	ERCT
Simple service exporter	ER
Portlet/servlet bean	ERTM
Connection Factory Manager	ERCT
Connection Factory Getter	ERTM
Creating instances of container factory beans	ERTM
Managing cookies for mock responses	ERCM
Managing sets and lists	ERCM

ER = Extract Role, ERCT = Extract Role Changing Types, ERCM = Extract Role with Configurable Methods, ERTM = Extract Role with Types and Methods

Table 9.10: Spring resolved concerns (Part I)

Concern	Used Refactoring
Resolved With Purpose Built Roles	
JpaVendorAdapter	ERCT
AnnotationMethodHandlerAdapter	ER
Template Dao Support	ERTM
Custom Editor for maps and Collections	ERCM
Isolation level setter	ER
ProxyFactory holder	ERCT
Entry with strings	ER
Dealing with Configuration Property Values	ERTM
Test support	ERCT
PostProcessInstantiation tasks	ER
Looking up and logging Jndi templates	ERTM
Defining names to use within Metadata providers	ERCM
Validating handler for HandlerMappings	ERCT
Setting base names for resource bundles message source	ER
Session and request management for request Attributes	ERTM
Character encoding and content management for requests	ER
Single Connection Factory	ERTM
Session holder for mock requests	ERTM
Shutdown and destroy executors	ERCT
Manipulating outputstream for MockResponses	ERCT
AnnotationMethodHandlerExceptionResolver common code	ER
Web service feature conversion	ER
Standard ThreadPoolTask Executor	ERCT
Creating Rmi registry	ERCM
Hibernate AccessException conversion	ER
TaskScheduler	ER
Generating maxValue incrementer as auto incrementer	ER
Abstract handler exception resolver for Portlets/Servlets	ERCT
HttpInvokerRequestExecutor response validator and Gzip Response body	ERTM
Deciding parameter name to use in a database call metadata provider	ER
Model+View for Portlets/Servlets	ER
Setting concurrent limits	ERCM
Parameter manager for mockers	ER
Invoking methods in invocation handlers	ER
Convert comma delimited strings	ER
PortletServlet utils	ERCT
Binding information to threads	ERTM
Check parameters and headers	ERCT
Request parsing	ERCT
Resolved With Roles Placed in the Library	
Multimapper ER	ERTM

ER = Extract Role, ERCT = Extract Role Changing Types, ERCM = Extract Role with Configurable Methods, ERTM = Extract Role with Types and Methods

Table 9.10: Spring resolved concerns (Part II)

Unresolved Concern	Reason
Checking for SQL Script delimiters	Static and normal methods
Static Factories	Small inner static classes
Locating a constructor/method on a type	Uses anonymous classes

Table 9.11: Spring unresolved concerns

etc) the ERCT was much used. When some of the methods were also different the ERTM was used.

While servlet/portlet were responsible for the majority of ERCT and ERTM some similar relations were also present. Some beans had the same handling but different types, some transactions just used different types, service experts also, among others. These all contributed to a larger amount of the use of these refactorings.

The "Multimapper" concern deals with maps that can have various values with the same key. It was used in the HttpHeaders and LinkedMultiValueMap classes. The key and value types are represented by generics and the methods names needed no configuration as they had the same name in both classes. However to be able to add them to the library we need to let method names be configurable. We did not count this as a ERTM refactory but as an ER library because in the Spring framework all methods had the same name and the types were already generic.

The number of concerns with greater LOC count in the roles was 13 (18,5%). 4 of those have less than 15 LOC and while having few requirements it still it does not compensate for the required methods that need to be implemented. All the others have many requires, some have more than 10 requires, and they all had at least two required methods that needed to be implemented. Adding to this many also had configurations to be done and this also contributed to the higher LOC count. As an example, in the "Handler methods invoker" one class has 34 and the other 35 lines so the total is 69 LOC. The role has a total of 57 code lines, 8 of them due to the required methods. The role also had to store the types of some classes that the clone used as arguments for some methods (example: PortletRequest.class). The assignment and variables were responsible for 10 more lines of code in the role. The required methods that needed to be implemented in the player classes also contributed to a greater LOC in the use of roles.

9.5.4 Explaining Unresolved Concerns

Spring had only 3 unresolved concerns. They all were due to JavaStage limitations. The "Checking for SQL Script delimiters" concern had a clone with identical methods. The problem was that in the ResourceDatabasePopulator class this method was a normal method whereas in the JdbcTestUtils class this was a static method. The method to be

Concern	Original LOC	Roles LOC	Roles/ Original
Context resource for servlets/portlets	74	49	66%
Nested Exception	56	34	61%
Link for JPA Dialect	32	25	78%
Data binder for servlets/portlets	18	20	111%
Basic trigger behavior	55	43	78%
Managing methods in MBeans operations	32	26	81%
Adapters for requests	67	55	82%
Mock implementation for multipart requests	36	21	58%
Property accessor for traversing beans	16	15	94%
Abstract handler mapping	104	78	75%
Invocation Handler Adapter	39	27	69%
Metadata bean initializer	18	14	78%
Mapping exceptions class names to view names	98	62	63%
RMI Client Interceptor	110	65	59%
Web context scope for porlets/servlets	56	37	66%
Common multiparts resolver	78	56	72%
Resource Editor	20	17	85%
Dispatcher for servlets/portlets	292	218	75%
Trigger interceptors after some action	138	46	33%
Context aware processor	28	35	125%
Multimapper	62	2*	3%
Handler methods invoker	69	77	112%
Lob Creation Synchronization	20	16	80%
Session Bean Manager	26	32	123%
Mock config implementation	34	25	74%
Configurable application context	108	50	46%
Transaction Manager	90	69	77%
String Expression	86	57	66%
HandlerExecutionChain	82	74	90%
Servlet/portlet Framework	108	98	91%
Proxy Bean Factory	54	43	80%
Support for ObjectXXXFailureException	28	18	64%
Alternate implementation of Property Editor	16	18	113%
Multipart Parameter Manager	58	53	91%
Supporting class for closing suppressing invocation handlers	77	35	45%
Simple service exporter	28	31	111%
Portlet/servlet bean	50	46	92%
Connection Factory Manager	22	20	91%
Connection Factory Getter	16	14	88%
Creating instances of container factory beans	82	63	77%
Managing cookies for mock responses	24	19	79%
Managing sets and lists	104	38	37%

* used role from library

Table 9.12: Spring LOC count (Part I).

Concern	Original LOC	Roles LOC	Roles/ Original
JpaVendorAdapter	42	46	110%
AnnotationMethodHandlerAdapter	78	61	78%
Template Dao Support	30	22	73%
Custom Editor for maps and Collections	28	31	111%
Isolation level setter	22	21	95%
ProxyFactory holder	49	39	80%
Entry with strings	12	15	125%
Dealing with Configuration Property Values	30	26	87%
Test support	84	55	65%
PostProcessInstantiation tasks	22	17	77%
Looking up and logging Jndi templates	18	15	83%
Defining names to use within Metadata providers	75	28	37%
Validating handler for HandlerMappings	12	15	125%
Setting base names for resource bundles message source	24	17	71%
Session and request management for request Attributes	48	47	98%
Character encoding and content management for requests	40	27	68%
Single Connection Factory	64	53	83%
Session holder for mock requests	32	32	100%
Shutdown and destroy executors	22	32	145%
Manipulating outputstream for MockResponses	82	51	62%
AnnotationMethodHandlerExceptionResolver common code	142	75	53%
Web service feature conversion	28	18	64%
Standard ThreadPoolTask Executor	100	73	73%
Creating Rmi registry	97	66	68%
Hibernate AccessException conversion	12	13	108%
TaskScheduler	88	57	65%
Generating maxValue incrementer as auto incrementer	58	37	64%
Abstract handler exception resolver for Portlets/Servlets	55	32	58%
HttpInvokerRequestExecutor response validator and Gzip Response body	16	16	100%
Deciding parameter name to use in a database call metadata provider	14	15	107%
Model+View for Portlets/Servlets	92	51	55%
Setting concurrent limits	18	15	83%
Parameter manager for mockers	62	34	55%
Invoking methods in invocation handlers	18	16	89%
Convert comma delimited strings	30	20	67%
PortletServlet utils	144	87	60%
Binding information to threads	40	30	75%
Check parameters and headers	92	54	59%
Request parsing	232	160	69%

* used role from library

Table 9.12: Spring LOC count (Part II).

declared on the role must be either static or non-static, and the player class cannot change its nature. We could put the methods as static and both classes could play the same role. However, we did not know the intentions of the class developer, and if marking the methods as static would be an acceptable move.

The ”Static Factories” concern could not be resolved because the replicated code was inside static inner classes in the WebApplicationContextUtils and PortletApplicationContextUtils class. These inner classes represented factories for various objects like SessionObjects or WebRequestObjects. Since roles do not support static inner classes we could not develop such a role.

Finally, the ”Locating a constructor/method on a type” used replicated code inside an anonymous class. Currently JavaStage does not allow roles to be played by anonymous classes so we could not develop a role for this concern.

9.5.5 Other Considerations

One distinguishing feature of this case study is the presence of several blocks of code that seem equal but vary in the types, where the best example is the servlet-portlet support. This resulted in various clones using the ERTM and EMCT refactories.

The development of the Spring framework also proved mature and very few clones could be resolved by other refactorings. This may be because the spring framework is very mature but it can also be a consequence of the 20 tokens filtering we did. This filtering could have removed these and other types of clones. Since we would ignore these clones all the same we feel that the number of clone left after the filtering still provided a good case study.

Various roles required several accessor methods, because the framework relied on protected fields. One such example is the logger. It was common for superclasses to have a logger object, that was used by subclasses. The logger object was declared protected so all subclasses had access to it directly and no getLogger() method was used. Several roles required such a method so it had to be developed for the player classes. If the logger was declared private and the getLogger() method available in the superclass, many roles would have even less code associated with it. For the majority of fields, however, the framework used the good habit of accessing them using the getter-setter methods.

9.6 Discussion

We present a summary of the case results in table 9.13. The first observation we can make is that the presence of concerns with clones that could be resolved with traditional refactorings is low. The clones removed with Extract Role could also be removed using Extract Class or even Extract Superclass but roles were always a better solution (code length and/or modeling) so we opted for roles.

We can also observe that the number of unresolved concerns is low. The greater value belongs to JHotDraw with 8 (19.5%). But they can be easily explained and derives from the fact that we considered as clones some code that could be considered false clones. They are clones only in the structure and not on the code itself. A "Creating undo activity" concern creates an undo activity object for each of the various tools and commands supported by the framework. Each tool class has an UndoActivity inner class hence all the constructors of these inner classes have similar structures and were marked as clones. Each tool class also has a method that creates an undo activity. Since this method is somewhat similar, it was marked as a clone too. We opted to maintain this code as clones because they dealt with the same concern.

System	JHotDraw	Open JDK	Spring
Concerns			
Considered	38	31	84
Solved	30 (79%)	24 (77%)	81 (96%)
Unsolved	8 (21%)	7 (23%)	3 (4%)
Refactorings Used			
ER	18	13	33
ERCT	2	4	21
ERCM	2	2	7
ERCM	9	7	20
Code Statistics			
Original clone LOC	1390	1571	4763
Final clone LOC	883	986	3360
Size Reduction	36,5%	37,2%	29,5%
Roles with less LOC	27	20	66
Roles with more LOC	2	3	13
Roles with same LOC	1	1	2

ER = Extract Role, ERCT = Extract Role Changing Types, ERCM = Extract Role with Configurable Methods, ERTM = Extract Role with Types and Methods

Table 9.13: Clone removal results summary

Another example from JHotDraw is persistence: figures are streamed so they all have a write and read methods with similar structures, but not quite identical code. Our role DisplayBoxed considerably reduced this duplicated code, though. In another unresolved

concern the clone method overrides the superclass method for performance issues that we failed to understand. We think that if the method was deleted, the code would be equally effective.

Some clones (3 from JHotDraw and 4 from javac) could be removed with roles but their configuration would be complex and since clones had, at most, 4 simple lines of code, we decided that the clone was a better solution.

Interestingly the system with more concerns was the one with the least (only 3) unresolved concerns. These concerns could not be resolved because of use JavaStage limitations. One concern could not be resolved because the clone was a static method in one class and a regular method in another class. Roles do not support changing the nature of a method, so it was left unresolved. Javac also had such an unresolved clone. The other limitations include the use of roles inside an anonymous class - responsible for one concern in Spring - the use of roles in enums - responsible for one role in javac - and the use of static inner classes in roles - responsible for one concern in Spring.

The remaining javac unresolved clone was a declaration of static constants in different classes. The initialization was made in a static block so we could not put them into a role. Our roles also do not support public static variables.

We can observe that in Spring the use of Extract Roles Changing Types is proportionally greater than in the other systems. This is explained by the number of clones that dealt with Portlets and Servlets. Nearly every class that dealt with one concern had a corresponding class that dealt with the other concern, and most used Extract Role Changing Types or Extract Role with Types and Methods. This shows that in these situations roles are really useful.

From this discussion we can see that roles could remove almost all the clones and the ones it didn't resolve were some too small (less than 4 lines), some could be counted as not real clones, and some had rarely used particularities.

We can see that for the majority of the resolved concerns roles needed fewer lines of code, ranging from a 29,5% to 37% reduction in code size. This seems to indicate a smaller effort when developing a system with roles. The concerns that had more LOC than the original code were concerns with few lines of code where the role requirements and configuration overhead did not overcome the replicated code. LOC are a good measure of the effort that each approach requires but, as already mentioned, it does not account for the modularity and maintenance issues. Since roles can place code in a single place thus it becomes more modular as changes are confined to that place.

One thing that affects LOC count is the way the systems is programmed. If a role needed to access a player's field it would require a getter or a setter method. If the system makes use of getters and setter to access fields as recommended by good practices then

those required methods are already present. On the other hand if the system relies on public (like Open JDK) or protected fields (like Spring) then those getters/setters are needed. Since they were not present in the original code we counted them as a solution LOC and not as a clone LOC. This affects LOC count and roles suffer when comparing with the initial solution. This explains some roles greater LOC count, especially in the Spring framework where the protected fields were widely used by subclasses.

9.7 Threats to Validity

9.7.1 Complexity of JavaStage

This study did not take into account the difficulty in learning the JavaStage language. Nevertheless we believe that the few extensions that JavaStage introduces are simple to understand and do not pose great difficulties. The renaming mechanism with the # use is the feature that may raise more comprehension problems. In our opinion, however, its great usefulness makes for the extra complexity. To address this issue we must conduct experiments involving developers.

9.7.2 System Comprehension and Evolution

Using roles to remove code clones does not mean that our solution is easier to understand and to develop than using clones. We believe that all roles we developed contributed to a better system, some more than others. Nevertheless some studies must be made to assess if systems with roles are easier to maintain/evolve than without roles. For this we intend to analyze the newer versions of these target systems and analyze the impact roles have on issues like maintenance and evolution.

There is also the work of Riehle [Rie00] that shows that modeling a system with roles is easier and provides a better comprehension so we expect to find those advantages when they are used in programming as well.

9.7.3 Analyzed Systems

We analyzed 3 open source systems from different fields, each with its particularities. We believe that they make a good case study, but nevertheless results could be different if we used other systems and/or a bigger number of systems. The uniformity of results from these three very different systems is somewhat reassuring, though.

9.7.4 Case Study Setup

The clone detecting settings can also affect the detected clones and that would lead to different concerns. That and the removal of clones from the same file could have removed important clones. However, we would need to reduce the amount of clone sets to a manageable number, otherwise there would be a greater number of false clones. We even went under the limit of 30 tokens recommended in [KKI02] for the limitation of false clones.

9.7.5 Clone Detecting Technique Used

We used a token-based clone detection tool for detecting clones. A characteristic of such tools is that results tend to have too much noise (false clones), and that was evident from our study. The need to filter the results, like we did on the Spring framework, may lead to concerns being disregarded. If we used another clone detection technique we could have different clones and therefore different results. Nevertheless, we believe that the amount of clones detected in our study is enough to provide a good code base from which to draw conclusions. This technique also does not detect all the code associated with a concern so when analyzing the code we must not limit ourselves to the clone code but also to its surrounding and associated methods. This implies an additional effort, which we did in our case study.

9.8 Summary

In this chapter we presented a case study where we identified and removed clones from three open source systems. We presented the case study setup and briefly described each analyzed system. The case study results showed that roles are capable of significantly reducing the amount of clones in a system. Results also showed that role could reduce the amount of code one has to write. The case study also gave insights on roles limitations, but none seemed to be a major limitation.

Chapter 10

Conclusions

This dissertation focused on the issues behind the replication of code in a system, especially on those clones that are present due to lack of composition mechanisms by using roles as a class composition block.

Object-oriented decomposition is a good methodology for developing large scale systems. So much so that it is the most used decomposition in today's systems. There are several reasons for its success, from inheritance to an easily implemented information hiding. These characteristics allow object-oriented modules to be reused and, with inheritance, easily extended. However, in spite of these characteristics, we still encounter replicated code in OO systems. And replicated code that cannot be avoided by using OO techniques, or would be so difficult to avoid that using replicated code is a better solution. The presence of clones in a system may be an indicator that it was badly designed or implemented, but sometimes it just means that the design has been impaired by the lack of better ways of decomposing the system.

Not all clones have negative consequences for a system. There are clones present for performance reasons, especially in real time systems or in systems where the risk of using new algorithms is great, like financial applications. Apart from these specific reasons the majority of clones do have a negative impact on the system they appear on. They require a greater effort in maintaining and evolving a system. This means that for the majority of systems clones are to avoid and that a system without code clones is a better system than one with clones.

This dissertation intended to prove that the use of another decomposition technique could be used to remove code clones or at least, reduce code cloning. Since most systems

are built around the object-oriented paradigm we wanted to provide a new decomposition technique that was an extension of the object-oriented one. Because roles were a canonical extension of OO [CD05] and provided a way of decomposing a class into sets of functionality it provides to each client we opted to use roles as the decomposition strategy.

The evidence collected from the various case studies showed that roles are indeed capable of reducing the replicated code. Furthermore they showed that roles can be used to build reusable code giving developers another way of effectively reuse code. To support this claim a library of roles was developed.

10.1 Key contributions

Briefly, the main contributions of the work presented in this dissertation are:

- **Providing roles with a supporting language.** Despite the fact that several authors proposed roles as a mechanism useful in system design and modeling, no role supporting language existed. To bridge this gap a role language was proposed - JavaStage - and a suitable compiler developed. The language extended the Java language by supporting roles as class building blocks. The language syntax and its implementation made it backward compatible with Java specifications. This means that prior systems are compilable with the JavaStage compiler without any modifications. It also means that systems that use roles are executable with every existing JVM transparently.

- **Refactorings for removing code clones.** The removal of clones is the primary focus of this dissertation so explaining how to use roles to remove duplicated code is a necessary contribution. To reduce the replicated code the identification of the clone types that cannot be removed using the available refactorings was a first step. Then the author presented a collection of four refactorings, each one for dealing with one of the clone types identified. These were presented with real examples taken from the case studies target systems..

- **A role library.** To demonstrate the reusability of roles a role library was developed. This library started with the study of the design patterns present in the seminal book of patterns [GHJV95]. These patterns provided several roles for the library. The library grew further when adding roles that were used in the other case studies. In these case studies some developed roles turned out to be generic enough to make it to the library. Some of the library roles could even be reused in the case studies. The development of such a library also proves that roles provide a better modularization than just classes as the library roles could not be developed using classes alone.

- **Evaluation of the extent roles can reduce code clones in a system.** Insight on the impact that roles have on reducing code clones was ascertained by using a series of case studies. In these case studies clones were identified and grouped according to the concerns they dealt with and then removed using one of the refactorings proposed. The results of these case studies supported our claim that roles can reduce the amount of replicated code. Not only the replicated code was decreased but the line count of the system also decreased. The target systems in the case studies represented a wide range of applications. In each system the amount of clones reduced as well as the reduction of lines of code, which seems to indicate that the final system is better than the original one.

10.2 Future work

Scientific research is always a work in progress and new directions emerge constantly, whether to improve the present findings or to explore new possibilities. The research paths described in the next sections are deemed, by the author, worth of pursuit.

10.2.1 Improve and Enhance the JavaStage Compiler

The JavaStage compiler is a functional one but it is a line of command tool and it is not part of an integrated development environment (IDE) like Eclipse or NetBeans. The development of the compiler need to focus on the following items

- **Java 1.7 Support.** The compiler need to be upgraded to support the current version of the Java language, so it stays up to date with the new language features.
- **IDE integration.** The use of an IDE is a must on today's development environments. The JavaStage compiler can be used as a command line tool, and called from inside an IDE. This was done in the case studies where we used NetBeans as the development environment and customized it to use the JavaStage compiler. But this leaves out some key functionalities from IDE, like auto complete code. So the first goal is to develop a compiler for the Eclipse platform and the NetBeans platform. .
- **Debugger.** The debugging of the code is not supported in the compiler, as the debugging occurs in the compiled code that has no link to the role code. We need to improve that part to provide debugging to be done in the role code. This is important to detect bugs when developing a role and perform other features like profiling.

- **Expanding javadoc.** We need to expand javadoc so it incorporates the roles related information into the documentation. For example it needs to inform on the roles each class plays. For roles it should inform the required methods and the configurable methods it offers.
- **Compiler robustness.** The compiler needs to gain more robustness as sometimes the errors messages report to the synthesized role code instead of the source role code they originate. There are also some bugs that sometimes cause the compiler to abort compilation. Unfortunately this is a situation common to many systems.

10.2.2 Improve the JavaStage Language

Besides the improvements to the compiler, some improvements to the language may be proposed.

One possible improvement is in the requirements list as it tends to be repetitive, when a supplier appears several times. We intend to study ways to reduce that repetition. With IDE support some of these features could be automatized.

10.2.3 Refine and Extend the Refactorings

The refactoring proposed cover the removal of code clones. These proved useful in the case studies, but, as everything, they can be improved and replaced by newer and better refactorings.

In this dissertation we only developed clones related refactorings as it was the scope of the dissertation. However many refactorings can be proposed that are not related to clones. Some of the refactorings catalogs could be improved by introducing roles and new, role related, additions can be made. There is much work here to be done.

10.2.4 Extend the Role Library

We started the role library using design patterns as a starting point. We increased it when reducing the replicated code from the systems used in the case studies. Nevertheless this library can be increased further. This will be an ongoing task as the work on roles programming advances.

10.2.5 Further Studies

The performed experiment presented in chapter 9, provided supporting evidence for the benefits brought by the proposed contributions. Even so, further studies are required to solidify the results and consolidate the research.

Developers feedback

JavaStage proved to reduce the clone code, and roles proved to be a good way of composing classes. But the view of developers on the language and the ease or difficulties in the development of roles using the language is an important study. We designed JavaStage to have a practical and intuitive syntax but developers may find it otherwise. We need to conduct a series of studies to assess what developers think of the proposed extensions and how they perform using it to do a series of tasks.

Impact on system maintenance and evolution

Our studies showed that the replicated code was greatly reduced and the code was restricted to a single place. This means that to change the code, either to remove bugs or to add new features one needs only to change the code in that restricted place. With this we can expect the maintenance and evolution of the system to be improved. But we need to conduct such tests. We propose to analyze the newer versions of the target systems and identifying the evolutions made to the original code. We could then assess the impact that the developed roles would have on the evolutions made.

Using roles in programming

The advantages of using roles in the modeling and documentation process are already established. With JavaStage we can take those modeling principles into the implementation. But do they actually bring benefits to the code? To study this a series of case studies should be made. We can make two kinds of studies. We can make studies where we develop a system from the ground up using roles all the way in the development process and compare it to the same system developed without roles. This case study allows us to see the practical benefits of roles. We can also refactor an existing system to use roles and compare it to the original. This kind of study is less time consuming but its outcome may not be enough to assess the amount of work needed to develop a role system as we cannot compare development time.

Using roles in frameworks

The case studies we conducted included two frameworks, JHotDraw and Spring, and a systems with an API, the OpenJDK. We did not, however, study the impact that roles could have on the clients of such systems. We even set a rule of not changing the framework interface so that clients would not be affected. But using roles can bring benefits to framework design. A framework has hot spots that enable clients to extend the framework, for example, by subclassing a framework class. Providing roles to their

clients can be a useful way of extending a framework. The clients just need to play the role (or a set of roles) to take full advantage of the framework. This can open a new door in framework usage.

Industrial settings

Empirical studies should be performed in a professional, non-academic setting, with developers engaged in full-scale software projects with defined time frames and development process. These case-studies should engage in critical reflection of results with periodic interviews, questionnaires and focus groups over an extended period of time. Lessons should be learned that might be useful to help others and to act as agents of change in a real-life problem setting.

Thank you for reading.

Appendices

Appendix A

Publications

A.1 Papers

2013

- Fernando Barbosa and Ademar Aguiar, "Using Roles to Model Crosscutting Concerns", Proceedings of Aspect Oriented Software Development (Modularity aosd13), Fukuoka, Japan, 2013.
- Fernando Barbosa and Ademar Aguiar, "Composing Classes: Roles vs Traits", Proceedings of the 8th Evaluation of Novel Approaches to Software Engineering (ENASE2013), Angers, France, 2013
- Fernando Barbosa and Ademar Aguiar, "Removing Code Duplication with Roles", Proceedings of the 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT13), Budapest, Hungary, 2013.

2012

- Fernando Barbosa and Ademar Aguiar, "Roles as Modular Units of Composition", Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2012), Wroclaw, Poland, 2012.
- Fernando Barbosa and Ademar Aguiar, "Modeling and Programming with Roles: Introducing JavaStage", Proceedings of the 11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT12), Genoa, Italy, 2012.

- Fernando Barbosa and Ademar Aguiar, "Modeling Crosscutting Concerns with Roles", Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA 2012), Lisbon, Portugal, 2012.

2011

- Fernando Barbosa and Ademar Aguiar, "Reusable Roles, a test with Patterns", 18th Conference on Pattern Languages of Programs (PLoP11), Portland, Oregon, USA, 2011.

A.2 Book Chapters

- Fernando Barbosa and Ademar Aguiar, "Using Roles as Units of Composition", Evaluation of Novel Approaches to Software Engineering, Springer Berlin Heidelberg, 2013

A.3 Posters

- Fernando Barbosa and Ademar Aguiar, "Developing a role library with JavaStage", 2nd Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL'09), Genoa, Italy, 2009

Glossary

AOP Acronym for **A**spect **O**riented **P**rogramming [KLM+97].

API Acronym for **A**pplication **P**rogramming **I**nterface.

ER Acronym for **E**xtract **R**ole.

ERCM Acronym for **E**xtract **R**ole with **C**onfigurable **M**ethods.

ERCT Acronym for **E**xtract **R**ole **C**hanging **T**ypes.

ERMT Acronym for **E**xtract **R**ole with **T**ypes and **M**ethods.

FOSD Acronym for **F**eature-**O**riented **S**oftware **D**evelopment [AK09].

FOP Acronym for **F**eature-**O**riented **P**rogramming [AK09].

GUI Acronym for **G**raphical **U**ser **I**nterface.

HTML Acronym for **H**yper**T**ext **M**arkup **L**anguage.

IDE Acronym for **I**ntegrated **D**evelopment **E**nvironment.

J2EE Java Platform, Enterprise Edition. A platform for server programming in the Java programming language.

JVM Acronym for **J**ava **V**irtual **M**achine.

kLOC Acronym for **k**ilo **L**ines **O**f **C**ode — effectively thousands of LOC.

LOC Acronym for **L**ines **O**f **C**ode.

MVC Acronym for **M**odel-**V**iew-**C**ontroller .

OO Acronym for **O**bject-**O**riented.

SDK Acronym for **S**oftware **D**evelopment **K**it.

UML Acronym for **U**nified **M**odeling **L**anguage .

References

[AGO95] Antonio Albano, Giorgio Ghelli, and Renzo Orsini, *Fibonacci: a programming language for object databases*, The VLDB Journal **4** (1995), 403–444. Cited on pp. 49, 52, and 55.

[AK09] Sven Apel and Christian Kästner, *An overview of feature-oriented software development*, Journal of Object Technology **8** (2009), no. 5, 49–84. Cited on pp. 23, 43, and 207.

[BA11] Fernando Sergio Barbosa and Ademar Aguiar, *Reusable roles, a test with patterns*, In proceeding of the 18th Conference on Pattern Languages of Programs (PLoP11), 2011. Cited on p. 136.

[BA13a] _____, *Composing classes: Roles vs traits*, In Proceedings of the 8th Evaluation of Novel Approaches to Software Engineering (ENASE2013), July 2013. Cited on p. 101.

[BA13b] _____, *Removing code duplication with roles*, Proceedings of the 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT13), September 2013. Cited on p. 111.

[BA13c] _____, *Using roles to model crosscutting concerns*, Proceedings of the Aspect Oriented Software Development (Modularity aosd13), March 2013. Cited on p. 83.

[Bac80] Charles W. Bachman, *The role data model approach to data structures.*, ICOD'80, 1980, pp. 1–18. Cited on p. 48.

[Bak92] Brenda S. Baker, *A program for identifying duplicated code*, Computer Science and Statistics: Proc. Symp. on the Interface, March 1992, pp. 49–57. Cited on p. 11.

[Bak95] B. S. Baker, *On finding duplication and near-duplication in large software systems*, Proceedings of the Second Working Conference on Reverse Engineering (Washington, DC, USA), WCRE '95, IEEE Computer Society, 1995, pp. 86–. Cited on pp. 2, 11, 12, and 19.

[Bar08] Fernando Sergio Barbosa, *Comparing three aspect mining techniques*, Simpósio Doutoral em Engenharia Informática (DSIE'08) (Porto), February 2008. Cited on p. 154.

[BB02] Elizabeth Burd and John Bailey, *Evaluating clone detection tools for use during preventative maintenance*, Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (Washington, DC, USA), SCAM '02, IEEE Computer Society, 2002, pp. 36–. Cited on p. 16.

[BC90] Gilad Bracha and William Cook, *Mixin-based inheritance*, Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (New York, NY, USA), OOPSLA/ECOOP '90, ACM, 1990, pp. 303–311. Cited on p. 23.

[BD77] Charles W. Bachman and Manilal Daya, *The role concept in data models*, Proceedings of the third international conference on Very large data bases - Volume 3, VLDB '1977, VLDB Endowment, 1977, pp. 464–476. Cited on p. 48.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz, *Classbox/j: controlling the scope of change in java*, Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '05, ACM, 2005, pp. 177–189. Cited on pp. 45 and 109.

[BMD+99] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis, *Measuring clone based reengineering opportunities*, Proceedings of the 6th International Symposium on Software Metrics (Washington, DC, USA), METRICS '99, IEEE Computer Society, 1999, pp. 292–. Cited on pp. 16 and 21.

[BMD+00] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis, *Advanced clone-analysis to support object-oriented system refactoring*, Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00) (Washington, DC, USA), WCRE '00, IEEE Computer Society, 2000, pp. 98–. Cited on pp. 21 and 67.

[Boo95] Grady Booch, *Object-oriented analysis and design with applications (2. ed.)*, Benjamin/Cummings series in object-oriented software engineering, Addison-Wesley, 1995. Cited on p. 50.

[BSI07] Matteo Baldoni, Università Studi, and Torino Italy, *Interaction between objects in powerjava*, Journal of Object Technology **6** (2007), 7–12. Cited on pp. 49, 78, and 109.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer, *Scaling step-wise refinement*, IEEE Trans. Softw. Eng. **30** (2004), no. 6, 355–371. Cited on pp. 43 and 104.

[BvDvET05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe, *On the use of clone detection for identifying crosscutting concern code*, IEEE Trans. Softw. Eng. **31** (2005), no. 10, 804–818. Cited on p. 37.

[BW00] Martin Büchi and Wolfgang Weck, *Generic wrappers*, Proceedings of the 14th European Conference on Object-Oriented Programming (London, UK), ECOOP '00, Springer-Verlag, 2000, pp. 201–225. Cited on p. 61.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier, *Clone detection using abstract syntax trees*, Proceedings of the International Conference on Software Maintenance (Washington, DC, USA), ICSM '98, IEEE Computer Society, 1998, pp. 368–. Cited on pp. 2, 11, 15, 16, and 19.

[CBDM09] Tom Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang Meuter, *Adding state and visibility control to traits using lexical nesting*, Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Berlin, Heidelberg), Genoa, Springer-Verlag, 2009, pp. 220–243. Cited on p. 101.

[CD05] Daniel Chernuchin and Gisbert Dittrich, *Role types and their dependencies as components of natural types*, Roles, An Interdisciplinary Perspective (University of Dortmund), American Association for Artificial Intelligence, The AAAI Press, 2005. Cited on pp. 107 and 198.

[CH07] Andy Chiu and David Hirtle, *Beyond clone detection: Cs846 course project*, Tech. report, Cheriton School of Computer Science, University of Waterloo, 2007. Cited on p. 23.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein, *Multijava: modular open classes and symmetric multiple dispatch for java*, Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '00, ACM, 2000, pp. 130–145. Cited on pp. 45 and 107.

[CLD05] Daniel Chernuchin, Oliver S. Lazar, and Gisbert Dittrich, *Comparison of object-oriented approaches for roles in programming languages*, Roles, An Interdisciplinary Perspective (University of Dortmund), American Association for Artificial Intelligence, The AAAI Press, 2005. Cited on pp. 3 and 107.

[CMLC06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers, *Multijava: Design rationale, compiler implementation, and applications*, ACM Trans. Program. Lang. Syst. **28** (2006), no. 3, 517–575. Cited on pp. 45 and 107.

[CMM+05] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe, *A qualitative comparison of three aspect mining techniques*, Proceedings of the 13th International Workshop on Program Comprehension (Washington, DC, USA), IWPC '05, IEEE Computer Society, 2005, pp. 13–22. Cited on p. 154.

[Cor03] J.R. Cordy, *Comprehending reality: Practical challenges to software maintenance automation*, Int'l Workshop on Program Comprehension, IEEE Computer Society Press, 2003, pp. 196–206. Cited on pp. 13 and 15.

[DBB+03] Premkumar Devanbu, Bob Balzer, Don Batory, Gregor Kiczales, John Launchbury, David Parnas, and Peri Tarr, *Modularity in the new millenium: a panel summary*, Proceedings of the 25th International Conference on Software Engineering (Washington, DC, USA), ICSE '03, IEEE Computer Society, 2003, pp. 723–724. Cited on p. 44.

[DER07] Ekwa Duala-Ekoko and Martin P. Robillard, *Tracking code clones in evolving software*, Proceedings of the 29th international conference on Software Engineering (Washington, DC, USA), ICSE '07, IEEE Computer Society, 2007, pp. 158–167. Cited on p. 23.

[Deu05] Arie Van Deursen, *Ajhotdraw: A showcase for refactoring to aspects*, In: Workshop on Linking Aspect Technology and Evolution. (2005, 2005. Cited on p. 154.

[DNSB06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, and Andrew P. Black, *Traits: A mechanism for fine-grained reuse*, Transactions on Programming Languages and Systems **28** (2006), 331–388. Cited on pp. 23 and 41.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer, *A language independent approach for detecting duplicated code*, Proceedings of the IEEE International Conference on Software Maintenance (Washington, DC, USA), ICSM '99, IEEE Computer Society, 1999, pp. 109–. Cited on pp. 11 and 19.

[DWBN07] Stéphane Ducasse, Roel Wuyts, Alexandre Bergel, and Oscar Nierstrasz, *User-changeable visibility: resolving unanticipated name clashes in traits*, Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (New York, NY, USA), OOPSLA '07, ACM, 2007, pp. 171–190. Cited on p. 103.

[EFM09] William S. Evans, Christopher W. Fraser, and Fei Ma, *Clone detection via structural abstraction*, Software Quality Control **17** (2009), no. 4, 309–330. Cited on p. 19.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook, *A virtual class calculus*, Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '06, ACM, 2006, pp. 270–282. Cited on pp. 46 and 109.

[FF98] Matthew Flatt and Matthias Felleisen, *Units: cool modules for hot languages*, Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (New York, NY, USA), PLDI '98, ACM, 1998, pp. 236–248. Cited on pp. 45 and 106.

[FF00] Robert E. Filman and Daniel P. Friedman, *Aspect-oriented programming is quantification and obliviousness*, Tech. report, 2000. Cited on pp. 37, 40, and 103.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst. **9** (1987), no. 3, 319–349. Cited on p. 20.

[Fow99] Martin Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley, Boston, MA, USA, 1999. Cited on pp. 12 and 21.

[FR99] Richard Fanta and Václav Rajlich, *Removing clones from the code*, Journal of Software Maintenance **11** (1999), no. 4, 223–243. Cited on pp. 11, 12, 21, and 67.

[GB02] Kasper B. Graversen and Johannes Beyer, *Chameleon*, August 2002, Masters thesis. IT-University of Copenhagen. Cited on p. 49.

[GFGP06] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger, *Relation of code clones and change couplings*, Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering (Berlin, Heidelberg), FASE'06, Springer-Verlag, 2006, pp. 411–425. Cited on p. 15.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995. Cited on pp. 2, 6, 14, 31, 34, 38, 45, 57, 58, 75, 79, 135, and 198.

[Gie07] Simon Giesecke, *Generic modelling of code clones*, Duplication, Redundancy, and Similarity in Software (Dagstuhl, Germany) (Rainer Koschke, Ettore Merlo, and Andrew Walenstein, eds.), Dagstuhl Seminar Proceedings, no. 06301, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. Cited on p. 15.

[Gra06] Kasper Bilsted Graversen, *The nature of roles—a taxonomic analysis of roles as a language construct*, Ph.D. thesis, IT University of Copenhagen, Denmark, 2006. Cited on pp. 23, 47, 48, 55, and 62.

[GSS+06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan, *Modular software design with crosscutting interfaces*, IEEE Softw. **23** (2006), no. 1, 51–60. Cited on pp. 37 and 103.

[Gua92] Nicola Guarino, *Concepts, attributes and arbitrary relations: some linguistic and ontological criteria for structuring knowledge bases*, Data Knowl. Eng. **8** (1992), no. 3, 249–261. Cited on p. 48.

[Her05] Stephan Herrmann, *Programming with roles in objectteams/java*, 2005. Cited on pp. 49, 55, 59, 78, and 108.

[HK02] Jan Hannemann and Gregor Kiczales, *Design pattern implementation in java and aspectj*, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '02, ACM, 2002, pp. 161–173. Cited on pp. 38, 104, and 135.

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue, *A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system*, J. Softw. Maint. Evol. **20** (2008), no. 6, 435–461. Cited on pp. 21 and 67.

[HKK+04] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, and Key Words, *Aries: Refactoring support environment based on code clone analysis*, In The 8th IASTED International Conference on Software Engineering and Applications(SEA 2004, ACTA Press, 2004, pp. 222–229. Cited on p. 12.

[HKKI04] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, *Refactoring support based on code clone analysis*, Kansai Science City, Springer, 2004, pp. 220–233. Cited on pp. 11, 12, 21, and 67.

[HO93] William Harrison and Harold Ossher, *Subject-oriented programming: a critique of pure objects*, Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), OOPSLA '93, ACM, 1993, pp. 411–428. Cited on p. 44.

[HSHK10] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto, *Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software*, Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) (New York, NY, USA), IWPSE-EVOL '10, ACM, 2010, pp. 73–82. Cited on p. 15.

[IE11] Matthias Diehn Ingesman and Erik Ernst, *Lifted java: a minimal calculus for translation polymorphism*, Proceedings of the 49th international conference on Objects, models, components, patterns (Berlin, Heidelberg), TOOLS'11, Springer-Verlag, 2011, pp. 179–193. Cited on p. 80.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner, *Do code clones matter?*, Proceedings of the 31st International Conference on Software Engineering (Washington, DC, USA), ICSE '09, IEEE Computer Society, 2009, pp. 485–495. Cited on pp. 2, 11, and 15.

[JH06] N Juillerat and B Hirsbrunner, *An algorithm for detecting and removing clones in java code*, pp. 63–74, 2006. Cited on pp. 11 and 12.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu, *Deckard: Scalable and accurate tree-based detection of code clones*, Proceedings of the 29th international conference on Software Engineering (Washington, DC, USA), ICSE '07, IEEE Computer Society, 2007, pp. 96–105. Cited on pp. 2 and 11.

[JO93] Ralph E. Johnson and William F. Opdyke, *Refactoring and aggregation*, Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software (London, UK, UK), Springer-Verlag, 1993, pp. 264–278. Cited on p. 29.

[Joh92] Ralph E. Johnson, *Documenting frameworks using patterns*, SIGPLAN Not. **27** (1992), no. 10, 63–76. Cited on p. 159.

[Joh93] J. Howard Johnson, *Identifying redundancy in source code using fingerprints*, Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1, CASCON '93, IBM Press, 1993, pp. 171–183. Cited on pp. 15 and 19.

[Joh94] ———, *Substring matching for clone detection and change tracking*, Proceedings of the International Conference on Software Maintenance, IEEE Press, 1994, pp. 120–126. Cited on p. 15.

[KAB07] Christian Kastner, Sven Apel, and Don Batory, *A case study implementing features using aspectj*, Proceedings of the 11th International Software Product Line Conference (Washington, DC, USA), SPLC '07, IEEE Computer Society, 2007, pp. 223–232. Cited on pp. 37 and 103.

[KBLN04] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin, *An ethnographic study of copy and paste programming practices in oopl*, Proceedings of the 2004 International Symposium on Empirical Software Engineering (Washington, DC, USA), ISESE '04, IEEE Computer Society, 2004, pp. 83–92. Cited on pp. 11 and 15.

[KdMM+96] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and Morris Bernstein, *Pattern matching for clone and concept detection.*, Autom. Softw. Eng. **3** (1996), no. 1/2, 77–108. Cited on pp. 11 and 15.

[KG04] Cory Kapser and Michael W. Godfrey, *Aiding comprehension of cloning through categorization*, Proceedings of the Principles of Software Evolution, 7th International Workshop (Washington, DC, USA), IWPSE '04, IEEE Computer Society, 2004, pp. 85–94. Cited on p. 16.

[KG06a] ———, *"cloning considered harmful" considered harmful*, Proceedings of the 13th Working Conference on Reverse Engineering (Washington, DC, USA), WCRE '06, IEEE Computer Society, 2006, pp. 19–28. Cited on p. 15.

[KG06b] Cory J. Kapser and Michael W. Godfrey, *Supporting the analysis of clones in software systems: Research articles*, J. Softw. Maint. Evol. **18** (2006), no. 2, 61–82. Cited on pp. 2, 11, and 120.

[KH00] Raghavan Komondoor and Susan Horwitz, *Semantics-preserving procedure extraction*, Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '00, ACM, 2000, pp. 155–169. Cited on pp. 11, 12, 21, and 67.

[KH01] ———, *Using slicing to identify duplication in source code*, Proceedings of the 8th International Symposium on Static Analysis (London, UK, UK), SAS '01, Springer-Verlag, 2001, pp. 40–56. Cited on p. 20.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of aspectj*, Springer-Verlag, 2001, pp. 327–353. Cited on pp. 23 and 103.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, *Ccfinder: a multilinguistic token-based code clone detection system for large scale source code*, IEEE Trans. Softw. Eng. **28** (2002), no. 7, 654–670. Cited on pp. 11, 16, 19, 76, 154, and 195.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin, *Aspect-oriented programming*, Springer-Verlag, 1997, pp. 220–242. Cited on pp. 37 and 207.

[KMPS09] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen, *Exploring the use of package templates for flexible re-use of collections of related classes*, Journal of Object Technology **8** (2009), no. 7, 59–85. Cited on p. 105.

[Kni96] Günter Kniesel, *Objects don't migrate! – perspectives on objects with roles*, Technical report IAI-TR-96-11, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, nov 1996. Cited on pp. 48 and 60.

[KO96] Bent Bruun Kristensen and Kasper Osterbye, *Roles: conceptual abstraction theory and practical language issues*, Theor. Pract. Object Syst. **2** (1996), 143–160. Cited on p. 60.

[Kon97] K. Kontogiannis, *Evaluation experiments on the detection of programming patterns using software metrics*, Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97) (Washington, DC, USA), WCRE '97, IEEE Computer Society, 1997, pp. 44–. Cited on pp. 11, 12, 16, and 20.

[KR98] Gerti Kappel and Werner Retschitzegger, *A comparison of role mechanisms in object-oriented modeling*, In Proc. of the GI-Workshop Modellierung'98, 1998, pp. 105–109. Cited on p. 48.

[Kri95] Bent Bruun Kristensen, *Object-oriented modeling with roles*, Proceedings of the 2nd International Conference on Object-Oriented Information Systems, Springer-Verlag, 1995, pp. 57–71. Cited on pp. 23, 51, and 96.

[Kri96] Bent Bruun Kristensen, *Architectural abstractions and language mechanisms*, Proceedings of the Third Asia-Pacific Software Engineering Conference (Washington, DC, USA), APSEC '96, IEEE Computer Society, 1996, pp. 288–. Cited on pp. 56 and 96.

[Kri01] Jens Krinke, *Identifying similar code with program dependence graphs*, Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01) (Washington, DC, USA), WCRE '01, IEEE Computer Society, 2001, pp. 301–. Cited on pp. 12 and 20.

[KS04] Christian Koppen and Maximilian Störzer, *PCDiff: Attacking the fragile pointcut problem*, European Interactive Workshop on Aspects in Software (EIWAS) (Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, eds.), September 2004. Cited on pp. 37 and 103.

[KS08] Hannes Kegel and Friedrich Steimann, *Systematically refactoring inheritance to delegation in java*, Proceedings of the 30th international conference on Software engineering (New York, NY, USA), ICSE '08, ACM, 2008, pp. 431–440. Cited on pp. xi, 29, and 31.

[KSN05] Miryung Kim, Vibha Sazawal, and David Notkin, *An empirical study of code clone genealogies*, 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005. Cited on pp. 12 and 15.

[LCHY06] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu, *Gplag: detection of software plagiarism by program dependence graph analysis*, Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (New York, NY, USA), KDD '06, ACM, 2006, pp. 872–881. Cited on p. 20.

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer, *A disciplined approach to aspect composition*, Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (New York, NY, USA), PEPM '06, ACM, 2006, pp. 68–77. Cited on p. 59.

[Lie86] Henry Lieberman, *Using prototypical objects to implement shared behavior in object-oriented systems*, Conference proceedings on Object-oriented programming systems, languages and applications (New York, NY, USA), OOPLSA '86, ACM, 1986, pp. 214–223. Cited on p. 29.

[LLMZ06] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, *Cp-miner: Finding copy-paste and related bugs in large-scale software code*, IEEE Trans. Softw. Eng. **32** (2006), no. 3, 176–192. Cited on pp. 2, 11, 12, 15, 16, and 19.

[LLT11] Tadeusz Lasota, Tomasz Luczak, and Bogdan Trawiński, *Experimental comparison of resampling methods in a multi-agent system to assist with property valuation*, Proceedings of the 5th KES international conference on Agent and multi-agent systems: technologies and applications (Berlin, Heidelberg), KES-AMSTA'11, Springer-Verlag, 2011, pp. 342–352. Cited on p. 154.

[LPM+97] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl, *Assessing the benefits of incorporating function clone detection in a development process*, Proceedings of the International Conference on Software Maintenance (Washington, DC, USA), ICSM '97, IEEE Computer Society, 1997, pp. 314–. Cited on p. 23.

[MDMR09] Marius Marin, Arie Deursen, Leon Moonen, and Robin Rijst, *An integrated crosscutting concern migration strategy and its semi-automated application to jhotdraw*, Automated Software Engg. **16** (2009), no. 2, 323–356. Cited on p. 154.

[MF06] Miguel P. Monteiro and João M. Fernandes, *Transactions on aspect-oriented software development i*, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 214–258. Cited on p. 38.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh, *Jiazzi: new-age components for old-fasioned java*, Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '01, ACM, 2001, pp. 211–222. Cited on pp. 45 and 106.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore Merlo, *Experiment on the automatic detection of function clones in a software system using metrics*, Proceedings of the 1996 International Conference on Software Maintenance (Washington, DC, USA), ICSM '96, IEEE Computer Society, 1996, pp. 244–. Cited on pp. 2, 11, 15, 16, and 20.

[MNK+02] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto, *Software quality analysis by code clones in industrial legacy software*, Proceedings of the 8th International Symposium on Software Metrics (Washington, DC, USA), METRICS '02, IEEE Computer Society, 2002, pp. 87–. Cited on p. 15.

[MO03] Mira Mezini and Klaus Ostermann, *Conquering aspects with caesar*, Proceedings of the 2nd international conference on Aspect-oriented software development (New York, NY, USA), AOSD '03, ACM, 2003, pp. 90–99. Cited on pp. 45 and 106.

[MTB11] Anshuman Mukherjee, Zahir Tari, and Peter Bertok, *A spring based framework for verification of service composition*, Proceedings of the 2011 IEEE International Conference on Services Computing (Washington, DC, USA), SCC '11, IEEE Computer Society, 2011, pp. 258–265. Cited on p. 154.

[NS03] E Nickell and I Smith, *Extreme programming and software clones*, 2003. Cited on p. 23.

[Odb94] Erik Odberg, *Category classes: flexible classification and evolution in object-oriented databases*, Proceedings of the 6th international conference on Advanced information systems engineering (Secaucus, NJ, USA), CAiSE '94, Springer-Verlag New York, Inc., 1994, pp. 406–420. Cited on p. 96.

[OT00] H. Ossher and P. Tarr, *Multi-dimensional separation of concerns and the hyperspace approach*, Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer Academic Publishers, 2000. Cited on p. 44.

[Par72] D. L. Parnas, *On the criteria to be used in decomposing systems into modules*, Communications of the ACM **15** (1972), 1053–1058. Cited on pp. 1, 75, and 77.

[Per90] B. Pernici, *Objects with roles*, Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems (New York, NY, USA), COCS '90, ACM, 1990, pp. 205–215. Cited on p. 52.

[Prz11] Adam Przybylek, *Systems evolution and software reuse in object-oriented programming and aspect-oriented programming*, Proceedings of the 49th international conference on Objects, models, components, patterns (Berlin, Heidelberg), TOOLS'11, Springer-Verlag, 2011, pp. 163–178. Cited on pp. 37 and 103.

[QB04] Philip J. Quitslund and Andrew P. Black, *Java with traits - improving opportunities for reuse*, In The MASPEGHI Workshop at ECOOP, 2004. Cited on p. 42.

[RC07] Chanchal Kumar Roy and James R. Cordy, *A survey on software clone detection research*, SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY **115** (2007). Cited on pp. xv, 2, 12, 13, 15, 16, 20, and 21.

[RD04] Filip Van Rysselberghe and Serge Demeyer, *Evaluating clone detection techniques from a refactoring perspective*, Proceedings of the 19th IEEE international conference on Automated software engineering (Washington, DC, USA), ASE '04, IEEE Computer Society, 2004, pp. 336–339. Cited on pp. 11 and 12.

[RG98] Dirk Riehle and Thomas Gross, *Role model based framework design and integration*, Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '98, ACM, 1998, pp. 117–133. Cited on pp. 3, 68, 74, and 104.

[Rie00] Dirk Riehle, *Framework design: A role modeling approach*, Ph.D. thesis, ETH Zürich, Zürich, Switzerland, 2000. Cited on pp. 23, 50, 52, 53, 68, 74, 76, 78, 156, and 194.

[RJ07] Ekaterina Razina and David Janzen, *Effects of dependency injection on maintainability*, Proceedings of the 11th IASTED International Conference on Software Engineering and Applications (Anaheim, CA, USA), SEA '07, ACTA Press, 2007, pp. 7–12. Cited on p. 154.

[RWL96] Trygve Reenskaug, Per Wold, and Odd Arild Lehne, *Working with objects - the ooram software engineering method*, Manning, 1996. Cited on pp. 3, 50, 68, and 76.

[SB02] Yannis Smaragdakis and Don Batory, *Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs*, ACM Trans. Softw. Eng. Methodol. **11** (2002), no. 2, 215–255. Cited on pp. 44 and 105.

[SD05] Charles Smith and Sophia Drossopoulou, *Chai: traits for java-like languages*, Proceedings of the 19th European conference on Object-Oriented Programming (Berlin, Heidelberg), ECOOP'05, Springer-Verlag, 2005, pp. 453–478. Cited on pp. xi, 42, 43, and 100.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black, *Traits: Composable units of behaviour*, Lecture Notes in Computer Science **2743** (2003), 248–274. Cited on pp. 23 and 41.

[SG99] Benedikt Schulz and Thomas Genssler, *Transforming inheritance into composition.*, EuroPLoP (Paul Dyson and Martine Devos, eds.), UVK - Universitaetsverlag Konstanz, 1999, pp. 93–102. Cited on p. 29.

[Sow84] J.F. Sowa, *Conceptual structures: Information processing in mind and machine*, Addison-Wesley, 1984. Cited on p. 48.

[SSSM95] Pedro Sousa, António Rito Silva, Ant'onio Rito Silva, and José Alves Marques, *Object identifiers and identity: a naming issue*, In International Workshop on Object Orientation in, IEEE Press, 1995. Cited on p. 96.

[Ste00] Friedrich Steimann, *On the representation of roles in object-oriented and conceptual modelling*, Data Knowl. Eng. **35** (2000), 83–106. Cited on pp. 3, 47, 48, and 50.

[Ste01] ———, *Role = interface: a merger of concepts*, Journal of ObjectOriented Programming **14** (2001), no. 4, 23–32. Cited on p. 96.

[Ste06] ———, *The paradoxical success of aspect-oriented programming*, Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), OOPSLA '06, ACM, 2006, pp. 481–497. Cited on pp. 37, 40, and 103.

[SZ89] Lynn A Stein and Stanley B. Zdonik, *Clovers: The dynamic behavior of types and instances*, Tech. report, Providence, RI, USA, 1989. Cited on p. 49.

[TBG04] M. Toomim, A. Begel, and S.L. Graham, *Managing duplicated code with linked editing*, Proc. IEEE Symp. Visual Languages: Human Centric Computing, IEEE Press, 2004, pp. 173–180. Cited on pp. 13 and 23.

[THP92] Walter F. Tichy, Nico Habermann, and Lutz Prechelt, *Summary of the dagstuhl workshop on future directions in software engineering: February 17-21, 1992*, SIGSOFT Softw. Eng. Notes **18** (1992), no. 1, 35–48. Cited on p. 4.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr., *N degrees of separation: multi-dimensional separation of concerns*, Proceedings of the 21st international conference on Software engineering (New York, NY, USA), ICSE '99, ACM, 1999, pp. 107–119. Cited on p. 44.

[Tru04] Eddy Truyen, *Dynamic and context-sensitive composition in distributed systems*, Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, November 2004. Cited on p. 96.

[TUI07] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama, *Objects as actors assuming roles in the environment*, Software Engineering for Multi-Agent Systems V (Ricardo Choren, Alessandro Garcia, Holger Giese, Ho-Fung Leung, Carlos Lucena, and Alexander Romanovsky, eds.), Springer-Verlag, Berlin, Heidelberg, 2007, pp. 185–203. Cited on pp. 79 and 108.

[VN96] Michael VanHilst and David Notkin, *Using role components in implement collaboration-based designs*, Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '96, ACM, 1996, pp. 359–369. Cited on p. 107.

[WCL97] Raymond K. Wong, H. Lewis Chau, and Frederick H. Lochovsky, *A data model and semantics of objects with dynamic roles*, Proceedings of the Thirteenth International Conference on Data Engineering (Washington, DC, USA), ICDE '97, IEEE Computer Society, 1997, pp. 402–411. Cited on p. 49.

[WGM89] André Weinand, Erich Gamma, and Rudolph Marty, *Design and implementation of et++, a seamless object-oriented application framework*, Structured Programming **10** (1989), no. 2, 63–87. Cited on p. 159.

[WSGF04] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer, *Clone detection in source code by frequent itemset techniques*, Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop (Washington, DC, USA), SCAM '04, IEEE Computer Society, 2004, pp. 128–135. Cited on p. 19.

[Yan91] Wuu Yang, *Identifying syntactic differences between two programs*, Softw. Pract. Exper. **21** (1991), no. 7, 739–755. Cited on p. 19.

[ZW98] Marvin V. Zelkowitz and Dolores R. Wallace, *Experimental models for validating technology*, Computer **31** (1998), no. 5, 23–31. Cited on pp. 4 and 5.