Run-time Variability with Roles

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doktor-Ingenieur (Dr.-Ing.)

at
Faculty of Computer Science
Technische Universität Dresden

by

M.Sc. Nguonly Taing

Born on February 10, 1982 in Phnom Penh, Cambodia

Supervisors:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Prof. Dr.-Ing. Thomas Schlegel
External Reviewer:
Prof. Dr. Kim Mens (Université catholique de Louvain)

Date of Submission: November 17, 2017
Date of Defense: April 4, 2018

Acknowledgements

The root of my doctoral study started with a chitchat between Prof. Alexander Schill, Mr. Des Phal and me in a conference in December 2012 in Cambodia. During the talk, we were finding all possible scholarships for my study. Unfortunately, there was no funding available for degree-seeking applicants. One year later, I contacted Prof. Schill for his supervision letter in order to apply for a scholarship from the Erasmus Mundus Program. Luckily, I was selected, and my Ph.D. journey began in October 2014. My most profound gratitude goes to him for his patience and continuous support throughout these years. I am impressed with his blazingly fast reaction. Whenever I had some issues demanding his advice, he responded promptly by email and was available for discussion in person if needed. I am fortunate to have him as my supervisor.

Finishing this dissertation would not be possible without the constant support from my advisor, Dr. Thomas Springer. Thomas spent restless hours for day-to-day advisory and contributed significantly to writing various papers. I remember we were once staying awake over the midnight to edit the paper altogether although we had to travel to a conference in the early morning. Thanks for your understanding, advice, and friendship.

Having worked on roles, I claim that being individual is important, but collaboration is even more crucial. Basically, I worked alone but without collaboration with others, I would have gotten lost my way and never perceived this topic in such a fine-grained level. I express my appreciation to the investigators of the RoSI project for allowing me to take part. Several seminars, workshops, and discussions made with the RoSI team shaped my view on roles. Among these colleagues, Markus Wutzler and Martin Weißbach were the closest ones not only in RoSI but also in the chair of computer networks. Regularly, I approached them whenever I had problems either academically or personally. Thanks for your contribution and friendship.

My gratitude extends to Prof. Nicolás Cardozo who constantly collaborated with me. He brought me a deep level of COP understanding and shared me the tranquility concept which later became a key contribution to this work. Through him, I met Prof. Kim Mens whom I am appreciated for his kindness to review this dissertation.

The quality of the dissertation could not be in this shape without a great help from Ivonne Wittig. She patiently proofread the dissertation back and forth. Besides Prof. Schill, I believe she is the only one who read my dissertation every written word. Well, I was wrong. Kim proved it, at least. He made exceptionally detailed comments on both the contents and the linguistic issues that I cannot overlook his remarkable contribution. Thanks for your great help.

I am indebted to the Swap and Transfer project of the Erasmus Mundus Program that financed this research for 33 months and offered great administrative support. Besides, I would like to thank the Graduate Academy of the TU Dresden for

granting me the four-month scholarship to complete the last three chapters.

Finally, I am very grateful to my wife, Ely, who took care of my daughters, Chumpounut and Ampuneat, during my absence and greatly supported me although I was more than 10,000 km away. I owe them sincere gratitude for their unconditional love, encouragement, and understanding. I dedicate this dissertation to them. I do love you all.

Abstract

Adaptability is an intrinsic property of software systems that require adaptation to cope with dynamically changing environments. Achieving adaptability is challenging. Variability is a key solution as it enables a software system to change its behavior which corresponds to a specific need. The abstraction of variability is to manage variants, which are dynamic parts to be composed to the base system. Run-time variability realizes these variant compositions dynamically at run time to enable adaptation. Adaptation, relying on variants specified at build time, is called anticipated adaptation, which allows the system behavior to change with respect to a set of predefined execution environments. This implies the inability to solve practical problems in which the execution environment is not completely fixed and often unknown until run time. Enabling unanticipated adaptation, which allows variants to be dynamically added at run time, alleviates this inability, but it holds several implications yielding system instability such as inconsistency and run-time failures. Adaptation should be performed only when a system reaches a consistent state to avoid inconsistency. Inconsistency is an effect of adaptation happening when the system changes the state and behavior while a series of methods is still invoking. A software bug is another source of system instability. It often appears in a variant composition and is brought to the system during adaptation. The problem is even more critical for unanticipated adaptation as the system has no prior knowledge of the new variants.

This dissertation aims to achieve anticipated and unanticipated adaptation. In achieving adaptation, the issues of inconsistency and software failures, which may happen as a consequence of run-time adaptation, are evidently addressed as well. Roles encapsulate dynamic behavior used to adapt players representing the base system, which is the rationale to select roles as the software system's variants. Based on the role concept, this dissertation presents three mechanisms to comprehensively address adaptation. First, a dynamic instance binding mechanism is proposed to loosely bind players and roles. Dynamic binding of roles enables anticipated and unanticipated adaptation. Second, an object-level tranquility mechanism is proposed to avoid inconsistency by allowing a player object to adapt only when its consistent state is reached. Last, a rollback recovery mechanism is proposed as a proactive mechanism to embrace and handle failures resulting from a defective composition of variants. A checkpoint of a system configuration is created before adaptation. If a specialized bug sensor detects a failure, the system rolls back to the most recent checkpoint. These mechanisms are integrated into a role-based runtime, called LyRT.

LyRT was validated with three case studies to demonstrate the practical feasibility. This validation showed that LyRT is more advanced than the existing variability approaches with respect to adaptation due to its consistency control and failure handling. Besides, several benchmarks were set up to quantify the overhead of LyRT concerning the execution time of adaptation. The results revealed that the overhead introduced to achieve anticipated and unanticipated adaptation to be small enough for practical use in adaptive software systems. Thus, LyRT is suitable for adaptive software systems that frequently require the adaptation of large sets of objects.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
COP	Context-Oriented Programming
CPU	Central Processing Unit
CROI	Compartment Role Object Instance
CROM	Compartment Role Object Model
DLL	Dynamic Link Library
DSL	Domain-Specific Language
DSPL	Dynamic Software Product Line
DSU	Dynamic Software Updates
FOP	Feature-Oriented Programming
GUI	Graphical User Interface
IDE	Integrated Development Environment
JIT	Just-In-Time
JMH	Java Microbenchmarking Harness
JPDA	Java Platform Debugging Architecture
JVM	Java Virtual Machine
JVMAI	Java Virtual Machine Aspect Interface
MOP	Metaobject Protocol
MP	Meta-Programming
OOP	Object-Oriented Programming
ROP	Role-Oriented Programming
SAS	Self-Adaptive System
SOA	Service-Oriented Architecture
SOP	Subject-Oriented Programming
SPL	Software Product Line
STM	Software Transactional Memory
TCP	Transmission Control Protocol
XML	Extensible Markup Language

Chapter 1

Introduction

Software systems in certain application domains have to cope with heterogeneous and dynamically changing environments such as cyber-physical systems, systems for smart cities, and the Internet of Things. Adaptability is a desired characteristic of such systems as it allows the program behavior to adapt in response to the changes of execution environments. Building a software system with adaptability is challenging. Traditionally, multiple `if` statements were employed, but this technique often ends up in tangled code which is hardly reusable. In principle, a good software practice should conform to the idea of modularization [Par72] and separation of concerns [Dij76], which allow developers to have clean and modular application code which is easily reusable and extensible. Adaptive software systems are no exception and should be designed with these perspectives.

Variability is a key solution to reusability and adaptability. Variability is the ability of a software system to change its behavior corresponding to a specific need [GWT+14]. Although variability was primarily studied in the area of Software Product Lines (SPLs) [CB10], it has been recently applied to many other types of today's software systems, e.g., adaptive software systems [Hil10, GWT+14]. The abstraction of variability is to manage both base systems, also known as static parts, and their *variants*, which are dynamic parts to be composed into the base systems via *variation points*. While variants are a small and reusable software feature, variation points are places to which those variants are composed. A composition technique then realizes these composed variants and variation points to create different behavior which features adaptability.

We[1] separate different types of variability based on the time to which those variants are composed. While variability focuses on the composition of variants at development or build time, *run-time variability* shifts the composition of those variants from development to run time[2]. Therefore, run-time variability allows the base system to dynamically adapt its behavior to different execution environments based on the composed variants. We consider this adaptation *anticipated adaptation* since the composed variants are given beforehand during the development time but dynamically composed to the base system at run time.

Furthermore, today's software systems often require evolving to adjust the changing requirements over an extended period of time. Running systems, such as a flight control application, cannot tolerate a system restart to address the new requirements which are unknown in

[1] In this dissertation, the term "We" refers to "the author".

[2] "Run time" is the time at which the application runs, e.g., the composition happens at run time. "Run-time" is used as a compound adjective, e.g., run-time adaptation. "Runtime" is an environment where the core mechanisms of a language are executed, e.g., Java's runtime.

advance. Therefore, those systems need the ability to incorporate those new requirements while running. This problem is normally addressed by Dynamic Software Updates (DSU), a research domain theoretically permitting almost no limit to update the running program. However, DSU is a low-level solution integrated tightly into the programming language runtime or the operating systems so that it does not consider any implementation patterns of incorporating of those new requirements in a clean and an extensible way. Developers have to tackle this problem on their own. Since DSU can change any part of the program on-the-fly, this can be perceived as *unanticipated adaptation* because the changes were unexpected. Although aiming for unanticipated adaptation is one of our goals in order to allow unforeseen change at run time, this kind of unanticipated adaptation is not desirable for two reasons. On the one hand, the way to achieve this unanticipated adaptation, as already mentioned, is not clean and modular. Obviously, depending on the implementation pattern, updating program code directly for several iterations may lead to tangled code. On the other hand, we cannot integrate the anticipated adaptation easily because DSU is not purposefully designed for adaptation.

We envision an adaptive system which is built with run-time variability in mind. The base systems are cleanly separated from the variants; the dynamic composition of those given variants enables anticipated adaptation. Run-time variability should not only address anticipated adaptation but also covers unanticipated adaptation to improve the high availability of the running systems. Adopting the principle of run-time variability, *unanticipated adaptation* is a behavioral change of the base system resulting from a dynamic composition of new variants which are not given beforehand. In other words, unanticipated adaptation extends anticipated adaptation with a mechanism to integrate new variants to the system at run time, but its variants are unknown until they are needed. Without run-time variability, supporting both adaptations at run time is challenging, and the resulting code is not clean and often poor in modularization. From this point onwards, anticipated and unanticipated adaptation, if not mentioned explicitly, refer to run-time adaptation whose variants, either given beforehand or later, are dynamically composed to the base system, and the base system adapts its behavior concerning the composed variants accordingly.

Although changes resulting from adaptation fit a software specification, they may come with certain drawbacks. Technically, a program's state and behavior are changed as a result of either anticipated or unanticipated adaptation. The adverse effect of such changes is prone to *inconsistency* and *instability because of failures* which should be avoided. We consider these problems as a *consequence of adaptation*. Inconsistency of an adaptive software system happens when the base system, which contains multiple objects[3], changes its behavior promptly due to adaptation during a series of method invocations. The series of these method invocations have the purpose of solving a common task which requires consistent behavior of those engaging objects from the start to the end. Therefore, to maintain the same consistent behavior, adaptation occurring in between these invocations is prohibited.

Another possible consequence of adaptation is a system instability because of an execution failure. The failure may happen when the base system is executing, and its behavior is adapted several times with different compositions of multiple variants. Among those compositions, one may contain a bug, e.g., `DivideByZero`, which propagates to a failure at run time. Without a proper failure handling mechanism, the whole system could crash.

[3]Objects refer to instances derived from a given type of Object-Oriented Programming (OOP)

This dissertation presents a novel approach to run-time variability aiming at achieving anticipated and unanticipated adaptation in a single run-time solution. It explores the basic mechanisms for dynamic binding at instance level as well as advanced mechanisms for maintaining system consistency and stability related to adaptation.

1.1 Problem Statement

Run-time variability provides both reusability and adaptability. It is, therefore, the most appropriate concept to date to design an adaptive software system. Several approaches have been proposed for run-time variability ranging from architectural to programming language solutions. Recently, the support of run-time variability has been integrated into the language level. Through language-level abstractions, this integration minimizes greatly the effort of implementing a system which requires frequent adaptations in response to *context* changes. Context is any information which is computationally accessible and can be used to characterize the situation of an object [Dey01, HCN08]. Context-Oriented Programming (COP) [SGP12a] and Role-Oriented Programming (ROP) [KLG+14] are emerging programming paradigms which are proposed in this direction.

Both COP and ROP enhance standard programming techniques by providing language constructs to feature adaptation. However, these language solutions tackle mainly on anticipated adaptation while neglecting unanticipated adaptation. Supporting unanticipated adaptation is challenging, and it depends on the level of adaptation. From a run-time perspective, adaptation can be performed at either type or instance level. This dissertation argues that adaptation should be performed at instance level to avoid a long disruption in changing behavior and to minimize inconsistency.

Instance-level adaptation allows individual instances to adapt independently. Because of that, the adapted instances can change behavior promptly. In contrast, type-level adaptation changes the behavior of all derived instances at once. In the case that some adapted instances are busy in execution, type-level adaptation deters the adaptation until all the instances are idle [CH05, HCH08, AHHM11, DVCH07]. If all instances are forcefully adapted at once, the system may face inconsistency. Without a proper mechanism to avoid inconsistency, supporting unanticipated adaptation rather causes problems than the benefit. Inconsistency also happens in anticipated adaptation, but the probability of having it in unanticipated adaptation is very high due to the lack of knowledge of when the runtime reaches a *tranquil state*. A tranquil state is a state in which an object or instance is safe to update its behavior without facing inconsistency [VEBD07]. Therefore, a mechanism to find this tranquil state for adaptation is needed not only for unanticipated adaptation but also anticipated adaptation.

COP solutions, such as EventCJ [KAM11], ServalCJ [KAM15], ContextJS [LASH11], and Context Traits [GMCC13], provide instance-level adaptation which should be possible for incorporating unanticipated adaptation. Nonetheless, these solutions ignore this option. Similarly, in the domain of ROP, while SCROLL [LA15] supports instance-level adaptation by design, OT/J [Her05] uses an inflexible predicate to achieve a similar result. Both of them are not designed for unanticipated adaptation. Additionally, the inconsistency[4] is fully addressed neither by any COP nor ROP solutions.

[4] Another kind of inconsistency resulting from a conflict of multiple context dependencies is well-addressed by CoPN [Car13].

Besides facing inconsistency, the adapted system may also encounter execution failures resulting from software bugs which appear due to a dynamic composition of variants. In case of unanticipated adaptation, this problem is even more critical because variants are incorporated lately. Although COP and ROP provide language abstractions to easily develop adaptive software systems, they have not yet proposed a testing framework for such systems [SGP12a]. For a small set of variants, a traditional software testing technique can be applied by testing all possible compositions of those variants. However, this is hardly achievable for a large set of variants. Although bug-finding tools, e.g., FindBugs [HP04], help to catch bugs, they often produce inaccurate results [JSMHB13, RAF04], and none of them is designed to find bugs in adaptive software systems. Therefore, bugs are likely to exist and cause failures at run time. A proactive mechanism to embrace and handle such failures at run time is necessary to improve run-time stability.

In a nutshell, the core challenge is the lack of a dynamic instance binding mechanism to enable instance-level adaptation in which anticipated and unanticipated adaptation should coexist in a single run-time solution. Unanticipated adaptation needs to be incorporated from the ground up so that additional mechanisms to handle inconsistency and failures can be proposed in order to minimize the consequences of adaptation.

1.2 Objective

This dissertation is developed with an observation that state of the art is inadequate to address the adaptation comprehensively in a single run-time solution. The role concept [KLG+14] is utilized due to its dynamic and context-dependent property. Since roles and their playing property naturally target on the instance level, selecting roles as variants helps to achieve instance-level adaptation. This dissertation aims at:

1. **Supporting anticipated adaptation.** With a separation of the base system and its given variants, the system should be adapted cleanly and modularly as a result of different variant compositions. Adaptation is activated dynamically by a context change. The adaptation performs at instance level in order to minimize the long disruption and inconsistency. In addition, it is easier to extend the support of unanticipated adaptation.

2. **Supporting unanticipated adaptation.** Such support tackles unnecessary runtime restart owing to bug fixes and new requirements. Although we do not expect to adapt every piece of applications, we follow the principle of run-time variability in which changing the variants adapts the base system. Most of the existing variants should be replaceable with new variants at run time. Those new variants are unknown beforehand, but the system adapts accordingly once their composition is done.

3. **Handling consequences of adaptation.** Adaptation changes state and behavior of the system, so that it might cause some problems which must be avoided. The issues to overcome are described as follows:

   (a) **State and behavioral inconsistency.** The inconsistency of the system resulting from adaptation should be avoided. That is the case when adaptation happens during a series of ongoing method invocations which require consistent behavior. Then, adaptation which occurs between these invocations is prevented to avoid inconsistency.

(b) **Execution failure.** Bugs are among the leading causes of software failures. Adaptive software systems are no exception and even more vulnerable to failures. Due to dynamic composition of variants, it is notoriously difficult to eliminate all the bugs during the testing phase. Thus, bugs are likely to exist during run time to cause system failures. This problem, if possible, must be avoided to improve run-time stability.

Hence, the main research questions can be formulated as follows:

1. How can anticipated and unanticipated adaptation at instance level in the context of run-time variability be achieved and what is the trade-off to realize those achievements?
2. How can consistent behavior of objects engaging in ongoing method executions be maintained if dynamic changes take place?
3. How can run-time stability be improved in the presence of run-time failures caused by variant compositions?

1.3 Overview of the Approach

Following the principle of run-time variability, roles are selected as variants, and we propose a *dynamic instance binding mechanism* as a realized composition mechanism of roles. With this mechanism, we achieve anticipated and unanticipated adaptation. Inconsistency and failures are handled by an *object-level tranquility mechanism* and a *rollback recovery mechanism* respectively. The three mechanisms are integrated into a runtime, called LyRT. This section briefly explains those mechanisms, but first, the concept of roles is illustrated.

1.3.1 Roles as Variants

The role concept has been applied in many disciplines, ranging from data modeling to conceptual modeling to programming [Ste00, KLG+14]. This concept consists of three main object abstractions: *players* (core objects), *roles*, and *compartments*. The relation between these three abstractions is defined as follows: players or core objects implement the base system logic. Roles are variants dynamically extending or adapting the behavior of the core objects to which they are bound. Roles are modeled as part of a compartment and collaborate with each other. The compartment also acts as a binding scope of core objects in which its activation adapts the core objects concerning the played roles. These abstractions are suitable for run-time variability in which core objects can be used to implement the base system and roles are implemented as variants. An activation of compartments facilitates the dynamic composition of core objects and their bound roles. Throughout this dissertation, the term *core objects* is used instead of *players* since the player denotes the role-playing object which can be either the core object or another role in the case of roles are allowed to play roles as well.

1.3.2 Dynamic Instance Binding Mechanism

We propose a *dynamic instance binding mechanism* which adheres to the role-playing model to achieve anticipated and unanticipated adaptation at the instance level. The mechanism

loosely binds role instances to the core objects by constructing a transient relation between them. The binding relation also contains a compartment instance with which the relation is associated. Conforming to run-time variability, the bound instances remain completely decoupled from each other while appearing as a single object. A lookup table is used to store the binding relations. Normally, programmers interact with the core objects, but a dynamic method dispatch selects the appropriated role for invocation based on the relation obtained from the lookup table. Therefore, manipulating the lookup table eventually triggers the adaptation. In case of unbinding, the relation is removed from the lookup table, and the associated role instances are destroyed making the core object to adapt accordingly.

In order to achieve context-dependent behavior, the dynamic method dispatch works only when a certain compartment instance is activated. In this regard, the same core object can bind to different roles located in different compartments, but the dynamic behavior of the core becomes effective if a particular compartment is activated. The dynamic instance binding mechanism allows a single compartment to be activated at a time in a thread. However, compartments can be active concurrently in different threads making the program parts to be adapted simultaneously and independent from each other.

In order to deal with unanticipated behavior at run time, the new role classes must be defined and compiled. Whenever the roles are loaded into the run-time environment by a dynamic class reloader, they immediately become available to be bound to other existing instances. In the case of removing existing roles, triggering the unbinding operation is necessary. Since the program is already running, role (un-)binding operations to support unanticipated adaptation are made through an Extensible Markup Language (XML) configuration file.

1.3.3 Object-Level Tranquility Mechanism

The tranquility concept [VEBD07] was proposed to find a consistent state, called *tranquil state*, to safely update the software components which engage in a *transaction*. A transaction is a series of method invocations that need to be executed atomically. The notion of this transaction is applied to component-based systems where participating nodes are presented as singletons, and their communication is statically defined via ports and connections. Due to these criteria, this concept is difficult to be applied at the object level [ESMJ10].

In order to achieve tranquility at the object level, we introduce a *consistency block* which is a construct to surround the block of code where a series of methods is executed, and we expect to have a uniform behavior regardless of any adaptation. The consistency block shares a similar characteristic of the transaction in the sense that it prevents the behavior of multiple objects engaged in the block from being changed. The consistency block exploits the dynamic method dispatch of the dynamic instance binding mechanism to hold the consistent behavior of those participating objects. Adaptation happening in the existence of the consistency block is deterred and soon becomes effective when the block has finished. The application developers must define the consistency block in the program. The program code outside the consistency block is considered to be performed in the tranquil state, and it is, therefore, safe to perform adaptation.

1.3.4 Rollback Recovery Mechanism

As bugs cannot be completely detected during development, we embrace them during execution as they can cause run-time failures. The main idea of a *rollback recovery mechanism* is to recover, upon failure, by rolling back to a recent checkpoint. The system generates a checkpoint before initiating a new adaptation. A *checkpoint* is a serialized representation of the current application configuration, i.e., the active compartments, a list of objects (representing roles) including their states and binding information reflecting the currently active play-relations between players and roles.

After the checkpoint is created, the system performs the specified adaptation, and the runtime reacts accordingly. The program may encounter failures caused by bugs which are introduced by newly installed or updated role implementations. The system has a specialized sensor to detect faults and to signal the runtime to roll back to the previous configuration by restoring the most recent checkpoint. That previous configuration is assumed to be error-free because failures had not been caught within that application configuration. Meanwhile, the runtime records the defective configuration to prevent it from being reactivated. The system also generates a notification to the developer responsible for bugs. The adaptation can be reapplied through unanticipated adaptation after the bug has been fixed.

1.4 Contributions

This dissertation potentially contributes to the field of adaptive software systems in which the integral concept is proposed with run-time variability in mind. Therefore, reusability and adaptability are the expected results of this work. In order to support adaptation comprehensively, we propose three mechanisms as briefly described in Section 1.3. Those mechanisms are our key contributions:

- **Dynamic Instance Binding Mechanism.** As explained in Section 1.3.2, this mechanism allows us to achieve anticipated and unanticipated adaptation at the instance level.
- **Object-Level Tranquility Mechanism.** As described in Section 1.3.3, this mechanism prevents the objects from changing their behavior while engaging in ongoing method executions in order to avoid inconsistency. An object is allowed to adapt its behavior only when it reaches a tranquil state.
- **Rollback Recovery Mechanism.** As mentioned in Section 1.3.4, this mechanism embraces and handles failures caused by software bugs resulting from a dynamic variant composition.

1.5 Publications

The publications which support the key ideas in this dissertation are fully described in Chapter 4 and are summarized as follows:

- Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "A dynamic instance binding mechanism supporting run-time variability of role-based software sys-

tems." In Companion Proceedings of the 15th International Conference on Modularity, pp. 137-142. ACM, 2016.

This paper presents the dynamic instance binding mechanism, a technique to achieve anticipated adaptation at instance level by utilizing the role concept. It also demonstrates a case study of a Tax Management System which is part of our validation described in Section 6.1.2. Besides, it provides a Snake Game to show that the mechanism is ready for unanticipated adaptation. This showcase is also part of our validation described in Section 6.1.3.

- Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "Consistent unanticipated adaptation for context-dependent applications." In Proceedings of the 8th International Workshop on Context-Oriented Programming, pp. 33-38. ACM, 2016.

  This paper offers an enhancement of the original tranquility concept to make it applicable for consistent adaptation at the object level. Additionally, it extends the previous paper by illustrating a run-time architecture that supports unanticipated adaptation. Moreover, the paper presents a File Transfer Application as a case study that is part of our validation described in Section 6.1.4.

- Martin Weissbach, Nguonly Taing, Markus Wutzler, Thomas Springer, Alexander Schill, and Siobhan Clarke. "Decentralized coordination of dynamic software updates in the Internet of Things." In Proceedings of the IEEE 3rd World Forum on the Internet of Things (WF-IoT), pp. 171-176. IEEE, 2016.

  This paper realizes the applicability of the two papers mentioned above in the adaptation of distributed systems in which LyRT was used as a local runtime waiting for a decentralized coordinating protocol to trigger the adaptation.

- Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "A Rollback Mechanism to Recover from Software Failures in Role-based Adaptive Software Systems." In Companion to the first International Conference on the Art, Science and Engineering of Programming, p. 11. ACM, 2017.

  This paper presents a consequence of dynamic variant compositions which may cause system failures due to the lack of test suite. Therefore, it proposes a rollback recovery mechanism for Role-based systems to handle execution failures. A case study is given which later becomes part of our validation described in Section 6.1.4.

1.6 Limitations

The goal of the dissertation is to propose the run-time architecture that features the adaptation by supporting variability and a partial update of the run-time behavior in terms of unanticipated adaptation. The prototype, a proof of concept, is presented as a software framework normally integrating to the mainstream OOP that allows programmers to write role-based applications.

However, the engineering principles of programming languages such as new language constructs, new compiler, formalization, and type systems are not the main focus. Additionally, this work does not intend to fully demonstrate Self-Adaptive Systems (SASs), which usually

rely on the control feedback loop to infer the adaptation [ST09]. The solution covers only the adaptation part executing in the local runtime without any consideration for the self-* properties of SASs.

1.7 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 introduces the foundation of run-time variability, and we include some key definitions that will be used for the whole dissertation. We demonstrate a chat server application as the motivating example. From that example, we derive the requirements for variability approaches that should satisfy the needs of run-time variability.

Chapter 3 discusses comprehensively related work in the perspective of run-time variability and adaptation of a runtime. We focus on variability approaches derived from programming language solutions and use the requirements that we developed in Chapter 2 to assess those state-of-the-art approaches.

Chapter 4 illustrates the key concept which includes (1) the dynamic instance binding mechanism to realize anticipated and unanticipated adaptions, (2) the consistency block to deal with arbitrary adaptation causing inconsistent behaviors, (3) the roll-back recovery mechanism to embrace bugs ensuring runtime progress and supporting continuous deployment. In addition, a full-fledged run-time architecture will be fully visualized and illustrated.

Chapter 5 explains an implementation technique of a prototype serving as a proof of concept for evaluation.

Chapter 6 presents the evaluation by implementing three case studies to demonstrate the practicability of the proposed run-time architecture. Besides, various performance metrics compared to the baseline counterparts will be discussed.

Chapter 7 concludes the dissertation by summarizing the contributions, and finally outlines potential future work.

Chapter 2

Background and Requirements Analysis

This chapter describes the foundations on which this dissertation is based. First, the concept of run-time variability which is used to achieve anticipated and unanticipated adaptation is illustrated. Second, a foundation of the role concept [KBGA15] is explained. Third, the tranquility concept [VEBD07], which defines a condition of a consistent state to safely update the system, is described. Finally, an example which is used to derive requirements for run-time variability is presented.

2.1 Run-time Variability

Variability is the ability of a software system to change its behavior corresponding to a specific need [GWT+14]. Variability specifies static and dynamic parts of a system in which changing the dynamic parts features adaptability. The static parts are typically fixed and used to implement core system functionalities. The dynamic parts are called *variants*, reusable software features which may interact with each other. Variants are designed as options that can be selected. Variability manages these variants by allowing them to be composed into the base system via *variation points*. A variation point specifies where a decision is to be made including how, when, and where variants may be introduced [Hil10].

Although variability features adaptability, the resulting adaptation is achieved differently based on when variants are managed. In Software Product Lines (SPLs), variability is used to synthesize different sets of variants in order to create multiple software products in the same product family. Since a composition of variants is done at development or build time, the adaptability in SPL can be expressed as those resulting products fit the need of respective clients. Variability is not limited to product lines but "it is a key fact of most, if not all, systems" [Hil10]. Many today's software systems are developed with variability in mind, e.g., self-adaptive systems [GWT+14]. Thus, rather than creating multiple software products, a single software product is created, but its behavior is adapted due to the changing of variants. Therefore, this variability is called *run-time variability*, in which variants are dynamically composed at run time in order to adapt the system's behavior.

Run-time variability offers adaptation, but this *adaptation is anticipated*. Anticipated adaptation is a process in which a system changes its behavior as a result of dynamic composition of variants that are given beforehand. Baresi et al. [BDNG06] mentioned that a running system might undergo a series of changes to address bug fixes and new requirements in order to operate in today's open-world settings. Therefore, run-time variability should extend its ability to incorporate new variants, which are unknown during design time, to the running system in order to address these new requirements. This ability is called

unanticipated adaptation. Unanticipated adaptation is a process in which a system changes its behavior as a result of a dynamic composition of variants that are unknown beforehand and only given at run time.

In this dissertation, run-time variability is the ability of a software system to change its behavior corresponding to a specific need in either anticipated or unanticipated manner. In literature, run-time variability that also supports unanticipated adaptation is called *run-time meta-variability* [HWS+09]. However, the term *run-time variability* is used in this dissertation since it is broad enough to cover both kinds of adaptation.

2.2 Roles

In 1977, Bachman and Daya first introduced roles for data modeling [BD77] with a limited notion of adaptation. Since then, the work on roles has been proposed in many domains ranging from databases [JKH+15] to conceptual modeling [Fow97, KLG+14] to programming languages [Her05, LA15]. Kühn et al. [KLG+14] mentioned that there is no common understanding of roles. This statement implies that researchers or practitioners proposed a different set of role features for their work. Steimann [Ste00] and Kühn et al. [KLG+14] compiled a list with all features from various domains in literature as presented in Table 2.1. While some features relate to model level($M1$), some are for instance level($M0$). This dissertation addresses those instance-level features, which are run-time aspects.

2.2.1 Nature of Roles

The given role feature list consists of three main abstractions: *player*, *role*, and *compartment*. Players, also called core objects, implement the core system behavior considered to be static over the entire lifetime. Roles encapsulate dynamic behavior that can dynamically extend or adapt the players' behavior. Compartments are a scope in which roles reside and collaborate. The activation of a compartment reifies a role binding process which adapts the player's behavior.

These role abstractions correspond to three natures, namely behavioral, relational, and context-dependent natures [KLG+14], which are suitable for run-time adaptation. The *behavioral nature* characterizes the objects that can acquire and abandon roles dynamically, adapting the object's behaviors. For instance, a person object can play a `student` or a `customer` role. As roles are defined as objects, they can also play other roles. The *relational nature* represents relations filled by roles which are subject to cardinality constraints. For example, a `student` played by a person works for a `professor` played by another person under the *work-for* relation, defined with a *many-to-one* cardinality constraint. The *context-dependent nature* realizes roles and relations encapsulating them within a context-dependent boundary scope, which is a compartment. For example, a person exclusively plays a `student` role in a university compartment and plays a `customer` role whenever that person enters a shop compartment. Not only roles but relations too are context-dependent because roles fill those relations within a compartment scope.

This dissertation mainly focuses on the behavioral and context-dependent nature. With respect to run-time variability, players or core objects represent the base system whereas roles represent variants. Compartments specify the variation points of players and roles

Table 2.1: Classifying features of roles. $M1$ and $M0$ are denoted as model level and instance level respectively.

No.	Description	Target
	Features compiled by Steimann, 2000 [Ste00]	
1	Roles have properties and behaviors	$M1$, $M0$
2	Roles depend on relationships	$M1$
3	Objects may play different roles simultaneously	$M1$, $M0$
4	Objects may play the same role (type) several times	$M0$
5	Objects may acquire and abandon roles dynamically	$M0$
6	The sequence of role acquisition and removal may be restricted	$M1$, $M0$
7	Unrelated objects can play the same role	$M1$
8	Roles can play roles	$M1$, $M0$
9	Roles can be transferred between objects	$M0$
10	The state of an object can be role-specific	$M0$
11	Features of an object can be role-specific	$M1$
12	Roles restrict access	$M0$
13	Different roles may share structure and behavior	$M1$
14	An object and its roles share identity	$M0$
15	An object and its roles have different identities	$M0$
	Features compiled by Kühn et al, 2014 [KLG$^+$14]	
16	Relationships between roles can be constrained	$M1$
17	There may be constraints between relationships	$M1$
18	Roles can be grouped and constrained together	$M1$
19	Roles depend on compartments	$M1$, $M0$
20	Compartments have properties and behaviors	$M1$, $M0$
21	A role can be part of several compartments	$M1$, $M0$
22	Compartments may play roles like objects	$M1$, $M0$
23	Compartments may play roles which are part of themselves	$M1$, $M0$
24	Compartments can contain other compartments	$M1$, $M0$
25	Different compartments may share structure and behavior	$M1$
26	Compartments have their own identity	$M0$

which are bound together. An activation of a compartment reifies the role (un-)binding processes to adapt the system's behavior.

2.2.2 Foundation of Compartment Role Object Model (CROM)

Compartment Role Object Model (CROM) [KBGA15] is a role-based model that captures the three natures of roles described above. CROM has been formalized, and it provides graphical tool support for modeling [KBRA16]. CROM is argued to be a promising modeling technique to represent the abstraction of the system complexity and dynamism by combining both relational and context-dependent property under the compartment. Besides compartment, role and object type, CROM gives another type of its elements called relationship type denoting role relations to fully support the three natures.

The distinction between these four elements in CROM is subtle. Kühn et al. [KBGA15] use three ontological properties, namely *rigidity*, *foundedness*, and *identity*, to differentiate between them. An instance of a rigid type can stand on its own without any dependence. For example, a `person` instance remains itself until it ceases to exist. An instance of a founded type can only exist if another instance of a rigid type exists at the same time. A `student` instance only exists when there is a `university` instance. The identity property describes the form of identity to be unique, derived or composite. A `person` instance has its own unique identity while the `student`'s identity is derived from the playing `person` due to the shared identity feature (Feature No. 14 in Table 2.1). A `teaching` relationship identity is composite because it is derived from the participation of `student` and `professor` roles in a `university`. A compartment, e.g., a university, has its state and behavior; its identity is unique. Table 2.2 summarizes the four types of CROM's concept with the three ontological properties. While a role instance is founded on the existence of a compartment instance, the compartment instance itself is also founded on the existence of participating roles. An activation of an empty compartment does not affect the players' behavior.

Table 2.2: Ontological foundation of CROM [KBGA15].

Property	Rigidity	Foundedness	Identity
Object Type or Natural Type	Yes	No	Unique
Role Type	No	Yes	Derived
Compartment Type	Yes	Yes	Unique
Relationship Type	Yes	Yes	Composite

CROM provides a constraint model on roles. In addition, it introduces *Role Groups* to apply the constraint, e.g., cardinality constraints, to a group of roles. The constraint in CROM is partly adopted from a model of Riehle and Gross [RG98]. The constraint model specifies that for a given pair of role types A and B, at least one role constraint value is defined. There are four constraint values: *role-dontcare*, *role-implied*, *role-equivalent*, and *role-prohibited*. The role-dontcare constraint specifies no constraint between role A and B. The role-implied constraint demonstrates the implication of an object that plays role A, also plays role B, but not vice versa. The role-equivalent constraint is a bi-directional role-implied constraint between A and B. The role-prohibited constraint prevents role A and B from being bound to an object within the same active compartment. These constraints map one-to-one of a given

pair of roles making them inflexible when dealing with cardinality. Role Groups overcome this problem by grouping roles with cardinality constraints. Kühn et al. [KBGA15] mentioned that Role Groups constrain a set of roles that an object is allowed to play simultaneously in a certain compartment. For example, persons can play `teaching_assistant` and `student` roles. A constraint specifies that `teaching_assistants` can assist ten `students` in a course, but they cannot assist themselves as a `student` of a particular course. By using Role Groups, this constraint is set as: $(\{(\{teaching\_assistant\}, 0, 0), student\}, 1, 10)$.

Kühn et al. [KLG+14] proposed a metamodel for role-based modeling, and CROM is only one possible role model which supports behavioral, relational, and context-dependent natures. Depending on the subset of role features to be addressed, other role models can be derived as well.

2.3 State and Behavioral Consistency

Adaptation changes state and behavior of a system in which parts of the base system are updated according to the composing variants. The system must be in a *consistent state* before the change is performed in order to ensure consistency [KM90]. "A consistent state is a state from which the system will be able to terminate correctly" [ESMJ10]. The correct termination refers to the point in which the system is ready to switch behavior. Since the system comprises multiple objects, a consistent state of the system depends on the state of individual objects which engage in run-time execution. Therefore, finding a consistent state of the system requires to find the consistent state of these objects.

Kramer and Magee [KM90] introduced the notion of *quiescence* which is a condition of putting a system into a consistent state before it is updated without restarting. The system model is represented as a directed graph in which *nodes* are the servicing entities, and *connections* reflect the interaction between those nodes. A node in their system refers to a computing node in distributed systems in which the node has only one single instance. The interaction between nodes is explicitly defined inside *transactions*. A transaction is a sequence of messages that must be executed atomically [KM90]. Since quiescence was proposed for distributed systems, the *messages* are passed over the networks to achieve a common service. From the perspective of a runtime of programming languages, messages refer to a series of method invocations of particular objects. The key idea of quiescence is not to update the nodes while they are engaging in the transaction because those nodes are not in a consistent state.

Quiescence is sufficient to place a node in a consistent state where the system can safely update the node. However, this approach causes a long disruption to reach a consistent state because a node inside the transaction is not allowed to update although it has finished its current services and will not service for the rest of the transaction. Vandewoude et al. proposed *tranquility* in order to to minimize this disruption with two observations [VEBD07]. First, a node should be able to update even when it is in a transaction if the node has not been servicing or will not service for the rest of the transaction. Second, a node engaging inside a *sub-transaction* can also be updated. A sub-transaction is also a transaction but it appears inside a parent transaction. Normally, nodes engaging in the transaction have no knowledge of the nodes participating in the sub-transaction. Hence, the nodes inside the sub-transaction are invisible to the parent transaction.

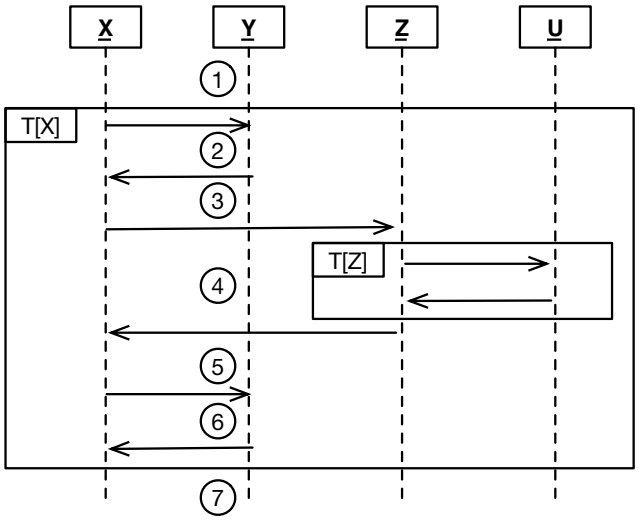


Figure 2.1: Nodes' updatability in transactions [VEBD07].

Figure 2.1 shows a transaction $T[X]$ which contains a sub-transaction $T[Z]$ and consists of four nodes (X, Y, Z, U). X initiates a communication in $T[X]$ where Y and Z are servicing nodes. Z further initiates a new sub-transaction $T[Z]$ where U is a servicing node. U is a hidden node from the viewpoint of X and Y. Regarding the nodes' updatability in a transaction, quiescence does not allow any node to be updated inside $T[X]$ so the nodes can only be updated at time 1 and 7. Similarly, U cannot be updated because it is hidden, and quiescence is not designed for this sub-transaction. In contrast, tranquility allows X and Y to be updated at time 1 and 7 while Z and U can be updated at any time except time 4 because they are servicing inside the sub-transaction $T[Z]$. Figure 2.1 clearly shows the benefit of the tranquility concept over quiescence. Tranquility minimizes the disruption time to update the nodes even they are participating in a transaction while a consistent state of those nodes is still maintained.

Although tranquility is more efficient than quiescence in terms of disruption, a consistent state is not always reachable [VEBD07]. Therefore, tranquility applies a fallback mechanism to switch to quiescence whenever waiting for the consistent state takes longer than a threshold period. The concepts of quiescence and tranquility apply to the component-based software systems and distributed systems in which nodes are singletons, and their interfaces are well-defined. Moreover, the communication between participating nodes is explicit via component's ports and connections. Therefore, it is statically determined which nodes are engaging in the transaction. In object-oriented systems, the notion of transactions is implicit, and classes, which are nodes in the tranquility concept, have multiple instances [ESMJ10]. In other words, there is no practical solution to achieve tranquility at the object level. Thus, consistently updating the object instance remains a challenging task to overcome.

This dissertation aims at providing a mechanism to achieve a consistent state at object level before performing anticipated and unanticipated adaptation. The term *object-level tranquility* is used throughout this dissertation to express the consistent state of objects in which adaptation can be safely performed.

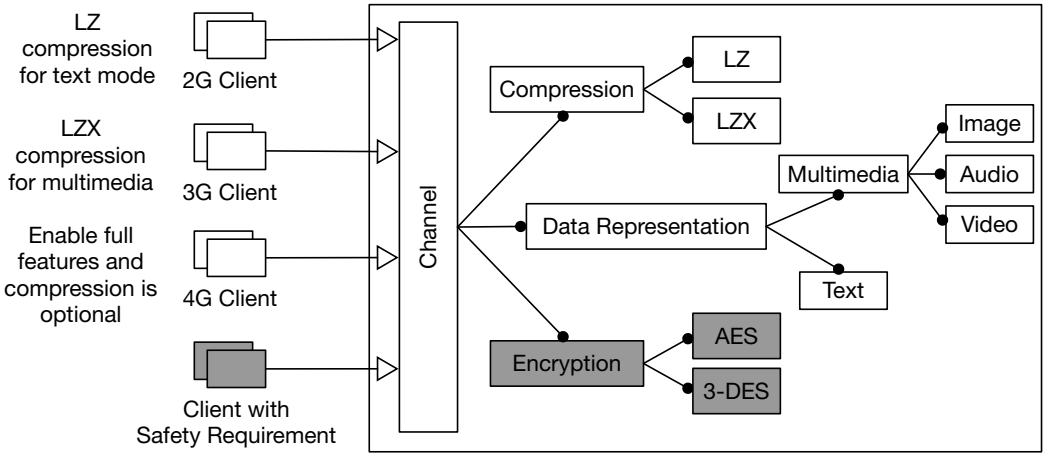


Figure 2.2: A chat server application with per-client adaptation.

2.4 Motivating Example

In order to highlight the need for run-time variability, a chat server application is used as an example. The example is motivated by a scenario of a typical adaptive software and is proposed to conform to the research questions. Afterwards, this example is used to derive requirements for approaches to support run-time variability.

The chat server application, depicted in Figure 2.2, supports multiple client connections simultaneously. The main functionality of the application is to handle data or message transmission between the server and clients. Messages can be transmitted in a raw, a compressed, or an encrypted format. Depending on client requirements, messages can be both compressed and encrypted before transmission. Technically, a shared `Channel` object is responsible for this transmission, and its behavior changes based on client requirements. The clients are heterogeneous in terms of network connectivity, for example, 2G, 3G, and 4G. With respect to these connectivities, the server chooses an appropriate data format for transmission. For example, the server only sends `LZ` compressed messages to 2G clients because of low bandwidth connection. The 3G clients have more bandwidth which allows them to transmit texts and images with a better `LZX` compression algorithm. The 4G clients have the bandwidth to support all multimedia formats, and data compression is optional.

During execution, chat server administrators may want to introduce encryption features (i.e., `AES` and `3-DES`) to the `Channel` object to respond to privacy concerns of their new users. Unlike `LZ` and `LZX` compressions, which are implemented and deployed to the runtime beforehand, the encryption algorithms are newly introduced at run time. Their introduction should not affect the existing client connections but rather give additional features for new clients or old clients with newly established connections.

Based on the description of this example, key features related to adaptation are illustrated as follows:

F1: Variations. Initially, the shared `Channel` object can send and receive only in a raw format, but this ability is variable in response to the types of connected clients which demand compression or encryption to the transmitted data.

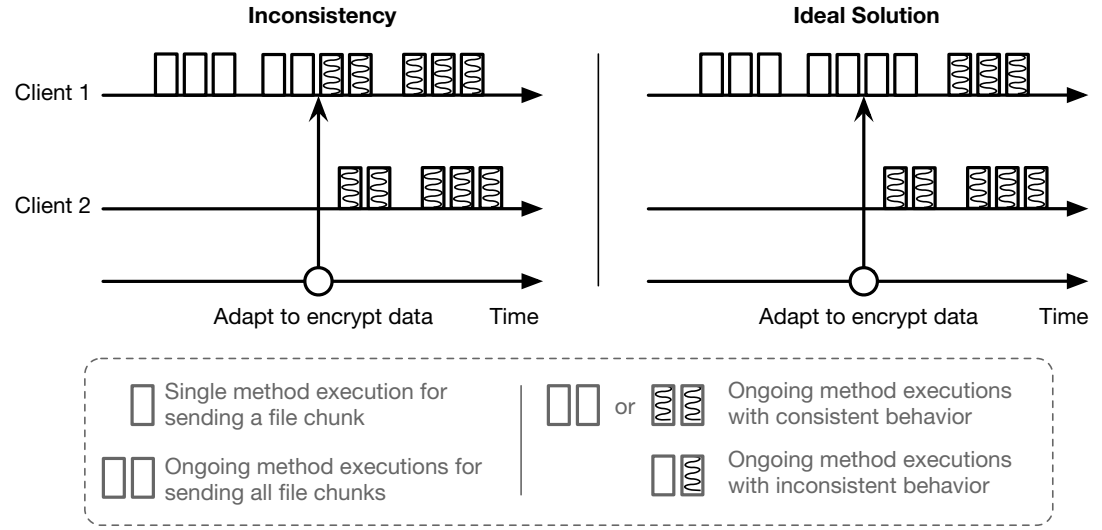


Figure 2.3: Behavioral change during a file transfer in the chat server application.

F2: Anticipated adaptation. Run-time adaptation is demonstrated when the shared `Channel` object switches between formats in order to transfer the data. The variants, such as `LZ` and `LZX` compressions, are deployed in advance.

F3: Unanticipated adaptation. Adding encryption features that have been unforeseen during development time requires a technique to load newly introduced variants and to dynamically bind the loading variants to the `Channel` object.

F4: Consequences of adaptation. Performing either anticipated or unanticipated adaptation may affect the run-time stability. The consequences are described as follows:

F4.1: State and behavioral inconsistency. A 4G client, denoted as `Client 1` in Figure 2.3, is receiving file fragments in a raw format, and in the meantime, the server is requested to add an encryption algorithm unanticipatedly. The shared `Channel` object adapts to the encryption behavior while it is still handling the transmission of the chunks without encryption for `Client 1`. This adaptation satisfies `Client 2` which requires security for data transmission, but it affects `Client 1`. The adaptation allows `Client 1` to receive file chunks partially in a raw format and partly in an encrypted format. Therefore, the chunks cannot be merged into a single file. This situation occurs as a result of the adaptation that is applied either during the execution of a long-running transaction[1] (i.e., sending file chunks) or when the system has not reached a consistent state.

F4.2: Software failures. Adaptation may happen frequently, and possibly variants are composed arbitrarily. These criteria make the system vulnerable to failures because bugs may appear in a composition. Bugs are common in software systems, and testing dynamic software systems is challenging [TSCS17]. This problem becomes even more critical when unanticipated adaptation is supported. For example, a composition of either `LZ` or `AES` to the `Channel` object is tested

[1]The transaction is discussed in Section 2.3

independently without error, but a bug may appear when the two variants are composed together to perform encryption and compression on the transmitted data. A simple bug, such as a `DivideByZero`, may cause the runtime to crash.

F5: Continuous deployment. Runtime should be conformally extensible by adding new features on-the-fly without the need for a system restart. For instance, the `Channel` object can have additional behavior such as new efficient compression and encryption algorithms. Software failures should not prevent from having an adaptation in such erroneous situations. For example, the composition of `LZ` and `AES` variants causes a `DivideByZero` bug that has been uncaught before deployment. Therefore, when performing unanticipated adaptation, the bug is manifested and crashes the runtime. In continuous deployment settings, the runtime resists to failures and lets the developers fix the bug and reapply the defective variants once again through unanticipated adaptation.

2.5 Requirements for the Run-Time Variability

Run-time variability has to support both anticipated and unanticipated adaptation while the consequences of those adaptations are also taken into account. This section presents requirements to accommodate the demanding features of the motivating example, discussed in Section 2.4. A summary of these requirements corresponding to the described features of the motivation example is given in Table 2.3. These requirements will also be used to assess the related work presented in the next chapter.

R1: Modularity. Modularity does not only improve reusability and comprehensibility of the application code but is also the foundation for run-time variability. Variants have to be defined modularly from the base system avoiding tangled code that limits the flexibility of the adaptation. At programming language level, variants should be first-class citizens associating to programming language abstraction with a mechanism to handle the dynamic activation of the variants during run time. Even though variants are kept as separated as possible from the base for reusability and adaptability, their connections to or glue code with the base system need to be constructive and informative in order to improve the comprehensibility of the program code. This requirement corresponds to *F1: Variations* of the motivating example.

R2: Dynamic Activation. Variants are bound to the base system with a particular binding mechanism controlling the dynamic addition and removal of variants. In contrast to static activation, where that binding process happens before run time or once at load time, dynamic activation takes place at run time. Dynamic activation transiently occurs several times in different places of a program. At programming-language level, dynamic activation can be scoped in a confined adaptation block to facilitate the change of several objects. The adaptation is deactivated automatically when the scope expires. Dynamically scoped activation can be extended to support parallel activation in a multi-threaded environment. For example, the chat server application allows the channel object to be activated simultaneously with different variants to serve multiple client requests. This requirement responses to *F2: Anticipated adaptation* of the motivating example.

Adaptation, a result of dynamic activation, is triggered from various sources such as application developers, end-users, and contextual values. At programming-language

level, adaptation is supported by language abstraction which allows application developers to provide the activation block explicitly for adaptation anywhere in the code. Some adaptive applications deliver the control of adaptation to their end-users to select specific variants of their demanding functionalities, i.e., plugins. In mobile applications or intelligent ambient systems, the adaptation typically comes from sensing data of the observable execution environment that are interpreted to be contextual information for adaptation transition.

Variants differ in size, form, or state (i.e., type, instance, component, service, etc.). Choosing between these variants for adaptation is a design decision. Since this dissertation aims for a run-time system derived from a language solution, the granularity of adapting variants can either be at *type* or *instance* level. Supporting adaptation at type level is not as flexible as at instance level because several instances instantiated from the same type uniformly adapt even if some of them are not in need. Another adverse effect is a long disruption of adaptation which requires all the instances to be in a consistent state in order to avoid inconsistency. Therefore, we prefer the adaptation to be applied at instance level.

R3: Late Variants Adoption. As a system keeps changing over time, there is a need for adding new requirements previously unknown. Supporting the adoption of new variants dynamically at run time improves service availability as the runtime does not require a restart. This requirement responses to *F3: Unanticipated adaptation* of the motivating example.

R4: Object-Level Tranquility. Dynamic adaptation is about changing the state and behavior of a running application. Adaptation can be applied to a single object or a group of objects at once. At run time, these objects are perhaps executing a certain task in a long-running transaction, i.e., sending file chunks in the motivating example. Therefore, adaptation must be applied when these objects are in a consistent state in order to avoid inconsistency. Object-level tranquility is an essential condition for placing these objects in a consistent state. This requirement responses to *F4.1: State and behavioral inconsistency* of the motivating example.

R5: Failure Handling. Developing robust systems is another challenge to be tackled. Variants are variable parts, and they are composed into the system. Testing the variants independently is possible but testing their compositions is quite difficult because the composition is realized only at run time, and the order of the composition matters. The number of possible compositions grows exponentially as the number of variants increases. Additionally, current testing frameworks lack contextual constructs or do not exist for context-dependent applications [SGP12a]. In such a case, variants are likely to be infested with bugs. This situation becomes even more critical when variants are introduced later at run time. Hence, after the adaptation, executing these defective compositions causes the runtime to crash. A mechanism to handle such failures is required. This requirement responses to *F4.2: Software failure* of our example.

R6: Continuous Deployment. This requirement is derived from *R3* and *R5*. If the system supports requirement *R3*, the defective variant composition, captured by a system supporting *R5*, can be fixed and reapplied using unanticipated adaptation. Therefore, supporting both *R3* and *R5* responses to *F5: Continuous deployment* in the motivating example.

Table 2.3: List of mapping between requirements and corresponding required features.

No	Requirements	Corresponding Features in the Example
1	R1: Modularity	F1: Variations
2	R2: Dynamic Activation	F2: Anticipated adaptation
3	R3: Late Variants Adoption	F3: Unanticipated adaptation
4	R4: Object-Level Tranquility	F4.1: State and behavioral inconsistency
5	R5: Failure Handling	F4.2: Software failures
6	R6: Continuous Deployment	F5: Continuous deployment

2.6 Chapter Summary

Run-time variability in the context of this dissertation has to capture both anticipated and unanticipated adaptation. Run-time adaptation comes at the expense of consequences that should be taken into consideration, such as inconsistency and failures. Roles can be a suitable candidate for variants due to their dynamic property that captures the behavioral, relational, and context-dependent nature. In order to avoid inconsistency, adaptation must be performed when the system is in a consistent state. Tranquility is a sufficient condition to find a consistent state of the system. The chat server example was to show the need for run-time variability which is later used to derive requirements for run-time variability. In the next chapter, the current state of the art will be comprehensively discussed based on the requirements mentioned before.

Chapter 3

Related Work

In Chapter 2, we identified a number of requirements for variability approaches that should be satisfied in order to support run-time variability that handles the coexistence of anticipated and unanticipated adaptation as well as resolving techniques when facing the consequences of such adaptation. These requirements are used to assess the state of the art of run-time mechanisms that are designed for adaptive software systems or context-dependent applications. Developing such systems or applications is challenging because of their transient and dynamic adaptability. Several approaches have been proposed to combat this problem at different levels, for example, software architecture, middleware, and programming level. Primarily, we focus on language-level approaches because the adaptation is integrated to the core machinery of a language allowing us to steer the extremely customizable adaptation.

In recent development, many linguistic approaches have been proposed tailoring to dynamic adaptation with elegant language constructs to coordinate various types of variants and to constitute a dynamic method dispatch for behavioral adaptation. This chapter conducts a systematic review of state-of-the-art run-time variability approaches realized by programming language models addressing the above requirements. We ignore the discussion on higher-level techniques such as component-based approaches or middleware solutions because they adapt software components at a coarse-grained level. Before we analyze the mainstream programming languages, we investigate languages facilities such as inheritance, mixins, and traits followed by a brief discussion of design patterns. Besides these language-related solutions, we also discuss the domain of DSU as it provides a platform for unanticipated adaptation. Finally, we synthesize all the discussed approaches into a survey list for a better comparison.

3.1 Language Facilities

This section focuses on the traditional element of reuse and modularization in the form of language support. Although language facilities do not address the run-time adaptation, they are the core building block of software composition. We discuss their strengths and weaknesses before we provide an evaluation that is summarized in Table 3.1.

3.1.1 Inheritance

In OOP, inheritance is the fundamental mechanism for code reuse. Modern programming languages such as Java and C# support only single inheritance to reduce unnecessary complexity and to avoid problems. Although single inheritance has been widely accepted, it is not

expressive enough to specialize the state and behavior in complex class hierarchies [SDNB03]. Multiple inheritance, on the other hand, provides more expressiveness but introduces more complexity which limits its usability [DMVS89, SG99]. Cook also mentioned that "multiple inheritance is good, but there is no good way to do it." [Coo87].

Indeed, multiple inheritance introduces ambiguity and raises a major problem known as the *diamond problem* which is described as a situation of a class inheriting from two different classes that share the same superclass. In this scenario, the class would get inherited twice which eventually leads to a duplication of state encapsulated in the superclasses. As a consequence, this results in serious state consistency issues [OAC+04]. Moreover, as pointed out by Schaerli et al. [SDNB03], although the aforementioned problems are intentionally avoided, multiple inheritance is still not the most appropriate element of reuse.

Another problem offered by the composition mechanisms of both single and multiple inheritance is that they are rigid and static at design time making them less suitable for run-time variability which requires variants to be (de-)composed dynamically.

3.1.2 Mixins

Mixins were first introduced to solve the problem of lacking expressiveness in single inheritance, and mixins also address the problem in multiple inheritance on the ambiguity of the diamond problem (i.e., naming, behavioral and state conflicts). The notion of mixins can be found in *Flavors* [Moo86] which defines mixins as small units of reuse not necessarily complete. Mixins can be *mixed in* orthogonally at arbitrary places in the class hierarchy [SDNB03]. Ancona et al. [ALZ00] define mixins as follows:

> *"A mixin is a uniform extension of many different parent classes with the same set of fields and methods, that is, a class-to-class function." [ALZ00]*

As already explained in Section 3.1.1, the diamond problem, which appears in multiple inheritance, does not exist in single inheritance. As mixins are based on single inheritance, they do not suffer from this problem either. However, mixins still maintain the same expressiveness as multiple inheritance. Mixins are applied to classes *one at a time* which generates new subclasses in a single inheritance hierarchy [DNS+06]. Mixin composition is linear so that the order of the mixins in the hierarchy matters. Methods defined in a mixin simply extend or override the methods of the class to which they are applied.

To be more concrete, we revisit the example from Ancona et al. [ALZ00]. Consider a schematic class as follows: `class H1 extends P1 { desc }` where `P1` is a parent, `desc` is a set of properties and methods are declared in subclass `H1`. Assuming that another `H2` is the heir of a parent `P2` but requires the same set of `desc` which can be expressed as `class H2 extends P2 { desc }`. Without mixins, in the single inheritance scenario the `desc` is duplicated resulting in redundant code. Mixins solve this problem by placing the common `desc` into a mixin structure `M` and constructing composition syntax to the classes `H1` and `H2` as follows:

```
mixin M { desc }
class H1 = M extends P1
class H2 = M extends P2
```

Snippet 3.1 shows a mixin implementation of the chat server application described in

Section 2.4 by using language constructs from Jam [ALZ00] which is a mixin prototype extended to Java. A `Channel` class is defined with basic `send` and `receive` function (Lines 2-10) and a mixin `LZCompression` has a specific `send` method performing compress function (Lines 13-18). For the sake of simplicity, the data to be compressed is enclosed with `<LZ>` tags. A `ChannelWithLZ` is a mixin class declared by allowing a `LZCompression` mixin to be a subclass of the `Channel` class (Lines 21-23). Therefore, when a `send` method is called (Line 32), the `send` method of mixins is first invoked to compress the data before it passes to the `send` method of the `Channel` object through `super` keyword (Line 16).

Snippet 3.1: The chat server application implemented with Jam mixins [ALZ00].

```
1 //Channel class with basic behaviors
2 class Channel {
3   void send(String data) {
4     Network.send(data);
5   }
6
7   String receive() {
8     return Network.receive();
9   }
10 }
11
12 //Mixin declaration
13 mixin LZCompression {
14   void send(String data){
15     String c_data = "<LZ>" + data + "<LZ>"; //perform compression
16     super.send(c_data); //call super method which is Channel object
17   }
18 }
19
20 //New class declaration with Mixins
21 class ChannelWithLZ = LZCompression extends Channel{
22
23 }
24
25 //Main program
26 public class Main{
27   public static void main(String[] args){
28     Channel channel = new Channel(); // without mixins
29     ChannelWithLZ chLz = new ChannelWithLZ(); // with mixins
30
31     channel.send("DATA"); //send without compression
32     chLz.send("DATA");    //send with compression
33   }
34 }
```

Similar to inheritance, mixins tackle reusability and composition at design time. They do not address run-time adaptation because they cannot be mixed in and mixed out dynamically. Evidently, mixins do not apply to the instance level, i.e., to the `Channel` object for example, but mixins are a separate type that extends the `Channel` class. In a nutshell, mixins follow the paradigm of inheritance to specialize behavior by means of subclassing. Therefore, mixins are still not applicable for run-time variability.

3.1.3 Traits

Traits are designed to eliminate the problem of multiple inheritance and mixins while improving the expressiveness and code reuse. Traditionally, traits are only concerned with behavioral reusability, thus there is no state support, to avoid the diamond problem found in multiple inheritance. Unlike classes, traits are a small unit of reuse containing a list of specific methods for composition, and thus traits are relatively *incomplete*. Unlike mixin composition that is hierarchical, trait composition is flat meaning that the semantics of a class using traits is the same as the class constructed from all the non-overridden methods of traits and placed inside itself. That is "if class `A` is defined using trait `T`, and `T` defines methods `a()` and `b()`, then the semantics of `A` is the same as it would be if `a()` and `b()` were defined directly in the class `A`." [SDNB03]. Additionally, mixin composition is linear while composition in traits is symmetric as the order does not matter. For example, given a trait composition of $T = A + B$, then a composition of $S = B + A$ is the same as T. Conflicting methods must be resolved explicitly. Traits also support multi-level or nested composition. However, the flattening property is still maintained. For example, given a trait composition of $T = A + X$, where $X = B + C$, then a composition of traits $S = A + B + C$ is the same as T.

Snippet 3.2 demonstrates how traits are declared and composed in the Scala language with respect to our chat server application. The normal `Channel` object is declared as a regular class in Lines 1-10 while the `LZCompression` trait extends the `Channel` object. In Scala, an object can be composed with trait either at the class declaration level (Lines 21-23) or at the instance initialization level (Line 34). In the main program, invoking the `send` method of objects with the trait (Lines 31, 35) dispatches to the `send` method of the `LZCompression` trait because that `send` method is embedded into the object during composition. Therefore, the transmitted data is compressed by the trait's method implementation and then sent over the network with the `super.send` method (Line 16) that executes the `send` method of the `Channel` object.

Snippet 3.2: The chat server application implemented in Scala Traits [OAC+04].

```
1 //A Basic Channel
2 class Channel {
3   def send(data: String): Unit ={
4     Network.send(data)
5   }
6
7   def receive(): Unit ={
8     return Network.receive()
9   }
10 }
11
12 //A trait with specialized LZ compression
13 trait LZCompression extends Channel{
14   override def send(data: String): Unit ={
15     val c_data = "<LZ>" + data + "<LZ>"
16     super.send(c_data); //call to Channel.send()
17   }
18 }
19
20 //A class with traits composition
21 class ChannelWithLZ extends Channel with LZCompression{
22
23 }
```

```
24
25 //Main program
26 object Main extends App {
27   val channel = new Channel // Basic channel feature
28   channel.send("DATA")      // Send with original behavior
29
30   val chLz = new ChannelWithLZ //Specialized channel with traits
31   chLz.send("DATA")            // Send with compression
32
33   //initialize class with trait
34   val anotherChLz = new Channel with LZCompression
35   anotherChLz.send("DATA")  // Send with compression
36 }
```

Originally, traits were proposed for the behavioral composition of a class without state support [SDNB03] but stateful traits were later presented [DNS+06]. However, these traits are considered static, i.e., the composition is fixed at compile time. Similar to inheritance and mixins, static traits do not address run-time adaptation as traits cannot be (de-)composed dynamically. On the other hand, dynamic traits such as *Chai*$_3$ [SD05] allow for the dynamic replacement of traits making it possible for run-time adaptation.

Table 3.1: Evaluation of language facilities.

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Inheritance	■	□	□	□	□	□
Mixins [BC90, ALZ00, Moo86]	■	□	□	□	□	□
Traits [SDNB03]	■	□	□	□	□	□
Dynamic Traits [SD05]	■	■	□	□	□	□

■: supported, ⊞: partially supported, □: not supported

Evaluation

The language facilities, in general, offer a great mechanism for code reuse and thus improve the comprehensibility and extensibility of software systems. Super class in inheritances, mixins, and traits are modular building blocks for these language facilities. However, they support static composition occurring at compile time, but they are less flexible at run time. Concerning the requirements we set for run-time variability in Section 2.5, these mechanisms respond only to *R1: Modularity*. Besides, dynamic traits offer an operation for trait replacement during run time to support run-time adaptation. Therefore, dynamic traits satisfy both *R1: Modularity* and *R2: Dynamic Activation*. Nonetheless, dynamic traits have no mechanism to adopt a newly introduced trait for unanticipated adaptation. Moreover, its adaptation is rigid lacking contextual and scoping activation. The consequences arising from these adaptations remain open for exploration. Hence, dynamic traits do not fulfill the

rest of requirements. Table 3.1 summarizes the evaluation of these mechanisms. Context Traits [GMCC13] adds contextual behavior to dynamic traits making it more advanced in terms of adaptation. Hence, we categorize it as a COP language which will be discussed in Section 3.3.5.3.

3.2 The Role Object Pattern

Design patterns are proposed to handle specific problems which repeatedly appear in software development practice. Design patterns are a promising technique for reusability, maintainability, and extensibility [GHJV95]. Some design patterns allow software artifacts to adapt their behavior at run time. However, this kind of adaptation is rigid in the sense that the code has to be designed and implemented upfront with those patterns [Car13, p. 27]. Cardozo [Car13] has already briefly mentioned and pointed out the limitation of existing design patterns that support adaptation such as *state pattern*, *strategy pattern*, *decorator*, *abstract factories*, and *dynamic proxies*. In addition to that, in this section, we discuss the role object pattern, which closely relates to our solution in terms of roles, and which is the earliest attempt to bring roles to programming languages for dynamic adaptation.

Bäumer et al. [BRSW98] introduced the Role Object Pattern to overcome the disadvantages of Fowler's role models [Fow97]. The pattern consists of four participants—*Component*, *ComponentCore*, *ComponentRole*, and *ConcreteRole*. The Component is an abstract interface containing both role management protocol and specialized role operations. The ComponentCore implements the Component interface including the role management protocol to handle the creation and management of the ConcreteRole instances. Similar to Fowler's Role Subtype model, the ComponentRole stores a reference to the ComponentCore and implements the specialized role operation interfaces by forwarding requests to its core attribute. The ConcreteRole is the role-specific implementation where the extension behaviors to adapt are located and the ComponentCore instantiates the ConcreteRole as an argument.

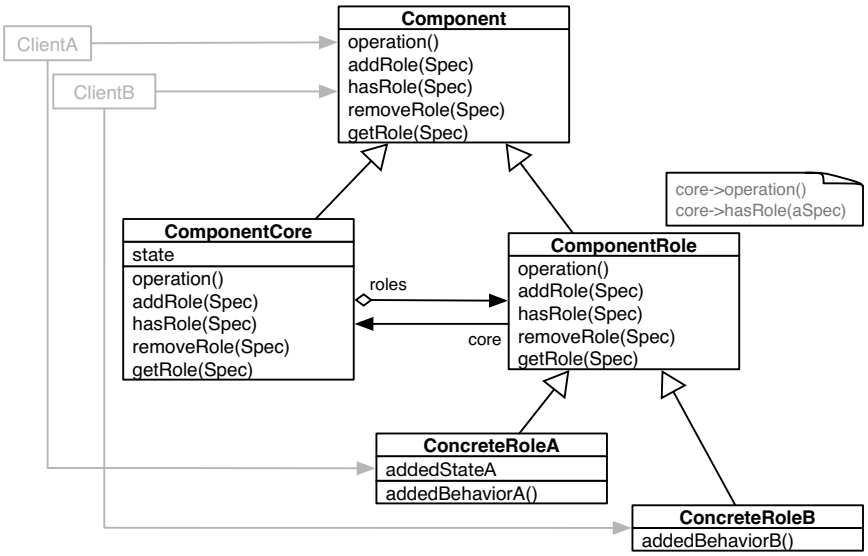


Figure 3.1: Structural diagram of the Role Object Pattern [BRSW98]

The role object pattern handles a core object and its role instances separately with a reference, but they maintain a conceptually shared identity to appear as one object which is subject to

object schizophrenia problem [Ste07].

> *"Object Schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity)." [Her10]*

Broken identity is a common issue in object schizophrenia problem. Herrmann [Her10] explained that the roles stored in a set-like structure might duplicate their identity. Therefore, those duplicated roles no longer adapt their core object behavior because the method dispatcher cannot find the appropriate roles due to their duplicated identity. The role object pattern suffers from this problem because roles (ConcreteRole) are stored in a set-like structure in ComponentRole that shares with ComponentCore as shown in Figure 3.1.

Besides the object schizophrenia problem, the role object pattern experiences an explosion of the class hierarchy when dealing with multiple roles that share a similar structure. Figure 3.2 shows the class diagram of our chat server example using the role object pattern. `ChannelCore` is a class supposing to play two main roles: compression and encryption (i.e., `AES`) roles. The compression role has two more specializations that are `LZ` and `LZX` implemented as a subclass of the `CompressionRole`. Adding these specializations results in an extra level of class hierarchy in the role object pattern making the class hierarchy rigid and fragile. Moreover, it also incurs run-time overhead as roles are queried recursively [BRSW98].

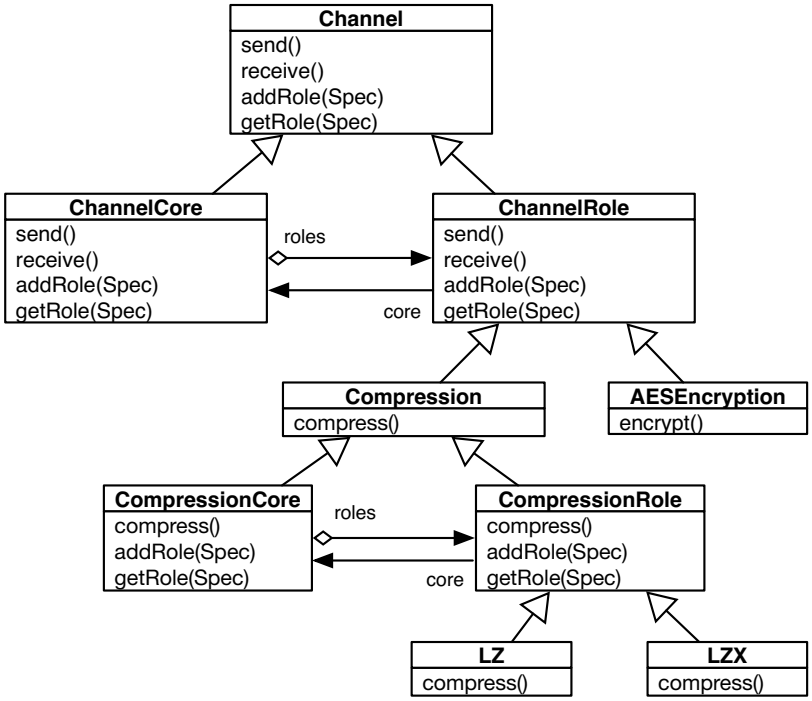


Figure 3.2: The chat server class diagram using the Role Object Pattern

In summary, the role object pattern has several advantages over Fowler's models [Fow97] in terms of decoupling the core and roles as well as the ability to add and remove role objects dynamically. The disadvantages are the object schizophrenia problem and the explosion of the class hierarchy for a large number of roles. This pattern also does not address unanticipated adaptation because roles are priorly required to construct class hierarchy.

Evaluation

The role object pattern achieves *R1: Modularity* by having separated role components loosely coupled to the core object even though it requires an explicit reference. It also satisfies *R2: Dynamic Activation* by offering a role management protocol for adding, removing and querying roles dynamically at run time. Although it provides run-time adaptation, its adaptation is rigid lacking contextual and scoping support. Additionally, it is incapable of supporting unanticipated adaptation, *R3: Late Variants Adoption*. Table 3.2 summarizes the evaluation of the role object pattern.

Table 3.2: Evaluation of the Role Object Pattern.

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Role Object Pattern [BRSW98]	■	■	□	□	□	□

■: supported, ⊞: partially supported, □: not supported

3.3 Language Solutions

This section discusses variability approaches derived from programming language solutions. The languages to be discussed are Subject-Oriented Programming (SOP), Feature-Oriented Programming (FOP), Meta-Programming (MP), Aspect-Oriented Programming (AOP), Context-Oriented Programming (COP), and Role-Oriented Programming (ROP). Originally, each solution was proposed as a single research agenda, but continuous contribution in the field resulted in a diversity of work on the same subject. For example in the domain of ROP, we can classify four groups based on the nature of roles. To keep the discussion focused, in each solution, we first describe the general concept and point out the key features that the solution tackles regarding run-time variability. A sample code snippet, which realizes our chat server application, described in Chapter 2, is given to exhibit how the solution implements run-time variability. Afterward, we classify the work of that particular domain based on its commonality. Finally, we evaluate each classification of the respective solution to our requirements.

Due to the great volume of research on the target domains, we cannot provide a complete review of every work. Instead, we present the prominent research development of each language area that has been well-discussed in the community. Furthermore, we do not review comprehensively each individual approach, but we highlight the essence of variability support and the connection of those approaches to our requirements.

3.3.1 Subject-Oriented Programming (SOP)

Harrison and Ossher [HO93, HOM95] critiqued the pure OOP languages that have only a single view as they cannot adequately describe any object from different views. Subjectivity is a concept to extend the object to have multiple views while still keeping the core object encapsulated. In SOP, *subjects* are "collection[s] of state and behavior specifications reflecting a particular gestalt, a perception of the world at large, such as is seen by a particular application or tool." [HO93]. Back to our motivating example, the `Channel` object can have multiple views to define its additional behavior such as `LZ` and `LZX` compression. In this section, we cannot give a code snippet which implements our chat server application because the Us language [SU96], a SOP language, presents incomplete syntax.

3.3.1.1 Subject Composition

Early work on SOP addressed only the composition domain rather than the behavioral adaptation [OKH+95, CHOT99]. In SOP, objects are created with subject binding through composition operators to transfer state and behavior from a subject to an object using method dispatch. Similar to inheritance, the subject composition into the object is performed at design time [UOK14]. A well-known extension to SOP is the multi-dimensional separation of concerns which is proposed to overcome the typical problem of composition concerning only one single dominant dimension of separation, also called the "tyranny of the dominant decomposition." [TOHSJ99]. The abstraction of the subjective concept provides a higher level concept of separation of concerns and modularization. More importantly, this idea has heavily influenced the development of other types of programming language solutions primarily designed for adaptation, and those solutions will be discussed later in this chapter.

Evaluation

Since this solution uses subjects as variants decoupling from the core objects, it satisfies the *R1: Modularity* requirement. There is no dynamic activation has been discussed in this solution. Therefore, the rest of the requirements are not supported. Table 3.3 shows the summary of this evaluation.

3.3.1.2 Subjective Dispatch

Smith and Ungar [SU96] proposed another kind of subjective programming, with their language called *Us*, to capture different perspectives of an object by means of a *layer*. The layer is the smallest but reusable unit containing specific behavior and state. A perspective is an ordered sequence of layers that is wrapped around the core object enabling an object perspective. Different perspectives can be applied to the core object and thus adapt the object's behavior. Their concept of perspective is similar to the subjectivity concept.

The Us language introduced *double dispatch* extending the typical method dispatch of OOP to include perspectives for method resolution resulting in a three-dimensional dispatch. Procedural programming relies on one-dimensional dispatch because the function can be called without specifying the sender. OOP relies on two-dimensional dispatch by placing an object as the message sender. Subjective dispatch, as in the Us language, puts both object and subject (perspective) into the method dispatch to enable a three-dimensional dispatch.

Nonetheless, this dynamic method dispatch lacks contextual support, which is addressed within the four-dimensional dispatch in COP proposed by Hirschfeld et al. [HCH08].

Evaluation

Similar to the subject composition solution, this solution still maintains the modularity of its layers and, therefore, satisfies the *R1: Modularity* requirement. Besides, its double dispatch technique offers a basic dynamic activation which is considered to support the *R2: Dynamic Activation* requirement. Table 3.3 summarizes this evaluation.

Table 3.3: Evaluation of Subject-Oriented Programming (SOP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Subject Composition [HO93, HOM95]	■	□	□	□	□	□
Subjective Dispatch [SU96]	■	■	□	□	□	□

■: supported, ⊞: partially supported, □: not supported

3.3.2 Feature-Oriented Programming (FOP)

FOP is a language solution to structure, customize and synthesize large-scale software systems with the core concept of *features* which can be activated and deactivated as needed [AK09]. A feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems." [KCH+90]. FOP is used to implement these features which can refine the functionality of the shared code base or other features in an incremental manner [GS12]. Configuring and activating a set of features creates several new software product lines with distinct behavior that share the common code base.

Snippet 3.3 presents the sample code of the chat server application implemented in Jak [BSR04]. Jak, a Java extension for FOP, allows implementing features using the `refines` keyword. A `Channel` class (Lines 2-10) is a code base that may be refined by features to have distinct behavior. Features are defined similar to a typical class (Lines 13-18). The `Super` keyword in Line 16 is to call the super method of a feature or code base when composed. The main program initializes the `Channel` object, and the `send` method is called in Line 26. The behavior of the `Channel` object depends on whether the features are assembled through configuration during built time. Therefore, we can assemble multiple product lines varying the `Channel` object behavior either with its own or with the compression feature if the `Channel` object composes of a compression feature. The composition code is not shown in Snippet 3.3 because the composition is made with FeatureIDE [TKB+14] which automatically generates the composition code.

Snippet 3.3: The chat server application implemented in Jak [BSR04].

```
1 //Code base for Channel
2 public class Channel{
3   public void send(String data){
4     Network.send(data);
5   }
6
7   public String receive(){
8     return Network.receive();
9   }
10 }
11
12 //Refinement LZ compression functionality over Channel
13 public refines class Channel {
14   public void send(String data){
15     String c_data = "<LZ>" + data + "<LZ>";
16     Super().send(c_data); //Call super's method
17   }
18 }
19
20 //Main class. Features composition is made by configuration at built time
21 public class Main{
22   public static void main(String... args){
23     Channel channel = new Channel();
24
25     //Behavior of send() depends on whether a feature is composed
26     channel.send("DATA");  //"<LZ>DATA<LZ>" if a feature is composed
27   }
28 }
```

3.3.2.1 Software Product Lines (SPLs)

SPLs are a software engineering approach that focuses on families of products which share commonalities but have different feature variations. SPLs also use the notion of features to produce different kinds of software products to fit the need of stakeholders. Although their design concept is based on features, their implementation is made without explicitly mentioning the features [AK09]. Usually, SPL's features are called *variants* integrated to the main code via *variation points*. In both SPL and FOP, features are implemented separately from the code base and they are composed together by various means of compositions. Most often, these fields overlap each other making the distinction between them difficult. We can view SPLs as a software architecture while FOP is the language solution. In this regard, FOP is a viable choice for developing SPLs. Because of the similarity, we place them into the same category to discuss their features against the run-time variability requirements.

Evaluation

R1: Modularity. FOP and SPL aim at modularity of *features* or *variants* [ALS08]. The decomposition of a software system in terms of *features* is the core requirement for both of them. This decomposition helps to construct well-organized software modules that later on

can be composed to meet the flavor of application scenarios. Apel et al. [ALS08] mentioned that "there are two key ideas of FOP: (1) Features are mapped one-to-one to modular implementation units called *feature modules* and (2) feature modules incrementally *refine* [other] feature modules already [presented] in a program". Wiring these feature modules obviously results in composition.

There are several ways to compose the features and these depend on the language design. For example, programmers can use C preprocessor `#ifdef` directives for the selection of features. This method may accommodate only a few features or it leads to "`#ifdef` hell" [SC92]. Feature composition in Prehofer [Pre97] and Jak [BSR04] relies on mixins which have some limitations discussed in Section 3.1.2, and the composition also needs a collaboration that borrows from role concept [RAB+92, VN96, SB02]. The development of AOP also influences the adoption of feature composition [LKKP06, Gri00, LSSP06, MO04, KAB07, AKLS07]. FeatureC++ [ALRS05], one of the FOP languages for C++, adopts the concepts of aspects and mixins for its feature composition.

To ease feature composition and configuration, a tool like FeatureIDE [TKB+14] is developed as a framework for the Eclipse Integrated Development Environment (IDE) to manage the development and to compose features. FeatureIDE supports several kinds of FOP implementation techniques such as AspectJ [KHH+01], FeatureHouse [AKL09], FeatureC++ [ALRS05], Jak [BSR04], Preprocessor directives and more to come.

R2: Dynamic Activation. FOP and SPL address feature composition bound statically to the code base before program execution to create product lines of a product family that are executed independently. Therefore, they are product-based or software-based adaptations while run-time variability emphasizes the adaptation of software features at run time in a single execution environment. Thus, FOP and SPL are not designed for run-time adaptation. That means, they cannot satisfy the requirement of dynamic activation. Comparable approaches that deal with this dynamic activation are discussed in Section 3.3.2.2.

R3: Late Variants Adoption. FOP and SPL are not designed to support unanticipated adaptation because they do not satisfy the *R2: Dynamic Activation* which is the prerequisite of the later variants adoption. Hence, they do not match this requirement.

R4: Object-Level Tranquility. Composing features interact with or depend on each other to be functional. Therefore, constraints between features are necessary to verify if the features contradict each other or if they need each other for their operations. Kang et al. [KCH+90] introduced the concept of feature-oriented domain analysis with the notion of *feature models* that describes the relationships and dependencies of a set of features belonging to a specific product. Figure 3.3 shows the feature model for the `channel` object in our chat server application. These constraints eliminate the conflicts significantly during the composition at development time but not at run time. Therefore, this requirement is not fulfilled because tackling the safe point to adapt the system characterized in the tranquility concept is infeasible.

Other Requirements. Both FOP and SPL present no match for other requirements. Table 3.4 visualizes this evaluation.

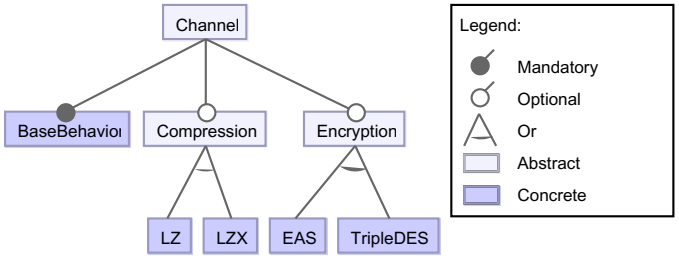


Figure 3.3: A feature model for the chat server application developed in FeatureIDE [TKB+14].

3.3.2.2 Dynamic Software Product Lines (DSPLs)

DSPLs follow the design principle of SPLs, but they manage variants dynamically in the configurable space for adaptable software systems through an explicit model called *late variability*. Similar to variability in SPL, which is a distinct property of software products, the dynamic variability can be represented as a dynamic feature that can be activated dynamically [DMFM10, HHPS08, HPS12, INPJ09, RSPA11, RSAS11].

The variants can be bound either *statically* at program compile time (i.e., FOP, SPL) or *dynamically* at run time (i.e., DSPL) [RSPA11]. There is a trade-off between these two binding mechanisms. Static bindings offer fine-grained customization and optimization without sacrificing the performance overhead (e.g., SPL built for embedded systems). In contrast, dynamic bindings introduce extra overhead but provide flexible feature addition to enable run-time adaptation [RSPA11]. However, not all features can be bound at run time. For example, features related to Central Processing Unit (CPU) architectures should be bound statically. In this regard, Rosenmüller et al. [RSAS11] introduced a hybrid binding technique balancing between performance and dynamic adaptation. The approach reduces the run-time overhead by statically combining multiple features into *dynamic binding units*, which are loaded and bound to the base system at run time. This technique minimizes the number of binding operations, and thus it improves the system performance. However, the overall system efficiency depends on whether all the bound features in the dynamic binding units are used or some of them are just loaded without usability.

Evaluation

R1: Modularity. DSPLs adopt the principle of variants separation from SPL but dynamically manage those variants for run-time adaptation. Hence, we consider DSPL to meet these requirement criteria without further discussion.

R2: Dynamic Activation. The main mechanism to enable dynamic variability in DSPL underlies the power of a dynamic feature binding mechanism that allows composition and decomposition of variants or features at run time with solving all the specified interdependencies and constraints between those features. Different dynamic binding approaches have been proposed. For example, Trinidad et al. [TCPB07] and Lee et al. [LK06] use a component-based approach to reconfigure variants at run time. FeatureAce [RSPA11] and

the work of Rosenmüller et al. [RSAS11] are integrated frameworks for DSPL developed on top of FeatureC++ [ALRS05]. Dinkelaker et al. [DMFM10] support dynamic variability based on the dynamic feature model and their architecture is implemented on top of the dynamic aspect run-time environment with meta-aspect protocol [DMB09]. Service-Oriented Architecture (SOA) can also be used for dynamic feature binding [INPJ09]. At the language level, rbFeature [GS12] is an FOP language implemented in the Ruby language to support dynamic variability and run-time adaptation based on the activation of a set of features. rbFeature relies on Metaobject Protocol (MOP) and functional programming to implement its features. However, feature adaptation is not triggered automatically with respect to executing context but through user intervention, for instance, from a graphical interface [CGDM11]. Despite rigid adaptation and lacking contextual activation, we assert that the DSPL satisfies this requirement.

R3: Late Variants Adoption. Although not mentioned explicitly, rbFeature [GS12] brings the power of the Ruby language that allows variants to be introduced and instantiated lately at run time. This is because features are implemented with a special Ruby object called `Proc`. `Proc`, as explained by Gunther et al. [GS12], is an anonymous block of code like other objects but has a unique ability. Similar to closures in functional programming, `Proc` can reference variables in its own creation scope. More importantly, code fragments written in `Proc` can be evaluated at run time by `eval` method. Therefore, rbFeature partially supports this requirement. Similarly, DSPL introduced by Helleboogh et al. [HWS+09] proposes a *meta-variability* model to document the way that the variability model may evolve. They use the concept of plug-ins as the late adoption of features to enable features addition on-the-fly though they lack programming support. Since we cannot find enough evidence showing that DSPL provides late variants adoption with full function, we assume DSPL to partially support this requirement.

R4: Object-Level Tranquility. Similar to SPL, variants in DSPL go through the process of feature modeling that defines interactive constraints for the composition as exhibited in Figure 3.3. Even if the composition happens dynamically, finding a consistent state to adapt the system is not addressed. Therefore, this requirement is not satisfied.

Other Requirements. DSPL supports neither *R5: Failure Handling* nor *R6: Continuous Deployment*. The evaluation summary can be seen in Table 3.4.

3.3.3 Meta-Programming (MP)

Maes [Mae87] brought computational reflection to OOP and defines "computational reflection to be the behavior exhibited by a reflective system, where a *reflective system* is a computational system which is about itself in a *causally connected* way." [Mae87]. Hence, computational reflection or just *reflection* is the ability to examine itself and its operational environment to possibly change its behavior. To perform this self-examination, reflective systems have a data structure representing their structural and computational aspects that are normally called *metaobjects*. Metaobjects are located in the metalevel that is causally connected to their objects at the base level. Metaobjects can be manipulated in the same manner as standard objects. Due to causal connection, changing metaobjects implies a

Table 3.4: Evaluation of Feature-Oriented Programming (FOP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Software Product Lines (SPLs)[1]	■	□	□	□	□	□
Dynamic Software Product Lines (DSPLs)[2]	■	■	⊞	□	□	□

■: supported, ⊞: partially supported, □: not supported
[1] Prehofer [Pre97], Jak [BSR04], FeatureC++[ALRS05], Aspect-based SPL [LKKP06, Gri00, LSSP06, MO04, KAB07, AKLS07]
[2] rbFeature [GS12], FeatureAce [RSPA11], Component-based DSPL [TCPB07, LK06], Aspect-based DSPL [DMFM10], SOA-based DSPL [INPJ09]

behavioral change of the objects in the base level. MP uses computational reflection to manage the metalevel through the abstraction of MOP. Therefore, MP allows the computational behavior of an object to be adjusted to meet a particular requirement without modifying the implementation of the object but rather by changing its metaobject [SW95]. This characteristic is suitable for building run-time variability.

To manage metaobjects, MOP is required. MOP contains a list of interfaces to give the programmer two distinct abilities, namely *introspection* and *intercession*. Introspection is the ability to examine the metaobject such as querying methods and inspecting the value of properties while intercession permits the code to be interceded and subsequently altered. The degree of supporting MOP for programming languages varies according to their nature. For example, statically typed languages such as Java provide rich structural introspection of an object while they limit intercession in a way that no structural class change is allowed. Dynamically typed languages such as Smalltalk, Ruby, Python, JavaScript, etc., express more freedom on the intercession. Although reflection is designed for reasoning about itself, it is often presented to support modularization by separating the concerns from the base code [SGP13]. Stroud [Str93] demonstrates that reflection addresses the non-functional properties such as fault tolerance and distribution transparency. A comparison study on dynamic adaptation over reflection, Dynamic Link Libraries (DLLs) and design patterns is made by Dowling et al. [DSC+99] in terms of performance, modular adaptation, and programming effort. The result concludes that reflection incurs more overhead, but it is flexible to offer significant advantages to support separation of concerns and adaptation.

Before we start evaluating the MP, we show how to implement our chat server example using Iguana/J [RC02], an extended framework for MOP to support dynamic behavioral adaptation. Snippet 3.4 shows the `Channel` and `LZCompression` class which are defined as typical classes. `MetaCompression` is the metaobject class that describes the interception of a method invocation where a method call is delegated to `LZCompression` in the base level (Lines 21-28). In this sense, the original behavior of the `Channel` object is propagated to an instance

of `LZCompression` through `proceed` method. The MOP of Iguana/J is defined to reify the method invocation (Lines 31-33) by manipulating the metaobject class, i.e., `MetaCompression`. In the main code, the protocol can be activated through the `Meta.associate` function that associates types or even instances to the reified protocol (Line 41) making the `Channel` object adapt its behavior. To deactivate, developers simply call `Meta.reset` method as shown in Line 44.

Snippet 3.4: The chat server application implemented in Iguana/J [RC02] using Reflection.

```
1 //Channel class with basic behaviors
2 public class Channel{
3   public void send(String data){
4     Network.send(data);
5   }
6
7   public String receive(){
8     return Network.receive();
9   }
10 }
11
12 //Specialized compression behavior of a Channel
13 public class LZCompression {
14   public void send(String data){
15     String c_data = "<LZ>" + data "<LZ>";
16     Network.send(c_data);
17   }
18 }
19
20 //Metaobject class
21 public class MetaCompression extends MExecute {
22   private LZCompression compression;
23
24   public Object execute(Object o, Object[] args, Method m){
25     if(compression == null) compression = new LZCompression();
26     return (proceed(compression, args, m));
27   }
28 }
29
30 //Protocol on adaptation (compiled with Iguana/J protocol compiler)
31 protocol CompressionProtocol {
32   reify Invocation: MetaCompression; //intercept method invocation
33 }
34
35 //Main Program
36 public class Main{
37   public static void main(String... args){
38     Channel channel = new Channel();
39     channel.send("DATA");  //Basic sending
40
41     Meta.associate("Channel", "CompressionProtocol"); //Activation
42     channel.send("DATA"); //Sending with compression
43
44     Meta.reset("Channel"); //Deactivate the adaptation protocol
45   }
46 }
```

Evaluation

We select three existing works in this domain that highly resemble the run-time variability. Those are Iguana/J [RC02], Reflex [TBSN01, TNCC03] and Geppetto [RDT08]. We revisit the abstract concept of run-time variability that is to manage variants and establish their binding to the base system. In this regard, MP uses metaobjects as variants and MOP as the binding mechanism.

R1: Modularity. Benefiting from the causally connected property, the decoupling of concerns can be achieved by placing those concerns in the meta level while keeping the core functions to the base object. MOP is extended to support the merging between concerns and main code. In this respect, Salvaneschi et al. [SGP13, p. 7] call it "*what* an object does is separated from *how* it behaves." Iguana/J [RC02], Reflex [TBSN01] and Geppetto [RDT08] follow this concept to achieve modularity for adaptation.

R2: Dynamic Activation. Iguana/J [RC02] is a perfect example to argue that MP through its MOP is powerful enough to design run-time architecture to support both anticipated and unanticipated adaptations. The reification of its protocol lets a metaobject change an object behavior in the base level through behavioral reflection. Iguana/J is implemented using a native dynamic library that is integrated closely to the Java Virtual Machine (JVM) via Just-In-Time (JIT) compiler interfaces making it less portable with different JVM implementations [TNCC03]. Another drawback of Iguana/J is the performance. Iguana/J is powerful but costly in terms of efficiency due to the occurrences of the reification process between metalevel and base level [RDT08]. *Partial behavioral reflection* is introduced to overcome this issue which provides a more fine-grained selection of *what* and *when* the object should be reified [TNCC03]. This reduces the unnecessary reflection process and thus improves the overall system performance. Tanter et al. [TNCC03] introduce the full-fledge partial behavioral reflection model in Reflex [TBSN01]. Reflex adds a hook mechanism as a link model that is applied to the individual objects at the base level using bytecode rewriting at load time. The reification process drives only the objects with the hook to change the behavior although their types are meta-associated.

R3: Late Variants Adoption. Iguana/J [RC02] relies on its own protocol to specify the association between objects and their metaobjects. The protocol description is compiled separately with a proprietary Iguana/J compiler, and the protocol triggers the activation any time using its own dynamic library with non-invasion to the existing code. Iguana/J supports unanticipated adaptation as reflective behaviors are associated dynamically with the target classes or objects at run time. Reflex [TBSN01, TNCC03], on the contrary, limits this requirement as it aims for portability and performance. Reflex relies on bytecode transformation at load time to insert the hook to the reifying objects. Even though reflective behavior occurs at run time, the hooking process needs to be anticipated at load time. Geppetto [RDT08] adopts the same partial behavioral reflection principle as Reflex, but it is extended to support unanticipated partial behavioral reflection. This can be done by shifting the hooking process to take place dynamically. Geppetto heavily depends on powerful reflection of the Smalltalk language and ByteSurgeon [DDT06] which transforms the object at run time to realize unanticipated adaptation.

Although MP is powerful enough to take advantage of run-time variability design, it is hardly portable across languages because different languages provide a different degree of MOP supporting. Additionally, there is a great effort to manage variability and drive adaptation. For example, developers need to manipulate metaobjects and MOP by themselves in Iguana/J to handle the adaptation which should be transparent to the developers.

Other Requirements. MP fulfills three requirements, i.e., *R1: Modularity*, *R2: Dynamic Activation*, and *R3: Late Variants Adoption*. The remaining requirements have not been investigated. It is possible to extend MP to support the rest of requirements, but an implementation mechanism is required. Table 3.5 summarizes the support of the three works described earlier with respect to the run-time variability requirements.

Table 3.5: Evaluation of Meta-Programming (MP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Iguana/J [RC02]	■	■	■	□	□	□
Reflex [TNCC03]	■	■	□	□	□	□
Geppetto [RDT08]	■	■	■	□	□	□

■: supported, ⊞: partially supported, □: not supported

3.3.4 Aspect-Oriented Programming (AOP)

AOP was proposed to respond to the lack of support for the rich expression of separation of crosscutting concerns in mainstream OOP [KLM+97]. Managing these concerns in OOP often results in unmanageable and tangled code. The concerns, for example, logging, security, persistent transaction, synchronization, and exception handling, are kept separate from the main code to improve modularization and thus comprehensibility and maintainability. Those separate concerns are composed to the main code to form a complete functionality by means of the *weaving* process. The concerns are often called *aspects* that usually contain a set of *pointcuts* and *advices*. In the main code, there are points which specify the behavior of the program execution such as method calls, variable initializations, etc. In AOP, these points are known as *joinpoints* to which *advices* are injected. To coordinate the injection, AOP defines a *pointcut* construct as a specification of *joinpoint* in the *aspect* module.

To understand the concept of AOP, we implement our chat server application using AspectJ [KHH+01], as illustrated in Snippet 3.5. As usual, the `Channel` object is defined as a normal class but its specialized behavior is implemented in aspect module (Lines 23-32). In aspect code, an execution pointcut is defined for the `Channel.send` method by specifying a custom decoration with `around` advice (Lines 28-31). Once aspect is woven, the code in the around advice is merged to the `send` method of the `Channel` object. The `proceed` (Line

30) is a multi-method which in this case represents the `send` method of the `Channel` object. Therefore, the call of `send` method in the main code (Line 18) results in compressed data before transmission.

Snippet 3.5: The chat server application implemented in AspectJ [KHH+01].

```
1 //Channel class with basic behavior
2 public class Channel {
3   public void send(String data){
4     Network.send(data);
5   }
6
7   public String receive(){
8     return Network.receive();
9   }
10 }
11
12 //Main program
13 public class MainApp {
14   public static void main(String... args){
15     Channel channel = new Channel();
16
17     //This is a joinpoint described in Aspect
18     channel.send("DATA"); //<LZ>DATA<LZ>
19   }
20 }
21
22 //LZCompression aspect
23 public aspect LZCompression {
24   //define pointcut over Channel.send() method
25   pointcut callSend(String data) : execution(void Channel.send(String)) &&
       args(data);
26
27   //"around" advice
28   void around(String data) : callSend(data){
29     String c_data = "<LZ>" + data + "<LZ>"; //Compression algorithm
30     proceed(c_data); //call the Channel.send()
31   }
32 }
```

With respect to the variability concept, *aspects* can be variants while *weaver* is the binding mechanism. A weaving process takes place at either compile time, load time or run time. Hence, the weaving strategy determines the adaptability of software entities. Compile-time weaving is considered as a static weaving process that presents better support for modularization at design time but it limits adaptation at run time [SGP13]. Both load-time and run-time weaving approaches are dynamic and open for adaptation. We discuss these two weaving strategies separately.

3.3.4.1 Static Weaving Mechanisms

Originally, AOP supported only compile-time weaving [KHH+01]. Compile-time weaving mechanisms merge advices to the code base at either source or byte code level. For example, `ajc`, an AspectJ compiler, is a post-processed weaving compiler that introduces the hooks for the advices to be integrated into the main program [SGP13]. This compiler-based implementation is beneficial in the sense that most of the errors are caught in advance and

without introducing unnecessary run-time overhead [KHH+01] but it limits the adaptability.

Evaluation

R1: Modularity. AOP is designed intentionally for separation of crosscutting concerns allowing the fragments of code to be scattered in different aspects. The pointcuts and advices, defined in an aspect module, provide expressively for merging the scattered codes into the code base. AOP, therefore, has high modularity but this modularity also comes at the expense of two main problems. On the one hand, the high degree of decoupling causes the *obliviousness* problem [FFN00] because there is no explicit glue code (i.e., such as binding) to link the aspect to the main code. Therefore, in order to comprehend the behavior of the program, developers need to be aware of all the relating aspects. Such problems can be minimized by using IDE that can visualize the main program and its aspects' symbiosis. On the other hand, the obliviousness problem leads to the *fragile pointcut* problem [KS04] because, AspectJ [KHH+01] for example, the joinpoint in the main code has to be matched with its pointcut defined in aspects. Hence, if the joinpoint is changed without modifying its expression pointcut in all aspects, there is a mismatch problem. However, this issue can be solved with the help of a compiler or a refactoring tool supported in IDE. Therefore, this static weaving AOP matches this requirement.

Other Requirements. Static weaving mechanisms such as AspectJ aim for separation of concerns and neglect run-time adaptation. Thus, they do not fulfill the *R2: Dynamic Activation* requirement. Besides supporting only the *R1: Modularity*, static weaving mechanisms do not satisfy the rest of the requirements. Table 3.6 summarizes this evaluation.

3.3.4.2 Dynamic Weaving Mechanisms

Static weaving approaches support better modularization at design time but limit adaptation at run time. Dynamic weaving or run-time weaving mechanisms overcome this limitation by shifting the aspect composition to load or run time.

The load-time weaving process is performed during the bootstrap process of application loading, and it happens only once at the initial stage. For Java-based implementation, bytecode weaving can be seen in the forms of load-time weaving, and usually, it relies on bytecode manipulating libraries such as BCEL [Dah99], Javassist [Chi00], and ASM [BLC02]. The advantage of this weaving strategy is the integration of legacy systems where source codes are unavailable; that is why they rely on bytecode rewriting tools to hook the advices to the existing code. Another benefit of this weaving strategy is to sense all the operational environments so that the adaptation can be pre-computed resulting in anticipated adaptation.

The run-time weaving mechanisms allow aspects to be woven dynamically and thus trigger adaptation. Several run-time weaving frameworks have been implemented using various techniques. Steamloom [BHMO04] customizes the JVM to support dynamic aspect weaving whereas PROSE [NAR08], Wool [SCT03], JAsCO [VSV+05], AspectWerkz [Bon04], and HotWave [VBAM09] rely on bytecode rewriting tools with different hot-swapping technologies. JAC [PDFS01] uses the Javassist [Chi00] class load-time MOP to implement aspect objects that can be dynamically deployed on top of running objects. TRAP/J [SMCS04]

takes advantage of the aspect principle to provide the necessary hooks to realize run-time composition while dynamic object extension can be done through wrappers under the framework of MOP. CaesarJ [AGMO06] offers virtual classes and propagates mixin composition for its aspect weaver.

Evaluation

R1: Modularity. Aspects of the dynamic weaving mechanism are just like the ones in static weaving approaches that have enough modularity expression. Therefore, we consider they meet this requirement without further discussion.

R2: Dynamic Activation. As explained earlier, dynamic weaving allows aspects to be (un-)woven dynamically to adapt the behavior of the core program. This adaptation changes the program flow that applies to the joinpoints. A pointcut and its advices, which are defined in an aspect, are merged into several joinpoints, and they also crosscut other aspects. This characteristic is challenging for the support of the instance level adaptation unless the particular pointcut is well described and filtered down to individual instances. Furthermore, Hirschfeld et al. [HCN08] categorized aspect into textual modularization instead of behavioral modularization that normally is found in inheritance. Therefore, aspects are suitable for structural adaptation or adaptation at a coarse-grained level rather than behavioral adaptation. Additionally, dynamically scoped activation for the fine-grained adaptation needs to be handled by the developers. Nonetheless, we judge dynamic weaving mechanisms to respond to this requirement.

R3: Late Variants Adoption. Although adaptation in AOP is considered structural and rigid [RC02], a few works manage to make unanticipated adaptation possible. PROSE [NAR08], for example, allows the aspect manager to insert and remove aspect in the breakpoint of the JVM through a special Java Virtual Machine Aspect Interface (JVMAI) but this solution is hardly portable. Different AOPs for a component framework such as JAC [PDFS01] and AAOP [JZ10], also provide a mechanism to inject aspects to the code base dynamically at a coarse-grained level. However, the full support of this requirement is not addressed by the authors. Thus, we consider dynamic weaver partially suitable for this requirement.

Other Requirements. Dynamic weaving mechanisms fully support the *R1: Modularity* and *R2: Dynamic Activation* requirements while partial fulfilling the *R3: Late Variants Adoption* requirement as exhibited in Table 3.6. The remaining requirements are not directly addressed by these mechanisms.

3.3.5 Context-Oriented Programming (COP)

Growing demand for supporting context-aware applications that adapt their behaviors based on the execution context imposes some challenges on software design. First, adaptation to the current context is transient, and it frequently occurs during the application lifecycle needing a novel activation mechanism. Second, dynamic adaptation modifies system logic crosscutting throughout the code base at run time that might tangle the code. Third, multiple contexts

Table 3.6: Evaluation of Aspect-Oriented Programming (AOP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Static Weaving Mechanisms [KHH+01]	■	□	□	□	□	□
Dynamic Weaving Mechanisms[1]	■	■	⊞	□	□	□

■: supported, ⊞: partially supported, □: not supported
[1] Steamloom [BHMO04], PROSE [NAR08], Wool [SCT03], JAsCO [VSV+05], AspectWerkz [Bon04], HotWave [VBAM09], JAC [PDFS01], TRAP/J [SMCS04], CaesarJ [AGMO06] and AAOP [JZ10]

often coexist that require the combination of many adaptation logics [SGP12a]. Although these adaptation characteristics can be implemented with multiple conditional statements, this traditional technique often results in a poor modularization subject to fragility that limits the comprehensibility and extensibility [CH05].

COP is an emerging language-level paradigm proposed to overcome the mentioned problems. COP treats the context as its first-class citizen and provides a higher abstraction to manage contextual adaptation through explicit language constructs. Context in COP is open; any computationally accessible information can be considered as context on which the behavioral variations depend [HCN08]. While allowing run-time adaptation, COP still maintains adaptation logic, known as *layer*, separately from the code base. COP tackles dynamic adaptation with a dynamic activation mechanism. The dynamic activation is the core mechanism to select and combine several layers to be activated and deactivated over the code base in response to context change [HCN08]. Therefore, COP covers structural modularity and dynamic adaptation that context-aware applications need.

Snippet 3.6 shows how COP fits to the implementation of our chat server application. As usual, the `Channel` class is defined as an ordinary class (Lines 2-10) while a specialized behavior is defined separately as a layer `LZCompression` (Lines 13-18). In the main program, invoking the `send` method of the `Channel` object in Line 26 results in normal behavior. When this method is called inside the `with` block of the corresponding `LZCompression` layer, the `send` method in the layer will be invoked. The `proceed` method in Line 16 works similarly to the `proceed` method in AspectJ [KHH+01] which calls the super method which is the `send` method in the `Channel` object. Thus, the result is data compression. The `with` block is the dynamic activation scope which expires the adaptation of the `Channel` object without explicit deactivation when it reaches the end.

In general, variants and their binding mechanism are the pillars to realize run-time variability. In this connection, COP languages present *layer* as variants decoupling from the code base but crosscutting the system which can be realized by *dynamic activation*. The dynamic activation selects and scopes the scattered layer definition to compose into the code base to

Snippet 3.6: The chat server application implemented in ContextJ [AHHM11].

```
1 //Channel class definition
2 public class Channel {
3   public void send(String data){
4     Network.send(data);
5   }
6
7   public String receive(){
8     return Network.receive();
9   }
10 }
11
12 //Layer definition of LZCompression
13 public layer LZCompression{
14   public void Channel.send(String data){
15     String c_data = "<LZ>" + data + "<LZ>"; //Compression algorithm
16     proceed(c_data); //super like method
17   }
18 }
19
20 //Main Program
21 public class MainApp{
22   public static void main(String... args){
23     Layer lzCompression = new LZCompression(); //initialize layer
24     Channel channel = new Channel(); //basic channel
25
26     channel.send("DATA"); //basic sending behavior
27
28     //activate layers
29     with(lzCompression){
30       channel.send("DATA"); //sending with compression
31     }
32   }
33 }
```

adapt the system behavior. Dynamic activation is an integral part of the COP languages, but different COPs introduce different dynamic activation with their own advantages and disadvantages. For example, ContextL [CH05] and ContextJ [AHHM11] provide activation locally on a per-thread basis eliminating the conflict issue. Ambience [GMC08] and Subjective-C [GCM+10], on the contrary, offer global activation that allows to trigger the adaptation from a global thread. This technique is suitable for any applications that need to adapt the behavior according to the external context. However, this activation may face state and behavior inconsistency that we will discuss in Section 3.3.5.3. Similarly, JCOP [AHM+10] and EventCJ [KAM11] control the layer activation based on user-defined events. ContextJS allows developers to create activation style on their own. However, it is argued that given this customization to the developers often leads to poor application design [KAM15]. ServalCJ [KAM15] uses the contexts and subscribers model to unify the existing activation schemes into a single solution. As a result, ServalCJ supports different kinds of activation found in ContextJ, Subjective-C, EventCJ, Flute, and ContextErlang. Flute [Bai12] is a reactive COP offering context pause and resumption to prevent from internal application state conflict resulting from adaptation. ContextErlang [SGP12b] combines COP and

the concurrency actor model for per-agent variants activation triggered by context-related messages. A detailed comparison of COP languages can be found in Cardozo [Car13, p. 77] and Salvaneschi et al. [SGP12a].

To keep the discussion focused, we classify the COP languages into three groups based on their activation mechanisms: *Local Activation*, *Global Activation*, and *Global Activation with Conflict Resolution*. The summary of their evaluation against the requirements can be found in Table 3.7.

3.3.5.1 Local Activation

This section evaluates the COP languages that support only local activation. The languages we classify to be in this group are ContextL [CH05], ContextJ [AHHM11], ContextS [HCH08], ContextR [Sch08], ContextPy [SP08].

Evaluation

R1: Modularity. COP deals with modularization through the concept of *layers* where variations are placed. However, COP supports *layer-in-class* and *class-in-layer* variations. The layer-in-class allows developers to define behavioral variations directly in the class of the code base which limits the modularization as the code base and layers highly couple. Even if layer-in-class improves readability, it suffers from reusability. In contrast, the class-in-layer keeps variations separate from the code base but still allows them to be activated in the main code. The code shown in Snippet 3.6 falls into this category. The class-in-layer is our preference because it supports better modularization easily extensible during run time but presents little effort on code fragment readability. We can perceive this characteristic as matching the modularity requirement.

R2: Dynamic Activation. Dynamic activation is the main mechanism of COP, and it is usually realized by dedicated language constructs that make the dynamic adaptation effortless for the developers. As shown in Snippet 3.6, the `with(){...}` construct enables the activation of a layer or multiple layers within its scope applying to corresponding objects. ContextJ also supports `without(){...}` block that can be nested inside `with()` block to disable certain adapting behavior.

R3: Late Variants Adoption. COP languages generally address anticipated adaptation assuming that developers know or plan in advance all possible variants [SGP13]. These variants are pushed to a stack-like structure making them ready to adapt the system behavior at run time when certain contexts are activated. Additionally, the local activation mechanisms support per-thread activation meaning that the adaptation happens locally in the current thread execution and ignores the activation from different threads. This technique limits the flexibility of unanticipated adaptation support as variants should be adopted at run time but activated globally from another thread. Therefore, we assume the local activation mechanism does not fulfill this requirement.

Other Requirements. Considering this classification, COP languages with local activation do not directly support the rest of the requirements.

3.3.5.2 Global Activation

This section discusses the evaluation of COP languages that support global activation. Those are JCOP [AHM+10], ContextErlang [SGP12b], ContextJS [LASH11], and ServalCJ [KAM15]. Table 3.7 reveals the evaluation of this classification against the requirements.

Evaluation

R1: Modularity. Similar to local activation mechanisms, modularity is still the central concept of the global activation approaches. *Layers* are still their units of reuse and adaptation, but the primary distinction is the activation scheme. We regard these global activation mechanisms to fulfill this modularity requirement.

R2: Dynamic Activation. In contrast to local activation mechanisms, the COP languages that support global activation allow variants to be activated globally and shared among all threads. This activation is flexible as it simplifies the process of variants activation through a managing thread [SGP12a]. However, this activation may result in system inconsistency because the activation from a global thread affects the adapting objects in the current thread that is executing a long-running method or transaction. Nonetheless, this activation mechanism matches the requirement.

R3: Late Variants Adoption. Most of the COP languages are designed for anticipated adaptation with an assumption that all the possible adaptations are given upfront. To enable unanticipated adaptation, a language should at least support late variants adoption and those variants become active via asynchronous activation from another thread. In this regard, ContextErlang can rely on Erlang language feature to leverage dynamic code loading but a simplifying procedure to coordinate the variants addition needs to be implemented. Since there is no direct support claimed by the authors, we consider this global activation classification to support this requirement only partially.

Other Requirements. The classified COP languages present no support for other requirements of run-time variability.

3.3.5.3 Global Activation with Conflict Resolution

This section evaluates the most sophisticated approaches of COP languages that provide different activation styles with constraint checking to avoid any conflict happening during the adaptation. The languages to be discussed are EventCJ [KAM11], Subjective-C [GCM+10], CoPN [Car13], Ambience [GMC08], Context Traits [GMCC13], and Flute [Bai12]. The evaluation summary of the classification of these languages is given in Table 3.7.

Evaluation

R1: Modularity. Just as the two groups discussed in Section 3.3.5.1 and Section 3.3.5.2, this category follows the same principle of modularity by separating the layers from their code base. Unlike traditional COP that treats context implicitly, languages such as Subjective-C, Ambience, and CoPN use context explicitly as their variant for adaptation that can be composed together comparable to a layer-based system. Similarly, Context Traits uses an explicit notion of context for adaptation, but it relies on the stateful *traits* as behavioral composition block. Flute realizes modules as its variants. No matter how these languages manage variations, the variants are designed as units of reuse and adaptation that crosscut the base system. Therefore, we judge them to support this requirement.

R2: Dynamic Activation. Dynamic enabling and disabling of the behavioral variations globally from different threads may yield unexpected application behavior executing in the current thread as pointed out in Section 3.3.5.2. Such strange behavior comes perhaps from a conflict between variants already deployed, or when the current running variants are removed in the course of adaptation. The COP languages in this classification do not only provide global activation styles but they also provide support to manage the conflict of adaptation resulting from the global activation.

Loyal, *Prompt* and *Prompt-Loyal* activation styles have been investigated [DVCH07, CGMD11] to design an activation strategy with conflict resolution. Context information is volatile as it is subject to change at any moment in time. The loyal strategy prevents objects executing a method in a context from abrupt change of their behavior in response to a new context activation. The languages discussed in Section 3.3.5.1 supporting local activation employ this loyal strategy. The prompt strategy, on the contrary, applies adaptation as soon as it is available meaning that the adapting objects change their behavior promptly. That makes the system potentially inconsistent as described earlier. The languages supporting global activation utilize this prompt strategy.

Loyalty makes the adaptation consistent but delays the adaptation whereas promptness promptly adapts the system behavior but the adaptation becomes potentially inconsistent. The prompt-loyal strategy combines promptness and loyalty in the sense that when a new context is activated, the adaptation is loyal within its current context until the objects finish their computation. However, new method calls are eager and select a behavioral configuration for the most recently activated context. Although lacking implementation, Cardozo et al. [CGMD11] proposed this concept for Ambience [GMC08]. Flute [Bai12], as described by Cardozo [Car13], also supports prompt-loyal activation. Adaptation in Flute is prompt but considered loyal because the currently executing context can be paused with state preservation and the adaptation transfers method invocation to a newly activating context. Once the method execution in the new context is done and the old context (the paused context) is re-activated, Flute resumes the execution of the paused context where it left off.

R3: Late Variants Adoption. Regarding unanticipated adaptation support by dynamically adopting new variants and activation, this mechanism is partially supported similar to Global Activation Mechanisms described in Section 3.3.5.2. Context Slice [CC15] introduces the context discovery model based on the ontology structure that groups contexts with

respect to their similar semantics. The context carries state and behavior advertised over the network. The discovered context that has not been known priorly can be adopted dynamically making the system adapting according to the specified context. However, context slice is not a complete COP solution, but it is an extension module to Context Traits that cannot represent all COP languages in this classification. Similarly, Mens et al. [MCD16] proposes a context-oriented software architecture which includes context discovery and resolution components to deal with the adoption of late variants. The prototype has not yet been fully integrated with a COP language. Therefore, we assume that the Global Activation Mechanisms with Conflict Resolution partially fulfill this requirement.

R4: Object-Level Tranquility. Although COP languages in this category fully support constraints for checking the conflict in adaptation, they do not address the tranquility. This requirement aims for finding the right time (consistent state) to perform adaptation when it is available. In other words, a system should not promptly react to the adaptation when the consistent state has not been reached. A consistent state is an idle or steady state when no active method or transaction is running. In this regard, the prompt-loyal strategy solves the problem at the method level but not at the transaction level. A transaction refers to a series of method executions to coordinate a common task or goal. Therefore, the adaptation should not be installed during execution of the transaction; otherwise, the system performs unexpected behaviors. Ambience [GMC08] introduces an `atomic{...}` code block comparable to the notion of the defined transaction to impede the adaptation. Similarly, context pause and resumption in Flute [Bai12] also express this concern. However, both of them do not consider parallel adaptation where an object is shared among other threads and the object may have distinct behavior per thread (i.e., the case of the channel object in our chat server application). Therefore, we assume that Ambience and Flute partially support the requirement.

Other Requirements. None of the COP languages in this category addresses software failures resulting from layer composition as the introduction or the withdrawal of adaptation. Hence, they support neither *R5: Failure Handling* nor *R6: Continuous Deployment*. The evaluation is summarized in Table 3.7.

3.3.6 Role-Oriented Programming (ROP)

The concept of roles is relatively old in computing, and it has been applied for data modeling [BD77] and conceptual modeling [Kri96, Kri98, KØ96, KM96, KLG+14]. Role modeling has been proven to capture the abstraction of dynamic and complex systems better than traditional OOP modeling [KBGA15]. ROP is designed to realize the dynamism of the role concept as well as the model for the run-time adaptation.

ROP aims at behavioral adaptation through the play-relation between objects and roles. At run time, this play-relation constitutes binding and unbinding operations of the object and its roles for polymorphic behaviors resulting in adaptation. We select ObjectTeams/Java (OT/J) [Her05], a well-known ROP within the community, to demonstrate the implementation of our chat server application. Snippet 3.7 shows that the `Channel` class representing a player that carries the basic behavior for data transmission (Lines 2-10). `CompressionNetwork` is a *team* that when activated reifies the role binding process (Lines 13-24). `LZCompression` is a

Table 3.7: Evaluation of Context-Oriented Programming (COP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Local Activation[1]	■	■	□	□	□	□
Global Activation[2]	■	■	⊞	□	□	□
Global Activation with Conflict Resolution[3]	■	■	⊞	⊞	□	□

■: supported, ⊞: partially supported, □: not supported
[1] ContextL [CH05], ContextJ [AHHM11], ContextS [HCH08], ContextR [Sch08] and ContextPy [SP08]
[2] JCOP [AHM+10], ContextErlang [SGP12b], ContextJS [LASH11] and ServalCJ [KAM15]
[3] EventCJ [KAM11], Subjective-C [GCM+10], CoPN [Car13], Ambience [GMC08], Context Traits [GMCC13] and Flute [Bai12]

role played by the `Channel` player providing specialized behavior to the player (Lines 15-23). In OT/J, roles are inner classes defined inside a team class and developers have to configure the binding process between player and role statically. The *callin* binding represented with `<-` symbol (Line 22) denotes as a method dispatch that transfers the call of the `send` method from the `Channel` object to the role. In the main program, the `Channel` player and the `CompressionNetwork` team are initialized as ordinary objects, but the behavior of the `Channel` object changes through team activation (Line 33). For example, the invocation of the `send` method in Line 34 dispatches to the `send` method of the `LZCompression` which performs compression before it calls back to the `Channel` player with the `base.send` construct in Line 18. As a result, `<LZ>DATA<LZ>` will be transmitted over the network.

Snippet 3.7: The chat server application implemented in OT/J [Her05].

```
1 //Channel class is a player
2 public class Channel {
3   public void send(String data){
4     Network.send(data);
5   }
6
7   public String receive(){
8     return Network.receive();
9   }
10 }
11
12 //Team declaration where roles are inside
13 public team class CompressionNetwork {
14   //LZCompression is a Role class
15   public class LZCompression playedBy Channel{
16     callin void send(String data){
17       String c_data = "<LZ>" + data + "<LZ>";
18       base.send(c_data); //call base (Channel) behavior
```

```
19      }
20
21      //configure callin binding from player to role
22      send <- replace send;
23    }
24 }
25
26 //Main program
27 public class Main {
28   public static void main(String... args){
29     Channel channel = new Channel(); // a player
30     CompressionNetwork c_net = new CompressionNetwork(); // a team
31
32     channel.send("DATA");  //Basic sending behavior
33     c_net.activate();      //Team activation
34     channel.send("DATA");  //Send with compression
35   }
36 }
```

In the previous sections, we have already presented several types of variants from different language-based solutions which are suitable for run-time variability. They are, for instance, *subjects*, *features*, *metaobjects*, *aspects* and *layers*. None of them mention explicitly the identity of these variants. Roles, which are variants for ROP, have their own identity, and they are independent entities from the core object or player. Roles are composed dynamically to their players with a binding mechanism to provide polymorphic behavior for those players. Such dynamic binding activation of ROP realizes run-time variability. To evaluate ROP languages against our requirements, we classify them into four groups based on the nature of adopting roles: *Static Roles*, *Dynamic Roles*, *Relational Roles* and *Contextual Roles*. Table 3.8 summarizes the requirement evaluation of these role classifications.

3.3.6.1 Static Roles

When dealing with complexity, developers use abstract concepts such as modeling to represent a system. Role modeling enhances this abstraction to visualize the complex and dynamic systems better than traditional object-oriented approaches. However, role modeling is insufficient without a realized programming support. JavaStage [BA12] is a programming technique realizing the role modeling that fundamentally focuses on role composition at development time and ignores dynamic adaptation. Similar to mixins and traits, JavaStage aims for code reuse while retaining advantages of role modeling. Any ROP languages with such intention are placed in the category of Static Roles.

Evaluation

R1: Modularity. Tackling modularity is the major concern of JavaStage. In its model, roles are units of composition that are defined as typical classes. Roles contain state and behavior composed to the object with a `play` directive. JavaStage provides an interpreter for translating to plain Java. During the translation process, the code of the bound roles is copied to the core object as an inner class. JavaStage has been evaluated independently to show its strength on modularity and code reuse. For example, Barbosa and Aguiar [BA13] compared JavaStage with traits [SDNB03] on code reuse. The result revealed that roles are

capable of reducing a significant amount of code than the traits [SDNB03] does. JavaStage is also proved to reduce replicated code significantly more than traditional refactoring method does [BA13].

Other Requirements. Static roles solely address software composition at design time for reusability without concerning run-time adaptation. Therefore, they do not satisfy the remaining requirements.

3.3.6.2 Dynamic Roles

In contrast to Static Roles, Dynamic Roles target run-time adaptation through a role binding processes. The resulting programming languages, such as Chameleon [GØ03] and Rava [HNL+06], typically address the dynamic behavioral extension of an object. A summary evaluation of Chameleon and Rava is given in Table 3.8.

Evaluation

R1: Modularity. Since Chameleon and Rava adopt the role concept and implement roles as separated entities from the core, they meet the modularity requirement. For example, Chameleon represents its roles as *constituent methods* similar to advices in AOP described in Section 3.3.4. Rava uses the mediator pattern to manage roles, and those decouple from the core instance. Since the player and its roles are unrelated, a mediator is used to manipulate logical references between player and role through a logical identifier denoting as `<coreObject, roleObject>`.

R2: Dynamic Activation. Even though Chameleon makes use of aspect weaving for its binding mechanism, its constituent methods (roles) are attached in advance to a respective core object for run-time adaptation. Rava stores the logical identifier into a list structure and this list is used to implement a dynamic method dispatch by exploiting reflection. Thus, we consider both Chameleon and Rava support this requirement.

R3: Late Variants Adoption. Both Chameleon and Rava do not support unanticipated adaptation as they limit the adoption of new roles at run time. Evidently, Chameleon uses static aspect weaving for its binding mechanism which needs to give beforehand the roles. Similarly, Rava requires an interpreter to translate its code to plain Java for logical identifier manipulation. Thus, we consider they do not address this requirement.

Other Requirements. Both Chameleon and Rava fail to meet other requirements.

3.3.6.3 Relational Roles

Static roles present role composition for static systems. Dynamic roles offer run-time adaptation for dynamic views of an object. None of them capture role relationships. Roles interact with each other in the form of relationships. For example, a professor role interacts

with student roles via the *teaching* relationship. Rumer [BGE07] introduces relationships as first-class citizens where roles are placed inside those relationships. By doing this, several constraints can be enforced in a relationship improving the system's robustness. In Rumer, roles can be accessed only from the relationship, not from the player, resulting in limited behavioral adaptation of an object [KLG+14]. We summarize the Rumer evaluation against the requirements in Table 3.8.

Evaluation

R1: Modularity. In Rumer, roles are regular classes carrying their own identity, state and behavior. Roles participate in a relationship type without inheritance support but they constrain each other. This decoupling makes Rumer fulfilling this requirement.

R2: Dynamic Activation. Roles exist in a relationship type. At run time, Rumer stores roles in a collection of objects that can only be accessed from the relationship. According to a review from Kühn et al. [KLG+14], Rumer supports at least feature numbers 3, 4, 5 of the role feature list, depicted in Table 2.1. Such support means that Rumer offers run-time adaptation by playing roles. We do not fully agree because there is no notion of playing roles presented in the approach. Therefore, the adaptation from the player perspective is rather abstract. In this regard, we assume Rumer to support this requirement only partially.

R3: Late Variants Adoption. Rumer presents no support for unanticipated adaptation because it is limiting the dynamic activation from the player perspective. Additionally, roles are statically given and defined in the relationship. Consequently, Rumer does not meet this requirement.

R4: Object-Level Tranquility. Rumer introduces the concept of *relationship invariants* to express the consistency constraints between participating roles in a relationship and between relationships themselves. The former refers to *intra-relationship* invariants that can be expressed in the form of multiplicity. The latter refers to *inter-relationship* invariants that can be constrained. For example, *Students* take a *Course* under the *Attend* relationship. A *Student* is hired to assist a *Course* under the *Assist* relationship. This invariant can be defined as $Attend \cap Assist = \emptyset$, which means the students are not allowed to assist the course in which they take part. While the constraints enforce run-time integrity, they do not solve the adaptation problem characterized in the tranquility concept.

Other Requirements. The rest of the requirements has not been addressed in Rumer, and thus they are not supported.

3.3.6.4 Contextual Roles

ROP languages in this classification rely on the context-dependent nature of roles described in Chapter 2 to construct their programming model to support context-dependent adaptation. Such context acts as a boundary scope of roles and their relationship. Some ROP languages

provide a context-like entity (the boundary scope) as their first-class citizen to encapsulate this context-dependent property, but they name it differently. For example, OT/J [Her05] names this entity a *team*, powerJava [BBVDT06] uses the term *institution*, EpsilonJ [TUI05] and NextEJ [KT09] refers to *context*, and SCROLL [LA15] denotes it as a *compartment*. Hence, we place OT/J, powerJava, EpsilonJ, NextEJ, and SCROLL into this classification for evaluation for which the summary can be found in Table 3.8.

Evaluation

R1: Modularity. Similar to the three categories described above, Contextual Roles follow the same principle to isolate role variants independently from the core object. Since roles exist only in a certain context, they are implemented as inner classes. OT/J, powerJava, EpsilonJ, and NextEJ are examples. Roles can be defined independently but can be imported to a context-like entity to be context-dependent. However, implementing the roles directly inside a context might be beneficial because it eases collaboration between roles and their associating context [LA15]. Even though introducing an additional context entity, these languages are still proved to achieve our modularity requirement.

R2: Dynamic Activation. In general, a core object or a player adapts its behavior based on roles that provide polymorphic method resolution. This selection is realized by the activation of a context with which those roles are associated. OT/J uses dynamic aspect weaving with the bytecode rewriting library to translate its *callin* and *callout* binding between the object and its roles to resolve dynamic method dispatch when a *team* is activated. SCROLL manages role instances in the form of an acyclic directed graph and takes advantage of the Scala language feature, *dynamic maker traits*, to translate statically polymorphic methods of the bound roles to the object. In powerJava, an object and its roles share the same interface. The object stores a reference to the role instances. Methods that are invoked from the object are cast to a given role type in an *institution*. For example, `((school.Student) person).study()` shows that a person object is cast to a `Student` role type in a `school` institution, and subsequently the `study` method is invoked. Similarly, EpsilonJ conceals a dynamic method dispatch with role reference and casting. For example, `person.(school.Student).study()` can be interpreted the same way as in powerJava. NextEJ is the successor of EpsilonJ by bringing back the type safety. Unfortunately, there is no real working compiler presented, but we assume it applies the same mechanism found in EpsilonJ. Although these approaches present different styles of dynamic activation, they support the requirement.

R3: Late Variants Adoption. The ROP languages, we discussed, are designed for anticipated adaptation where roles are given upfront. None of them has claimed its support for adopting new roles dynamically to satisfy the unanticipated adaptation. However, OT/J presented a case study of a flight bonus example to demonstrate the support of software evolution without invasion to the existing code, i.e., no source code is required, because OT/J works on the bytecode level. The adapting subsystem is implemented in its team with roles inside. OT/J manages to integrate the adapting roles with team activation to the existing code by taking advantage of the dynamic aspect weaving mechanism with the help of the ASM [BLC02] bytecode rewriting tool. The resulting system has new behavior. However,

we call this a software adaptation or evolution, not run-time adaptation, because we need to restart the legacy system. Similar to the dynamic aspect weaving strategy discussed in Section 3.3.4, we consider these ROPs to support this requirement only partially.

R4: Object-Level Tranquility. A few ROPs pay attention to system inconsistency resulting from the conflict of adapted roles. The conflict may happen either when the new role contradicts semantically to the one already deployed in the system or two roles need to coexist but adaptation demands for one of them to be removed. In this regard, only SCROLL offers constraint checking following the role constraints of Riehle and Gross [RG98]. Similar to the classification of Relational Roles [BGE07], none of ROPs in this category handles the tranquility concept although the constraints are supported.

Other Requirements. The ROP languages mentioned in this classification support neither *R5: Failure Handling* nor *R6: Continuous Deployment*. The evaluation is summarized in Table 3.8.

Table 3.8: Evaluation of Role-Oriented Programming (ROP).

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Static Roles[1]	■	□	□	□	□	□
Dynamic Roles[2]	■	■	□	□	□	□
Relational Roles[3]	■	⊞	□	□	□	□
Contextual Roles[4]	■	■	⊞	□	□	□

■: supported, ⊞: partially supported, □: not supported
[1] JavaStage [BA12]
[2] Chameleon [GØ03] and Rava [HNL+06]
[3] Rumer [BGE07]
[4] OT/J [Her05], powerJava [BBVDT06], EpsilonJ [TUI05], NextEJ [KT09] and SCROLL [LA15]

3.4 Dynamic Software Updates (DSU)

Running software systems are inevitably subject to change due to bug fixing or adding both functional and non-functional requirements. Traditional approaches involve restarting the system to patch new functionalities. Such restart presents unsatisfactory to end-users as they may redo all the unsaved jobs. Mission critical and highly available systems cannot tolerate the disruption and DSU is, therefore, needed.

DSU relates to run-time variability in the sense that DSU ideally allows programmers to transparently update every part of a program making their runtime adaptable to a new version of the updating system on-the-fly. DSU is a perfect candidate for unanticipated adaptation. Depending on the implementation patterns, update program code directly for several iterations may result in tangled code. This is not the case for systems developed using COP and ROP in which programmers must follow a given architecture or language semantics. These languages, if they support unanticipated adaptation, may allow the developers to update only the variable parts of the system while the static parts are kept intact.

The degree of a system's updatability in DSU differs from one solution to another, and it also depends on the target language. DSU normally addresses three main issues: a level of run-time objects to be updated, time taken for the update, and state consistency. First, the level of run-time objects to be updated is determined whether the solution allows only the method body of an object to be changed or supports advanced modification of the class schema, such as adding/removing methods/properties, adding/removing interfaces, modifying class hierarchy, and so forth. A comparison of the state-of-the-art on the class modification can be found in Gregersen and Bo [GJ11]. Second, the time taken for the update should be as short as possible. Approaches, like DCEVM [WWS10] and JVolve [SHM09], suspend the JVM execution and apply updates to all objects at once. These techniques seem to have low-disruptive updating but suspending the active JVM usually results in a longer disruption and is error-prone. Javelus [GCX+14] uses a lazy updating technique that updates objects only when the JVM reaches a safe point. With its specialized transformer, Javelus manages to achieve low disruptive updating. Third, the state consistency is the core requirement that all the DSU approaches fulfill. A detailed comparison of the DSU approaches can been found in Seifzadeh et al. [SAM13].

In summary, DSU is suitable for general software upgrades regardless of the software architecture. Therefore, it is possible to apply for unanticipated adaptation. The consistency of the program state during and after the update are usually guaranteed. However, programmers must develop their own way to deal with modularization and anticipated adaptation. Implementing a complete solution for DSU is very hard and tedious job as a simple bug may corrupt the program and consequently it is restarted. With respect to our requirements, DSU fulfills the *R3: Late Variants Adoption* and *R4: Object-Level Tranquility*.

3.5 Synthesis of the State of the Art

The existing variability approaches that mainly derived from programming language solutions are compared concisely with the requirements. Table 3.9 summarizes this comparison. The results show that none of the presented solutions captures all the requirements. Some of the solutions are developed merely for modularization such as inheritances, mixins, traits, subject composition, SPL, static aspect weaving, and static roles.

Apart from the group mentioned above, the remaining solutions satisfy the requirement *R2: Dynamic Activation*. For example, dynamic traits, role object pattern, subject dispatch, dynamic aspect weaving are solutions designed for anticipated adaptation. Approaches, which provide both anticipated adaptation and conflict resolution, are DSPL, COP, and ROP. The DSPL is an architectural solution that adapts software systems at a coarse-grained level as its variants are composed of many class dependencies. This kind of adaptation is suitable for systems that change infrequently. In contrast to DSPL, COP and ROP provide

Table 3.9: The state-of-the-art survey for run-time variability.

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Language Facilities						
Inheritance	■	□	□	□	□	□
Mixins [BC90, ALZ00, Moo86]	■	□	□	□	□	□
Traits [SDNB03]	■	□	□	□	□	□
Dynamic Traits [SD05]	■	■	□	□	□	□
Role Object Pattern [BRSW98]	■	■	□	□	□	□
Subject-Oriented Programming (SOP)						
Subject Composition [HO93, HOM95]	■	□	□	□	□	□
Subjective Dispatch [SU96]	■	■	□	□	□	□
Feature-Oriented Programming (FOP)						
Software Product Lines (SPLs)[1]	■	□	□	□	□	□
Dynamic Software Product Lines (DSPLs)[2]	■	■	⊞	□	□	□
Meta-Programming (MP)						
Iguana/J [RC02]	■	■	■	□	□	□
Reflex [TNCC03]	■	■	□	□	□	□
Geppetto [RDT08]	■	■	■	□	□	□
Aspect-Oriented Programming (AOP)						
Static Weaving Mechanisms [KHH+01]	■	□	□	□	□	□
Dynamic Weaving Mechanisms[3]	■	■	⊞	□	□	□
Context-Oriented Programming (COP)						
Local Activation[4]	■	■	□	□	□	□
Global Activation[5]	■	■	⊞	□	□	□
Global Activation with Conflict Resolution[6]	■	■	⊞	⊞	□	□
Role-Oriented Programming (ROP)						
Static Roles [BA12]	■	□	□	□	□	□
Dynamic Roles [GØ03, HNL+06]	■	■	□	□	□	□
Relational Roles [BGE07]	■	⊞	□	□	□	□
Contextual Roles[7]	■	■	⊞	□	□	□
Dynamic Software Updates [SAM13]	□	□	■	■	□	□

■: supported, ⊞: partially supported, □: not supported

[1] [Pre97, BSR04, ALRS05, LKKP06, Gri00, LSSP06, MO04, KAB07, AKLS07]

[2] [GS12, RSPA11, TCPB07, LK06, DMFM10, INPJ09]

[3] [BHMO04, NAR08, SCT03, VSV+05, Bon04, VBAM09, PDFS01, SMCS04, AGMO06, JZ10]

[4] [CH05, AHHM11, HCH08, Sch08, SP08]

[5] [AHM+10, SGP12b, LASH11, KAM15]

[6] [KAM11, GCM+10, Car13, GMC08, GMCC13, Bai12]

[7] [Her05, BBVDT06, TUI05, KT09, LA15]

language abstractions to enable anticipated adaptation at a fine-grained level. Therefore, such change can happen dynamically and frequently. However, both COP and ROP support partially unanticipated adaptation. Also, they do not adequately address inconsistency and run-time failures.

A few solutions in MP support unanticipated adaptation, for instance, Iguana/J [RC02] and Geppetto [RDT08]. However, these solutions are hardly portable because different host languages support various types of MOP. Moreover, context activation is missing, and developers have to address it separately.

DSU supports unanticipated adaptation at the level of the JVM or the operating systems. DSU considers neither reusability nor anticipated adaptation. Application developers must tackle on their own to bring anticipated adaptation to the system.

3.6 Chapter Summary

In this chapter, we reviewed systematically existing variability approaches derived from programming languages, which are good candidates for achieving run-time variability. We classified these approaches into three broad categories: Language facilities, Design patterns, and Language solutions. Language facilities represent native mechanisms of the respective languages and are designed for code reuse. A design pattern closely related to ours is the role object pattern which inspired the development of many ROP languages. Language solutions are our main focus and we discussed them primarily in the area of SOP, FOP, MP, AOP, COP, and ROP. Due to the great volume of research in each language domain, we classified the work in each domain based on their similarity to ease the discussion. We assessed these groups against the requirements we set in Chapter 2. DSU was also discussed as it crosscuts the requirements of unanticipated adaptation and state consistency. Finally, we synthesized these evaluations into the list depicted in Table 3.9 for comparison.

From this survey list, we can draw the conclusion that DSPL, COP, and ROP are the most promising approaches. Although DSPL seems to be a viable candidate for run-time variability, its adaptation is rigid and structural. That is because DSPL lacks contextual and scoping support for adaptation at a fine-grained level. COP and ROP integrate powerful dynamic activation into an integral mechanism for run-time adaptation. However, they offer limited functionalities to cover unanticipated adaptation and manage the consequences of such adaptation. In the next chapter, we will discuss our approach that addresses these shortcomings.

Chapter 4

LyRT: Role-based Runtime, Concept and Design

The goals of this dissertation are twofold. On the one hand, we investigate the run-time variability features based on the concept of roles to steer the coexistence of anticipated and unanticipated adaptation support. On the other hand, we further examine the implications or consequences of adaptation on run-time systems in which those implications affect the system consistency and stability. To meet these objectives, in Chapter 2 we derived requirements that language engineering techniques should fulfill in order to achieve run-time variability. In Chapter 3, we conducted a systematic review of the state of the art and pointed out that existing variability approaches fail to fully cover these requirements of run-time variability.

In this chapter, we illustrate our approach, called LyRT, a role-based runtime, which is proposed to circumvent the shortcomings of existing approaches and to meet the requirements of run-time variability. First, we will give an overview of the LyRT run-time architecture followed by a discussion of its individual components in Section 4.2. We highlight the primary concept of the dynamic instance binding mechanism for anticipated and unanticipated adaptation in Section 4.3, the tranquility mechanism for consistent adaptation in Section 4.4, and the rollback recovery mechanism for handling software failures due to role composition caused by the adaptation in Section 4.5.

To facilitate the discussion of our approach, two examples are used. An example of a tax management system will be illustrated in Section 4.3.2 to depict the supporting features of the runtime in terms of its role-playing relation. Another example is that of a chat server application which has already been described in Chapter 2 to demonstrate the dynamic behavior, inconsistency and the need for the rollback recovery mechanism. This conceptual chapter is presented as generic as possible without sticking to a host language. However, some code snippets are introduced based on a Java syntax to highlight the main abstractions.

4.1 Terminology

Before introducing LyRT, the following terminology is defined for its concepts and design.

4.1.1 Kinds of Objects

LyRT adopts the notion of CROM [KBGA15], discussed in Section 2, but it mainly focuses on the instance level.

Definition 4.1. ***(Core or Core Object)*** *A core is an instance derived from a typical class in object-oriented languages (i.e., natural type in CROM). Cores dynamically bind to or unbind from several role instances for achieving dynamic behavior.*

Definition 4.2. ***(Role)*** *A role instance is a typical object whose properties and methods substitute to a core object with a method dispatch when bound. A role instance is created in the binding process as it cannot stand on its own due to non-rigid property. Role instances can be bound and are active if the compartment instance with which those roles are associated is active. Like a core, a role instance can bind another role instance.*

Definition 4.3. ***(Compartment)*** *A compartment instance is a typical object comprising properties and methods. A compartment is used to objectify a collaboration of participating roles and a fixed scope of bound roles. Although the compartment is a container of roles in CROM, its instance is distinct from role instances. Like a core, a compartment instance can bind another role instance in another compartment instance.*

Definition 4.4. ***(Player)*** *An instance of core, role and compartment can be a player due to its ability to play other roles.*

4.1.2 Relations

Core, role, and compartment relate to each other with certain kind of relations.

Definition 4.5. ***(Play-Relation or Bind-Relation)*** *A play-relation is defined by any kind of objects denoting as core, role and compartment instance binding to a role within a compartment instance.*

Definition 4.6. ***(Deep-Play-Relation)*** *A deep-play-relation refers to a stack of play-relations between a core to a role and the bound role to another role in the form of role-playing-role.*

Definition 4.7. ***(Link-Relation)*** *A link-relation is defined as a relationship between a pair of roles within a compartment instance in which those roles are bound to the respective cores.*

4.1.3 Activation

To realize dynamic behavior of core objects, a compartment instance of the prescribed bound roles needs to be activated.

Definition 4.8. ***(Application Configuration)*** *An application configuration is described as a set of compartments and a set of roles with bindings to cores. Adaptation is a process to transfer from one application configuration to another by means of compartment or adaptation block activation.*

Definition 4.9. ***(Compartment Activation)*** *In order to enable the dynamic behavior of cores, a compartment instance containing bound roles has to be activated as a result of adaptation. Once activated, behavior of the cores is lifted or dispatched to the bound roles.*

Definition 4.10. ***(Adaptation Block)*** *An adaptation block is a construct to reconfigure the role bindings of a set of cores within an active compartment instance to adapt the behavior of the particular cores to the newly bound roles without activating a new compartment instance.*

4.1.4 Mechanism to Navigate Instances for Method Dispatch

Adopted from ObjectTeams/Java [Her05], lifting and lowering are the main mechanisms to navigate the target instance for dynamic method dispatching.

Definition 4.11. ***(Lifting)*** *Multiple role instances can be bound to a core by stacking up together in the form of a deep-play-relation within a compartment instance. The lifting mechanism navigates from a core to the last role instance in the deep-play-relation to invoke a method sent from a core.*

Definition 4.12. ***(Lowering)*** *The lowering mechanism is the inverse of the lifting mechanism, i.e., navigating from a role instance to its core. Lowering can be cascaded down through a deep-play-relation to create a partial method by visiting each role instance in the relation. Lowering can be lowered directly to a core without visiting each role instance if needed.*

Definition 4.13. ***(Collaboration)*** *The collaboration mechanism specifies the selection of a compartment instance and role instance of a given link-relation for method invocation. Since roles are not defined directly inside a compartment, the collaboration mechanism allows a role instance to access the compartment's properties and methods. Additionally, a given pair of two role types has a relationship, a role instance bound to a core can invoke all role instances of a given type bound to different cores within an active compartment instance.*

4.1.5 Consistency Mechanism

A core should adapt its behavior in a consistent manner.

Definition 4.14. ***(Consistency Block)*** *A consistency block is a construct specified by a developer to group multiple method invocations of different cores in which those invocations require the same consistent behavior from the start to the end of the block.*

Definition 4.15. ***(Tranquil State)*** *A tranquil state of a program is defined as a region of executing code located outside the consistency block. A core executing in a tranquil state is allowed to adapt its behavior; otherwise, it has to wait until a tranquil state is reached (i.e., the consistency block expires).*

4.1.6 Rollback Recovery Mechanism

A core object changes its behavior with regard to the bound roles. If the roles are not properly tested. Bugs in those roles are brought to the runtime during adaptation. An execution of a method in the core propagates to the bound roles in which the bugs turn out to be failures. Consequently, the runtime may crash.

Definition 4.16. ***(Failure)*** *A failure or a run-time failure refers to a method execution of a core which might cause a runtime to crash. The failure is caused by a software bug which appears in roles. When a defective role is bound to a core, its behavior is lifted for invocation. Therefore, common bugs in that defective role, such as a DivideByZero, can crash the runtime.*

4.2 An Architecture of Role-based Run-time Variability

Role-based applications normally consist of two main parts—*static* and *dynamic* ones. Cores or core objects are considered to be static parts while roles are modeled as dynamic parts. As the name suggests, the static parts remain fixed within the life cycle of an application. The roles are variable and can be exchanged in response to the need for the adaptation. The play-relation of players and roles constitutes the binding process to activate dynamic behaviors of the players as roles encapsulate the polymorphic behaviors of the corresponding players. This abstraction aligns with the notion of run-time variability whose variants, dynamic parts, are composed to the system's static parts by means of a binding process to achieve variability.

LyRT is a role-based run-time engine with the goal to comprehensively support run-time variability and its implications for system consistency and stability. The proposed approach relies on the concept of roles as specified in the CROM [KLG+14, KBGA15], a unified model of compartments, roles and objects (cores) which captures the three natures of roles discussed in Chapter 2. That is behavioral, relational and context-dependent nature. LyRT aims to achieve the behavioral and context-dependent nature of roles but only partially fulfills the relational nature due to the lack of constraint support.

4.2.1 LyRT Architecture

The architecture of LyRT is depicted in Figure 4.1. Its main parts are the Tranquility Controller (TC), the Role Execution Engine (REE), and the Rollback Recovery Controller (RRC). The process of adaptation is either initiated by a context change that triggers an anticipated adaptation or by developers that update or add roles on-the-fly to address new or changed requirements with unanticipated adaptation.

The Tranquility Controller checks the requested adaptation to determine when the runtime should perform the changes—promptly or restrainedly to avoid inconsistency. The Role Execution Engine is the execution environment of LyRT in which anticipated and unanticipated adaptation is performed. Compartments, roles, and objects located inside REE are the object representations of our run-time model. The Rollback Recovery Controller is an additional component introduced to detect and handle software failures due to dynamic role composition as a result of the adaptation. All three main parts will be explained in detail in the following sections.

4.2.2 Context Discovery

Context is the main source to trigger adaptation. Although compartments are introduced to achieve context-dependent property by means of activation and deactivation, the compartment itself is not a context. Compartments objectify the collaboration of roles played by different cores [KLG+14]. Such objectification expresses not only the activation of the dynamic behavior of those core objects but also the activation of a relationship or collaboration of the participating roles which are played by different cores. This technique facilitates the adaptation resulting from context changes. We adopt the notion of context which is defined as "any computationally accessible information" [HCN08]. This definition is broad and open but it is confirmed to be valid in developing context-dependent applications [SGP12a].

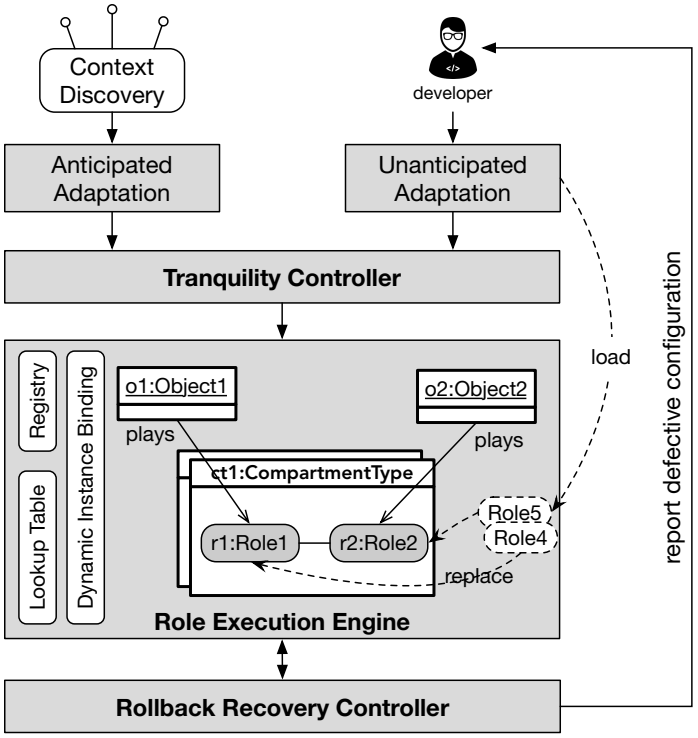


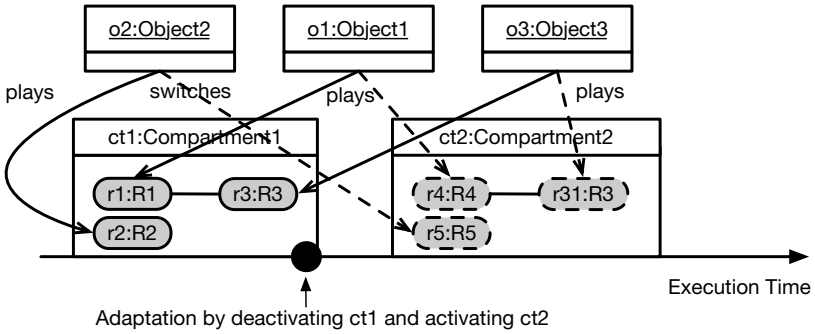
Figure 4.1: Overview of LyRT architecture.

Information to derive context can be obtained from outside, i.e., the environment or the users, but it can also be originated internally from the value of sensors attached to the system.

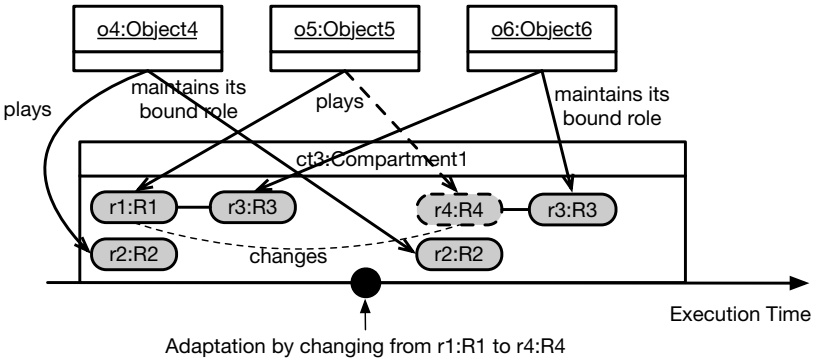
Similar to many COP languages [AHH+09], the context in LyRT is abstract but it allows different parts of the system, e.g., different threads or different instances of the same object type, to live in separate contexts. Therefore, the resulting adaptation is applied independently to each application part. This technique is more flexible than the approaches adopted by Ambience [GMC08] and Subjective-C [GCM+10] which provide a unique global context shared by all application parts. From a conceptual point of view, having a single global context brings a more intuitive model for systems such as mobile applications, but allowing multiple contexts to exist independently is more flexible in practice [SGP12a]. With respect to the chat server example, described in Chapter 2, the context in LyRT allows each thread, which handles a client session, to adapt differently to the specific conditions of the client engaging in service. Additionally, it is possible to adopt the unique global context. A compartment in LyRT is a normal object which can be declared as a singleton class to steer the activation globally based on certain context change, i.e., an event processing. Anyway, we assume developers can derive the context to trigger the adaptation. LyRT offers a way to (de-)activate dynamic behavior, but it provides neither context model nor context reasoning, e.g., a context that is derived from a machine learning technique.

4.2.3 Adaptation

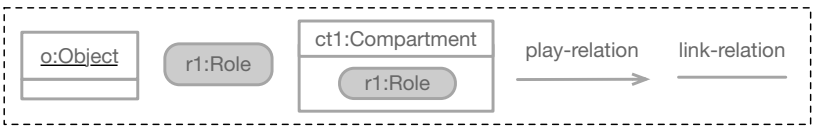
According to the role concept [KLG+14] and CROM [KBGA15], the adaptation of objects is a result of role bindings in conjunction with an activation of a compartment with which the bound roles are associated. Since dynamic behavior is encapsulated in roles, an application configuration can, for instance, be changed by switching from one compartment to another or by performing a set of role binding and unbinding operations within an active compartment. Figure 4.2 depicts the two different kinds of activation. To be consistent, we use the notation as depicted in the legend box (Figure 4.2c) to represent the core object, the role, the compartment, and their relations throughout this dissertation. While this section mainly discusses the abstraction of anticipated adaptation, unanticipated adaptation will be explained in Section 4.3.8.



(a) Adaptation as an activation of a compartment.



(b) Adaptation as a change of roles in an active compartment.



(c) Legend.

Figure 4.2: Activation strategies to achieve dynamic behavior.

4.2.3.1 Adaptation as an Activation of a Compartment

Figure 4.2a visualizes the concept of an activation of a compartment by allowing objects `o1`, `o2`, and `o3` to have initial behaviors based on instances of the bound roles `r1`, `r2`, `r3` respectively in the compartment instance `ct1` which is currently active. During the execution, the adaptation happens resulting in following:

```
ct1.deactivate();
ct2.activate();
```

These objects eventually adapt new behavior encapsulated in roles specified in the new instance `ct2` of another compartment. The adaptation is straightforward, and it allows a large structure of application to be changed because compartments do not only capture behavioral change of certain objects in isolation but also their interactions in terms of role relationship (i.e., the link-relation between `r1` and `r3`).

Snippet 4.1: Adaptation by switching compartments.

```
1  ct1.initBinding(){  //initialize the binding
2    o1.bind(R1.class); //describes the binding/playing process
3    o2.bind(R2.class);
4    o3.bind(R3.class);
5  }
6
7  ct2.initBinding(){  //initialize the binding
8    o1.bind(R4.class); //describes the binding/playing process
9    o2.bind(R5.class);
10   o3.bind(R3.class);
11 }
12
13 ct1.activate(); //compartment is activated
14
15 //... executing
16
17 //Adaptation occurs
18 ct1.deactivate(); //compartment is deactivated
19 ct2.activate();   //a new compartment is activated
20
21 //... executing with new behavior
```

Snippet 4.1 shows what the program to describe the play-relation and adaptation in LyRT with respect to Figure 4.2a should look like. LyRT is developed to support the role binding and activating happening totally at run time. Once core objects (e.g., `o1`, `o2`, and `o3`) and a compartment (e.g., `ct1` and `ct2`) are initialized, the role binding process can be constructed with the `initBinding`-block construct (Lines 1-5). After the compartment is activated in Line 13, those core objects adapt their behavior to roles `R1`, `R2`, and `R3` respectively. Afterwards, adaptation triggers the deactivation of the current compartment and activates a new one (Lines 18-19). After the adaptation, the core objects adapt their behavior according to the bound role instances in the compartment `ct2`. The bound roles are not destroyed when the compartment is deactivated so that when the compartment is reactivated, the bound roles become active immediately without reinitialization.

4.2.3.2 Adaptation as a Change of Roles in an Active Compartment

If there is a small behavioral change of a single object (e.g., `o1` to unbind from `r1` and to bind to `r4` in the same `ct1` compartment), using CROM results in two separate compartment types. One compartment has the same representation of `Compartment1` and another has a slight variation that changes from `r1:R1` to `r4:R4`. The problem of having two compartment types for these configurations is that the state of unchanged roles (i.e., `r2`, `r3`) cannot be preserved at run time in the transition from one compartment to another. This is because

switching between compartments requires a new instantiation of roles although they are derived from the same type.

LyRT extends the capability of supporting adaptation by not limiting to the (de-)activation of compartments but also allowing roles to be changed inside an active compartment. In Figure 4.2b, the adaptation needs `o5` to change its behavior by unbinding from `r1:R1` and binding to `r4:R4` role instance whereas the rest of role-play relations remains untouched. This technique seems to contradict to the model specified in CROM in which the number of roles is fixed in a given compartment. However, to be CROM compliant, we assume that in the model all the role types (i.e., `R1`, `R2`, `R3`, `R4`) are given in the compartment. We further argue that context triggering the adaptation can be attached to both compartment and role inside a compartment. Supporting these types of adaptation provides the runtime with different degrees of variability—from a fine-grained level of individual role instance to a coarse-grained level of the compartment. Supporting unanticipated adaptation, which will be explained in Section 4.3.8, falls into this category.

In order to change a new role in an already active compartment, an `AdaptationBlock` construct is used to provide the necessary role operations such as binding, unbinding and transferring in response to adaptation. Snippet 4.2 exhibits an implementation based on the adaptation process shown in Figure 4.2b. The adaptation takes place in Lines 12-15 mentioning that the core object `o5` first unbinds from role `R1` and then binds to a new role `R4`.

Snippet 4.2: Using `AdaptationBlock` to change roles in an active compartment.

```
1  ct3.initBinding(){  //initialize the binding
2    o4.bind(R2.class);
3    o5.bind(R1.class);
4    o6.bind(R3.class);
5  }
6
7  ct3.activate();
8
9  //... executing
10
11 //Adaptation occurs
12 AdaptationBlock ac = new AdaptationBlock(){
13   o5.unbind(R1.class);
14   o5.bind(R4.class);
15 }
16
17 //... executing with new behavior
```

4.3 Role Execution Engine

According to Figure 4.1, the adaptation goes to the tranquility controller to check whether the adaptation should be performed now or later. The consistency mechanism built in the tranquility controller relies on the dynamic instance binding mechanism which is a core component of the role execution engine. Therefore, the role execution engine is discussed first in order to understand how the adaptation works internally.

4.3.1 Run-time Model

When dealing with complexity, a model is developed to abstract the problem. Models are typically used at design time to capture the blueprint of software systems aiming at improving the understandability among developers. Useful information conveyed by the model often gets lost at run time as the model elements are transformed into run-time artifacts which no longer maintain their originality [ODPR08]. Engineering adaptive software systems requires the knowledge of run-time model to easily monitor and change their behavior while still providing a complete overview of the running systems. We follow this principle to design the run-time model of LyRT.

Although CROM [KBGA15] inspired us to design LyRT, it is a model which can have different implications on designing a run-time model. Compartment Role Object Instance (CROI) [KBGA15] is an instance model, formally specified, representing CROM[1] at run time but it does not provide a computation model (e.g., representation of objects in a runtime and method dispatching). Furthermore, the current formalized model does not support the *deep-play*-relation (role-playing-role) which is insufficient for building the *partial method*, described in Section 4.3.5. We use pure object-oriented technologies to construct the representation of our run-time computation model similar to CROI. Figure 4.3 represents our run-time model. LyRT's run-time model contains three main types of instances derived from a compartment, role and natural type[2] (i.e., to instantiate core object) respectively. A *play*-relation fulfills a set of role instances played by cores in a compartment. An instance of a natural type (i.e., core object) may play multiple instances of roles from different types. Similarly, an instance of role which is played by an instance of natural type may also play other instances of different role types (i.e., *deep-play*-relation). Like an instance of the natural type, a compartment instance may play multiple instances of roles in another compartment. A *link*-relation establishes a relationship between a pair of roles participating in the same compartment.

In short, the features of CROM that we adopt for LyRT are object-plays-role, role-plays-role, compartment-plays-role, and partial role-relationship[3]. We cannot fulfill all the features specified in CROM. For example, role constraints and constraints on the relationship are not directly supported because normally constraints are enforced at the type level whereas we focus on the instance level. Instead, LyRT's run-time model provides the *deep-play*-relation to support partial method cascading over the deep roles. The supporting features are sufficient for run-time adaptation because we can dynamically change the *play*-relation within a compartment. Activating the compartment reifies those *play*-relations for object adaptation.

To achieve the flexible run-time model that can be changed at any point in time during execution, we arrange the instances of the distinct types to be decoupled from each other and stored them in an instance pool as shown in Figure 4.4. We rely on the *dynamic instance binding mechanism*, which will be described later in Section 4.3.3, to dynamically bind and unbind those instances to reconstruct the run-time model as depicted in Figure 4.3.

---

[1] The discussion of the distinction between compartment, role and object is given in Chapter 2

[2] A core object is instantiated from a natural type

[3] LyRT does not support relationship type, but it allows the expression of method invocation reflecting the collaboration which will be discussed in Section 4.3.5

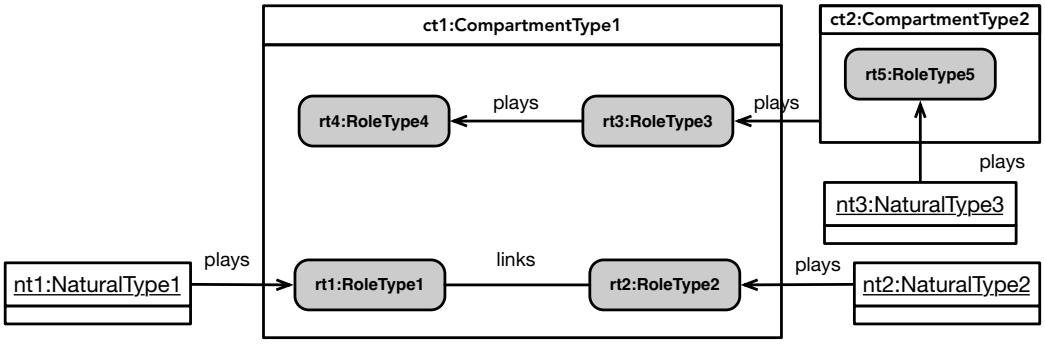


Figure 4.3: LyRT's run-time model.

The binding relations of those instances are stored in the lookup table, which will also be presented later, to be manipulated for changing the run-time model to achieve adaptation. Note that from the run-time model perspective roles are located inside a compartment, but they are physically independent of each other to maintain the flexibility of adaptation by just replacing roles at run time.

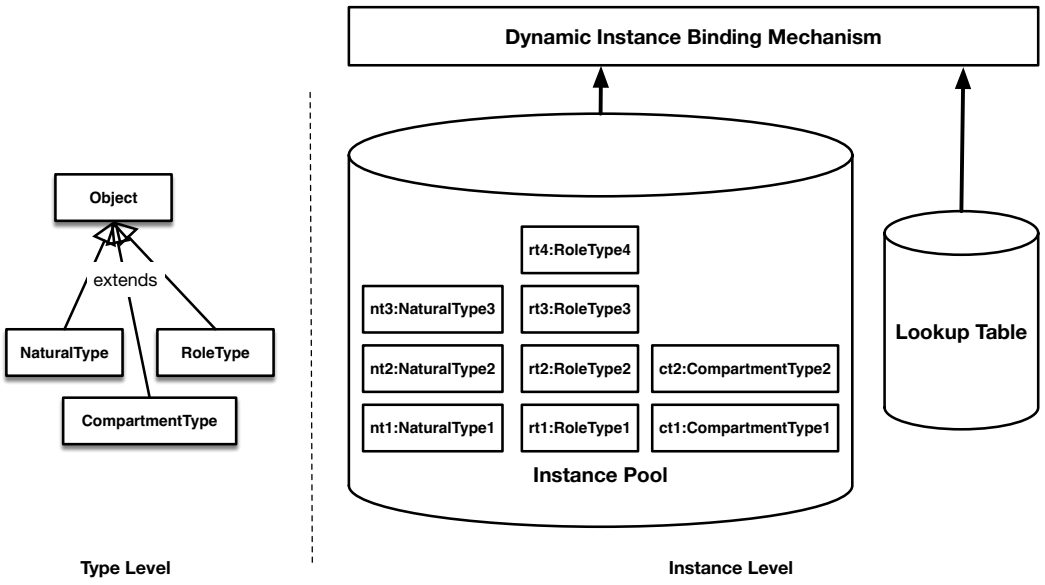


Figure 4.4: Arrangement of run-time objects to represent the run-time model.

4.3.2 An Example of a Tax Management System

Figure 4.5 shows an example of a Tax Management System which is represented as a run-time model to highlight the role-playing features supported in LyRT. The detailed functionality of this example will be explained as a case study in Section 6.1.2. The basic functionality is to manage a company comprising different roles to be played by persons. The example further presents how taxes are collected from the company and a freelancer which is a role expected to be played by a person. Based on regulation, the company and the freelancer pay taxes at a different rate.

`Person` is a natural type that has several instances (e.g., `ely, bob, alice, ana`); each plays

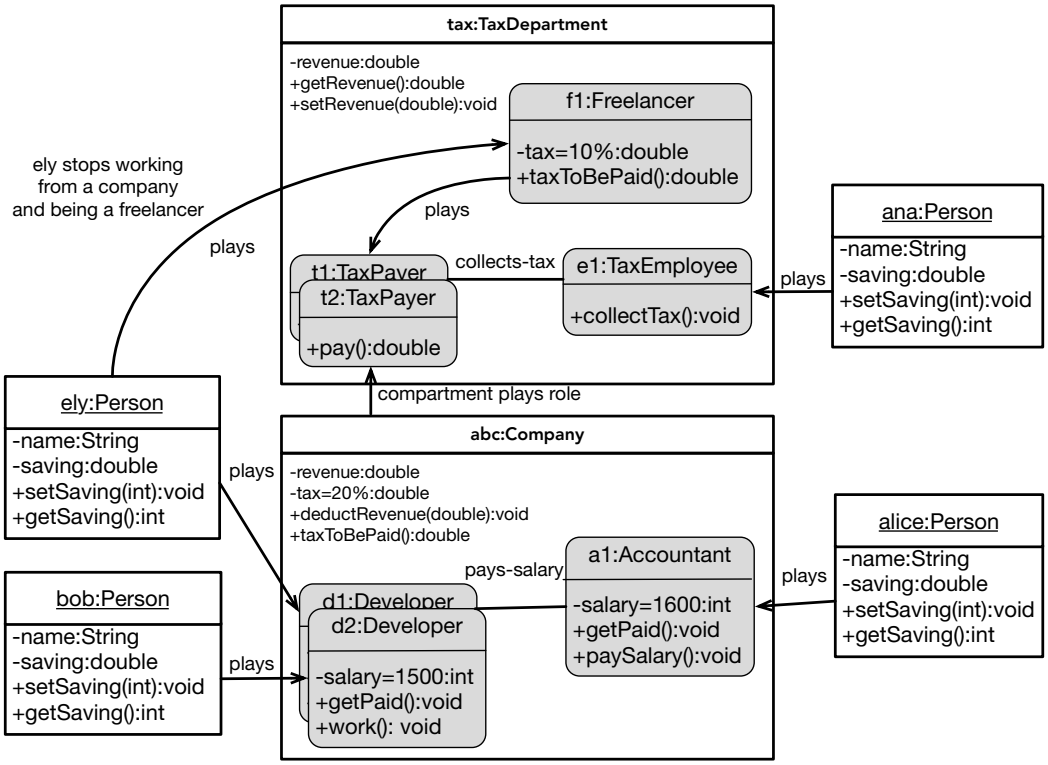


Figure 4.5: An example of the tax management system.

different roles. While `ely` and `bob` play the same `Developer` role, they bind to different instances of the `Developer` role. The behavior of each person instance is adapted according to the played role.

Besides scoping roles, a compartment is like a normal object consisting of state and behavior and may play roles. A compartment `abc` instantiated from a `Company` in which the `abc` company plays a `TaxPayer` role located in a `TaxDepartment` compartment in order to pay taxes.

The play-relation is not restricted between objects and roles, but a role is also allowed to play another role. A person *ely* stops working for `abc` company and becomes a `Freelancer` who in turn needs to play a `TaxPayer` role in to order pay taxes in the scope of `TaxDepartment` compartment.

Roles which are played by different objects can interact with each other in terms of relationship expressed as a method call, affecting all cores playing the role. For instance, the `Accountant` role has a *pays-salary* link-relation with `Developer`. To facilitate this link-relation, `alice` who plays `Accountant` role calls the `paySalary()` method to pay a monthly salary to all person instances playing the `Developer` and `Accountant` roles. Besides, roles in a compartment can access attributes and methods of that compartment and vice versa. For example, when the `Accountant` role calls `paySalary()`, this method withdraws money from the `revenue` attribute of the `abc` company, to distribute it to the person instances playing `Developer` and `Accountant` roles. Similarly, a *collects-tax* link-relation between `TaxEmployee` and `TaxPayer` in the `TaxDepartment` compartment can be expressed in the invocation of a `collectTax` method of the `TaxEmployee` role. We call this feature a *collaboration*.

4.3.3 Dynamic Instance Binding Mechanism

Code weaving is a mechanism to bind the implementation of two or more objects [KLM+97]. This mechanism is used in different programming paradigms geared towards dynamic object (re-)composition, such as AspectJ [KHH+01] in AOP [KLM+97], ContextJ [AHHM11] in COP [HCN08], and ObjectTeams/Java [Her05] in ROP [KLG+14].

Normally, objects are woven either at the source code or bytecode level. Weaving at source code level usually happens at compile time by means of a pre-processor and a modified compiler. This mechanism does not affect the performance of the system. However, it does not allow to split woven code, or re-weave new behavior at run time because when woven, two objects become one indivisible system entity. Furthermore, it is not possible to weave two system classes or legacy systems for which their source code is not available. To solve this problem, bytecode weaving mechanisms are introduced hoping that bytecode rewriting allows to split and merge objects at some points during the system execution. Bytecode weaving may happen at post-compile time, load time, or run time. Nevertheless, like source code weaving, bytecode weaving at post-compile and load time does not support (re-)weaving because it generates a system snapshot. Run-time bytecode weaving provides code reweaving but may result in a performance overhead. System crashes may occur when reweaving as this mechanism does not assure the validity of the woven code. Even though weaving may happen at run time, bytecode weaving is usually applied to object types. Supporting bytecode weaving for instance level is very limited as this may require to destroy the instance and re-initialize it after weaving. Therefore, this process must wait until the instance is idle and its state is copied to map it back after the newly woven instance has been instantiated. Thus, this technique is not suitable for highly adapted systems which require instance-level adaptation.

We propose a dynamic instance binding mechanism in which, rather than weaving at the source code or bytecode level, we dynamically bind two or more role instances to object instances by constructing a transient relationship between them at run time. A binding relation shows the run-time association between binding instances. This mechanism utilizes the concept of *sharing* and *dispatching* which are two fundamental concepts of object-oriented languages to manage the flexibility of an object that can change its behavior at run time [BD96, SLU88]. Sharing is used to extract common functionalities, i.e., behaviors, from other objects without the need for redundant implementation of those functionalities. Dispatching then realizes the execution operations of the shared functionaries among the sharing objects. Inheritance is a standard mechanism known for static sharing as it cannot change the class hierarchy during run time. In contrast, delegation is considered as a dynamic sharing technique which has been used for supporting the late binding of an object to dynamically invoke a method to achieve variability at the instance level. To be flexible, the dynamic instance binding mechanism, therefore, considers the implementation of the concept of the dynamic sharing and dispatching based on role-playing-relation to achieve context-dependent variability.

In the playing-relation, once a core object binds to a role instance, their properties and methods are shared among each other. A lookup table is used to store the binding relation at run time. Note that bound instances remain completely decoupled from each other while appearing as a single object. Programmers interact with core objects; however, their method invocations are dispatched to the bound roles based on delegation.

For example, in the tax management system in Figure 4.5, the `ely:Person` instance binds to the `d1:Developer` instance by saying that the `Person` plays a `Developer` role. The `ely:Person` instance has access to all state and behavior of the `d1:Developer` through delegation. In order to do this, a proper method dispatching mechanism is needed (see Section 4.3.5). In our system, unbinding object instances simply removes the binding relation from the lookup table and destroys unused instances. For example, when removing the play-relation between the `Person` and the `Developer` instances, the `Person` no longer has access to the state and behavior of the `Developer`. The rebinding operation follows the same process as the initial binding. Our mechanism enables dynamic behavioral variations by merging and splitting different object instances. A split role instance can also be attached to another core while preserving all its states.

In order to deal with unanticipated behavior at run time, the new behavioral classes must be defined and compiled. Whenever the bytecode classes are loaded into the run-time environment, they immediately become available to be bound to other existing instances. Manipulating data in the lookup table is required to construct the binding relation. A further discussion is explained in Section 4.3.8.

In summary, our dynamic instance binding concept is suitable for highly dynamic run-time systems that require continuous modification to cope with an ever-changing environment. As a result, run-time variability can be achieved by allowing anticipated and unanticipated adaptation at the instance level.

4.3.4 Lookup Table

The dynamic instance binding mechanism heavily relies on the lookup table to maintain the run-time model for computation. As mentioned earlier, compartment, object, and role are normal object-oriented classes that do not necessarily relate to each other at type level. At instance level, they can be dynamically bound according to the role-playing model. The relationship in the lookup table is extended to capture the different types of instance bindings.

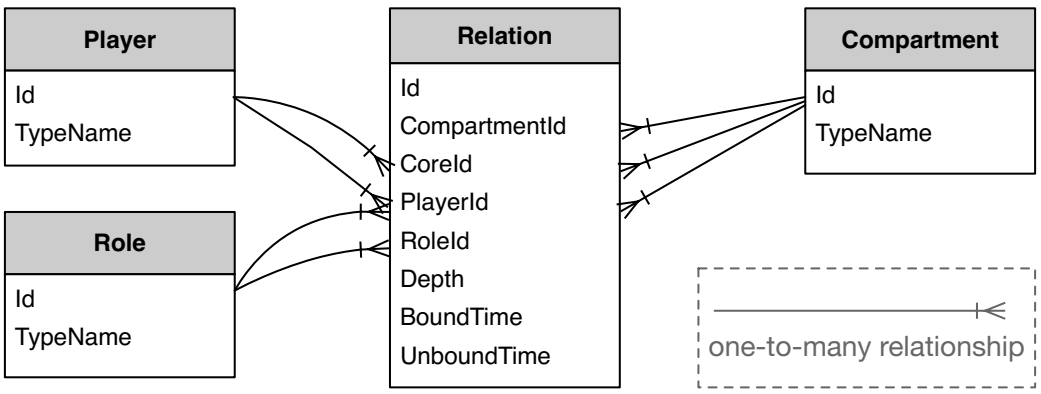


Figure 4.6: Structure of the lookup table.

The structure of the lookup table is conceptually expressed in the form of a relational database schema as shown in Figure 4.6. Each instance type is stored in the respective table of `Player`, `Role`, and `Compartment` according to their type. Those tables contain the real identity of the instances and their type name, e.g., `Person` or `Developer`. We use *hashcode* as an internal identity to identify the object, and this code is stored in the `Id` attribute of

those tables. The `Player`, `Role` and `Compartment` tables have a one-to-many relationship to the `Relation` table.

The `Relation` table holds the relationship of those binding instances. The attributes in the `Relation` table are sufficient to express the *play*-relation and to be used for the dynamic method dispatch. The `Id` is an auto-generated identity of each relation. The `CompartmentId` denotes the identity of a compartment instance. The `CoreId` refers to the identity of a core object derived from a natural type, e.g., the identity of `bob`. The `PlayerId` is the identity of the player which can be an identity of a compartment, role or core object. The `RoleId`, as the name suggests, is the role's identity to be bound to a player. The `Depth` represents the depth from a core object instance that occurs in a deep-play-relation. In the example in Figure 4.5, `ely` plays the `f1:Freelancer` role and the `f1:Freelancer` plays the `t1:TaxPayer` role. In this case, the `t1:TaxPayer` has a depth of 2 level while the `f1:Freelancer` has a depth of 1 level with respect to the distance from `ely`.

The `BoundTime` and `UnboundTime` in the `Relation` table present the time in which a role is bound to and unbound from a core object respectively. These two attributes are used for handling consistency which will be explained in Section 4.4.

Table 4.1 shows the sample data of the run-time model depicted in Figure 4.5. Instead of using an integer for the instance identity, we use plain text easing the identity tracking. The first and second rows in the table represent the type of a play-relation, between `ely` and `d1:Developer`, and `bob` and `d2:Developer` instances inside the `abc` compartment instance. In contrast to `bob`, `alice`, which is also a `Person` instance, plays another role, i.e., `a1:Accountant`, as shown in Row 3. Rows 4 and 5 demonstrate the deep-play-relation in which `ely` has the `f1:Freelancer` role and this role binds to the `t1:TaxPayer` role. These two relations virtually weave the `f1:Freelancer` and `t1:TaxPayer` roles to the `ely` instance. A compartment may play a role like any core object. For example, the `abc` compartment plays the `t2:TaxPayer` role inside the `tax` compartment in Row 6.

Table 4.1: Sample data of the tax management system in the relation table.

Id	CompartmentId	CoreId	PlayerId	RoleId	Depth
1	abc:Company	ely:Person	ely:Person	d1:Developer	1
2	abc:Company	bob:Person	bob:Person	d2:Developer	1
3	abc:Company	alice:Person	alice:Person	a1:Accountant	1
4	tax:TaxDepartment	ely:Person	ely:Person	f1:Freelancer	1
5	tax:TaxDepartment	ely:Person	f1:Freelancer	t1:TaxPayer	2
6	tax:TaxDepartment	abc:Company	abc:Company	t2:TaxPayer	1

The design of the lookup table should be simultaneously accessible and thread-safe. This is because LyRT supports multiple contexts which activate the adaptation of different program parts, i.e., different threads and different instances, simultaneously and independently. While from a conceptual point of view the structure of the lookup table is valid, it may face several problems such as state inconsistency of the lookup table, a single point of failures and performance bottlenecks. We leave these problems to the implementer of LyRT to resolve the particular issues.

An implementation suggestion is that the lookup table should be globally locked to avoid

any race condition issues of the concurrent access. The lookup table is used to manage the internal representation of the run-time system while it should not impose too much on system performance. According to role-based applications, the lookup table is modified in two stages—*object creation*, and *role binding/unbinding* (i.e., adaptation). The object creation involves in instantiating of core object and compartment in which their instances are placed into the instance pool. Roles are instantiated during the binding process in which their binding relations in the lookup table are also created. Additionally, once the role is bound, its methods are cached locally in the associating compartment instance for the dynamic method dispatch, discussed in Section 4.3.5. This technique ensures that dynamic method invocations do not require to navigate for a target role instance through an expensive query in the lookup table.

4.3.5 Instance-based Method Dispatch

This section describes the composition and method dispatch in LyRT. The execution model in LyRT is player-centric. A method invocation starts from a core object to roles. In order to demonstrate the navigation between these two object types, we borrow the terminology of *lifting* and *lowering* from ObjectTeams/Java [HHM04]. While lifting navigates from a core object to roles which are composed to the core object itself, lowering traverses from a role to the core object to achieve partial method behaviors. Besides, *collaboration functions* are necessary for method invocation of a role participating in a relationship and method invocation of the associated compartment instance. Figure 4.7 shows a visualization of these functions with respect to the run-time model. While LyRT allows programmers to access directly the methods of the relevant instances, accessibility to those instances' states can be done indirectly by wrapping the state in `get` and `set` methods. Additionally, the method translation disregards the polymorphism of class inheritance. For example, a role `R1` is a subclass of a role `R2`, and a core object `O` plays the role `R2`. This play-relation is not inherited to `R1`.

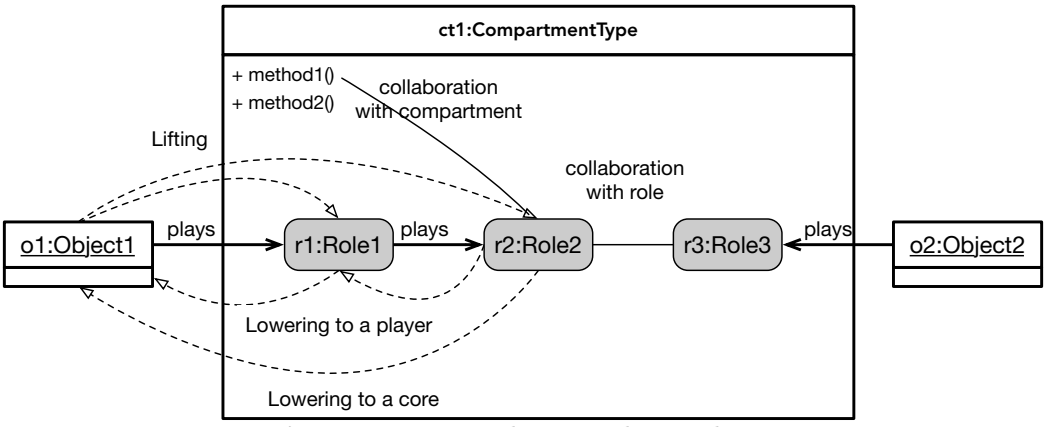


Figure 4.7: Translation polymorphism.

4.3.5.1 Lifting Functions

A core has access to the state and behavior of all roles bound to it. Whenever there is a method call from a core object, a method dispatch mechanism lifts the method by looking for the method to call among all bound roles in the `Relation` table and invokes it within an

active compartment. If the lookup method is not found in the bound roles, the dispatcher will look into the core object itself, and raise a run-time error if no method is matched. This process applies to binding relations in which the core and its roles have no duplicated method signatures (i.e., no method polymorphism). In other words, the core object and its bound roles become one compound object in which all methods are accessible from the core object as they would have been implemented directly in the core object.

In the example of the tax management system illustrated in Figure 4.8, `ely` instance can call the `getPaid` method of the bound `Developer` role by `ely.invoke("getPaid")`. Similarly, `alice` instance can invoke the `paySalary` method due to its binding to the `Accountant` role.

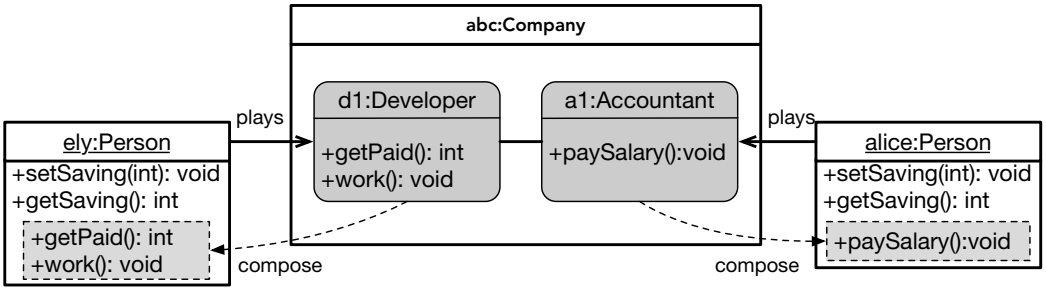


Figure 4.8: Composition using lift function.

In case of method polymorphism (i.e., methods sharing the same signature), the method invocation is always resolved first for the method implemented in roles. The priority of the invocation is given to a role with the *deepest relation* with respect to the core object. For example, in Figure 4.7, if `Object1`, `Role1`, and `Role2` share the same `m` method, the `m` method of the instance of `Role2` is invoked. Algorithm 4.1 shows how lifting can select an appropriate role for invocation when a method is called from a core within an active compartment.

Algorithm 4.1: Lifting algorithm.

Input : core, method
Output : object

```
1  compartment := GetActiveCompartment(CurrentThreadId)
2  if compartment is not null then
3      if HasBoundRoles (compartment, core) then
4          role := GetDeepestRole(compartment, core, method)
5          return role
6      else
7          return core
8      end
9  else
10     return core
11 end
```

In the play-relation, an object is allowed to bind to multiple roles, but those roles must stack up together to construct a deep-play-relation in the form of role-playing-role as seen in Figure 4.7. In other words, within a compartment, an object cannot play two or more roles at the same time if those roles are not related to each other. Attempting to bind two different roles yields a replacement of the old with the new one. In the example of the tax management system, a person instance cannot be bound to the `Developer` and the `Accountant` roles at the same time because these two roles are semantically not supposed to be composed to the core object together.

4.3.5.2 Lowering Functions

The operation of lowering is the inverse of the lifting function. Therefore, the primary task of the lowering function is to get a player instance, which can be queried from the lookup table, if we know the current role instance performing a lowering function. In the deep-play-relation, `invokePlayer(method, argsType, args)` cascades down by visiting each role instance in the play-relation while `invokeCore(method, argsType, args)` lowers directly to the core object (Figure 4.7). This lowering function has the same semantics of the `proceed` method in AOP and COP, and it also shares a similarity of `super` call in typical object-oriented languages.

In Figure 4.5, `ely.invoke("getPaid")` calls the `getPaid` method in the `Developer` role, but the implementation of the `getPaid` method will transfer the salary of 1500 to the `saving` attribute of `ely` by calling `this.invokePlayer("setSaving", 1500)`. The keyword `this` refers to the current instance of `Developer` role while `invokePlayer("setSaving", 1500)` results in a call of the `setSaving(1500)` method of `ely`, an instance of `Person`.

The lowering operation can be used to construct *partial method* calls when there is a deep-play-relation (i.e., a role plays roles). A partial method is a method whose definition is scattered among several bound role instances. The lowering technique allows each role instance, including the core object, to be invoked partially by visiting each instance in the relation. Therefore, behavioral variations can be achieved by constructing different role bindings in which the order of binding also matters. The lowering pseudocode is straightforward and is shown in Algorithm 4.2.

In order to highlight the partial method, let us revisit the chat server example, described in Chapter 2.4. The `Channel` object may have different behaviors according to the played roles `LZ` compression and `AES` encryption. Snippet 4.3 shows how `format` methods are defined in which each of them is responsible for specific behavior (i.e., while `LZ` for compression, `AES` for encryption). Figure 4.9 illustrates the behavior of the `Channel` object with respect to the different binding configurations. When the `format` method is called, it is lifted to the deepest role in the relation, and it is lowered down based on the lowering function implemented in each role. Hence, the method calls can be simulated as a stack which can be popped for execution until the stack is empty. Given different bindings as shown in Figure 4.9a, Figure 4.9b, Figure 4.9c, the end result is produced differently.

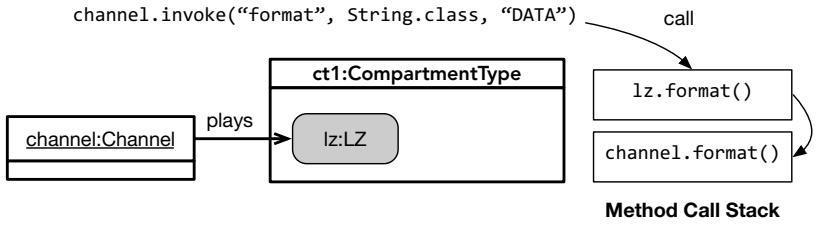
Algorithm 4.2: Lowering algorithm.

Input : currentRole, method
Output : object

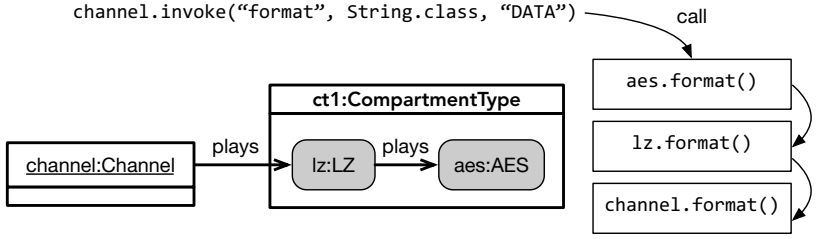
1 *compartment* := `GetActiveCompartment`(*CurrentThreadId*)
2 **if** *compartment is not null* **then**
3 | *role* := `GetPlayer`(*compartment*, *currentRole*, *method*)
4 | **return** *role* if not null; *core* otherwise
5 **else**
6 | Raise no active compartment exception
7 **end**

Snippet 4.3: Definition of `Channel`, `LZ` and `AES` in the chat server application.

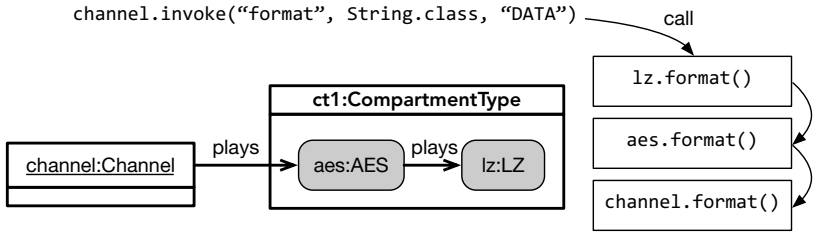
```
1 class Channel extends CoreObject{ //Core object
2   public void format(String data){
3     System.out.println(data);
4   }
5 }
6
7 class LZ extends Role{  //Role
8   public void format(String data){
9     String f_msg = "<LZ>" + data + "<LZ>";
10    invokePlayer("send", String.class, f_msg); //lowering
11  }
12 }
13
14 class AES extends Role{ //Role
15   public void format(String data){
16    String f_msg = "<AES>" + data + "<AES>";
17    invokePlayer("send", String.class, f_msg); //lowering
18  }
19 }
```



(a) Result is <LZ>DATA<LZ>.



(b) Result is <AES><LZ>DATA<LZ><AES>.



(c) Result is <LZ><AES>DATA<AES><LZ>.

Figure 4.9: Partial method with lowering function.

4.3.5.3 Collaboration Functions

The lowering and lifting functions specify the method dispatch between the core object and its played roles. *Collaboration Functions* further extend the accessibility to methods of an associating compartment and methods of roles which are played by different objects, but those roles have a relationship within the compartment.

From the run-time model perspective, roles seem to be declared inside a compartment as inner classes which allow them to access the outer class (compartment) easily like in ObjectTeams/Java [Her05]. As mentioned, LyRT decouples this association by treating all instances of core object, role, and compartment to be physically independent of each other. Therefore, a collaboration function, `invokeCompartment(method, argsType, args)`, is needed to access the compartment's methods. This design decision is to make the replacement of role instances at run time possible to support both anticipated and unanticipated adaptation.

The invocation of a relationship is embedded in the method call `invokeRel(RoleType, method, argsType, args)`. Algorithm 4.3 demonstrates how the underline mechanism of this function works. First, this function must be performed inside an active compartment because the actual method invocation targets on roles. Second, querying all the cores binding to the passing role type to invoke the respectively passing method.

In the example of the tax management system, described in Figure 4.5, the `Accountant` role has a *pays-salary* relationship with the `Developer` role in a `Company` compartment.

When `alice` invokes the `paySalary` method via the bound role `Accountant`, the `invokeRel(Developer.class, "getPaid")` method in Line 4 of Snippet 4.4 is executed by querying all the person instances which play the `Developer` role (i.e., `bob` and `ely`) to invoke the `getPaid` method as shown in Lines 10-15. In turn, the `getPaid` method deducts the salary from the revenue of the company compartment, specified in a `salary` property, from a `revenue` property of the compartment with collaboration function (Line 12) and then the amount is accumulated to the `saving` property of the respective persons by the lowering function (Line 14).

Algorithm 4.3: Algorithm of `invokeRel()`, a collaboration function.

Input : roleType, method

1 $compartment :=$ `GetActiveCompartment`$(CurrentThreadId)$
2 **if** *compartment is not null* **then**
3 $\quad \{cores_i\} :=$ `GetCores`$(compartment, roleType)$
4 $\quad$ **for** $i := 1$ **to** $n - 1$ **do**
5 $\quad\quad cores_i.invoke(method)$
6 $\quad$ **end**
7 **else**
8 $\quad$ Raise no active compartment exception
9 **end**

Snippet 4.4: Collaboration functions in the tax management example.

```
1 class Accountant extends Role{ //Role
2   public void paySalary(String data){
3     //invoke getPaid() from all persons binding to Developer role
4     invokeRel(Developer.class, "getPaid");
5   }
6 }
7
8 class Developer extends Role{ //Role
9   int salary = 1500;
10  public void getPaid(){
11    //deduct revenue from a compartment
12    invokeCompartment("deductRevenue", Double.class, salary);
13    //set saving to a person
14    invokePlayer("setSaving", int.class, salary);
15  }
16 }
```

4.3.6 Activation Styles

Anticipated adaptation is determined by application developers who give all the necessary compartments and roles to be activated or deactivated during development time although the decision of performing adaptation can be made at run time. LyRT is designed to support adaptation, on a per-thread and per-instance basis. Compartments can be activated in

different threads, thus adapting program behavior at the thread level. However, in each thread, there is only one compartment instance which can be active at a time. Role binding is performed at the instance level when a compartment is activated, only particular core objects binding to roles change their behavior.

4.3.6.1 Synchronous Activation

In a single thread environment, adaptation can be performed sequentially during the run-time execution by means of activating a compartment instance or activating of adaptation block within an active compartment as mentioned in Section 4.2.3. This activation technique is also known as synchronous activation or per-control-flow activation in COP [KAM15]. Typically, this kind of activation is straightforward, easy to understand, and refrains from the problem of inconsistency (see Section 4.4). The reason for that is the programmer's awareness of the execution point when the dynamic behavior becomes active.

4.3.6.2 Asynchronous Activation

LyRT allows core objects to change their behavior inside an active compartment from another thread which is running asynchronously from the main thread. The reason is that adaptation can be triggered at any point in the program and from different sources such as from context and/or an event. The active compartment and the core objects to be adapted must be shared across different threads so that they can be used to reconfigure the new role binding with `AdaptationBlock` construct of an active compartment when a predefined condition is met. Although the technique is flexible, it faces inconsistency issues when the core objects are not ready to adapt. This problem is addressed in Section 4.4.

Snippet 4.5 shows a sample of the chat server scenario on how the activation could happen asynchronously in order to modify the behavior of the current executing objects. The `Channel` object can have two different behaviors, i.e., `LZ` or `LZX` compression to transmit the data. These compression algorithms are changed with respect to the bandwidth of the network, which is monitored on another thread (Lines 9-25). Therefore, in the main thread, the `Channel` object sends file chunks with the specified algorithm (Lines 28-31).

Snippet 4.5: Asynchronous activation triggered from another thread.

```
1  Channel channel = Registry.newCore(Channel.class); //Core
2  Session s1 = Registry.newCompartment(Session.class);//Compartment
3
4  s1.initBinding(){ //Initial binding
5    channel.bind(LZ.class);
6  }
7
8  //Spawn a thread to monitor the network bandwidth for adaptation
9  Runnable eventMonitor = () -> {
10   while(true){
11     if(Network.bandwidth() > 1000){ //bandwidth is above 1Mbps
12       s1.activate(); //activate for rebinding process
13       AdaptationBlock ab = new AdaptationBlock(){
14         channel.bind(LZX.class); //binding new LZX compression role
15       }
16       s1.deactivate();
17     } else {
```

```
18          s1.activate(); //activate for rebinding process
19          AdaptationBlock ab = new AdaptationBlock(){
20            channel.bind(LZ.class); //bind to LZ compression role
21          }
22          s1.deactivate();
23        }
24      }
25    }
26
27    //Main execution
28    s1.activate(); //activate for dynamic behavior
29    while(!EOF(file)){
30      channel.invoke("send", getChunk(file));
31    }
```

4.3.7 Life Cycle of a Role

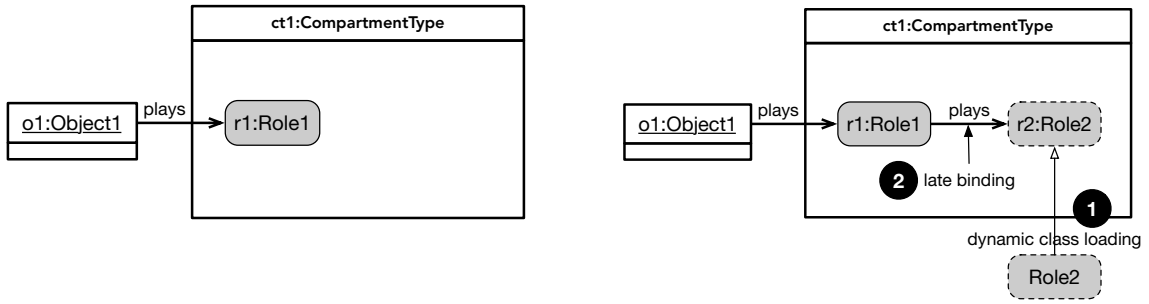
A life cycle of a role instance happens in the following stages:

- **Creation**: A role instance is implicitly created during the binding process to a core object. The newly created role instance might become active immediately if its compartment instance is currently active; otherwise, it has to wait until its compartment instance is activated.
- **Activation**: Because of context-dependent property, a role instance becomes active only when the compartment instance it associates with is active.
- **Transfer**: A role instance can be transferred from one core object to another without losing its state. However, during the transfer process, methods corresponding to the role instance should not be actively invoked. Due to decoupling from core objects, a role instance can be transferred from current core object to another core object by simply updating the binding relation in the lookup table.
- **Deactivation**: If its compartment is deactivated, a role instance becomes inactive but its physical instance still exists. There is no passive/active state of the role instance supported in an active compartment.
- **Destruction**: A role instance is destroyed in the unbinding process, or when the compartment instance is destroyed. Again, deactivation of the compartment instance does not destroy the role instance.

4.3.8 Supporting Unanticipated Adaptation

The rigorous design of the dynamic instance binding mechanism enables easy replacement of roles dynamically at run time, enabling the possibility of (re-)loading existing or new roles which eventually trigger unanticipated adaptation based on asynchronous activation.

A concept of bringing roles to be bound at run time is depicted in Figure 4.10. First, it is necessary to have a support for dynamic instance binding in a runtime as shown in Figure 4.10a. Second, there is an active compartment which specifies roles bound to an existing core object. Figure 4.10b exemplifies the steps. Initially, before loading new roles



(a) Before loading a new role. **(b)** During and after loading a new role.

Figure 4.10: Loading and reloading roles dynamically.

there is an active compartment `ct1` containing a binding relation between core object `o1` and role `r1`. Next, the new role type, `Role2`, can be loaded dynamically via a *dynamic class loader* in Step ① before in Step ②, the binding relation is constructed. Section 5.4 discusses in detail about an implementation of the dynamic class loader. The entire process does not physically affect the existing core objects. The application states are fully preserved. This increases flexibility and reduces disruption while performing updates, as the change of every instance of a given type, is not necessary. Once the binding relation is constructed, the behavior of the core object is dispatched to the newly bound role respecting to the lifting operation, described in Section 4.3.5.

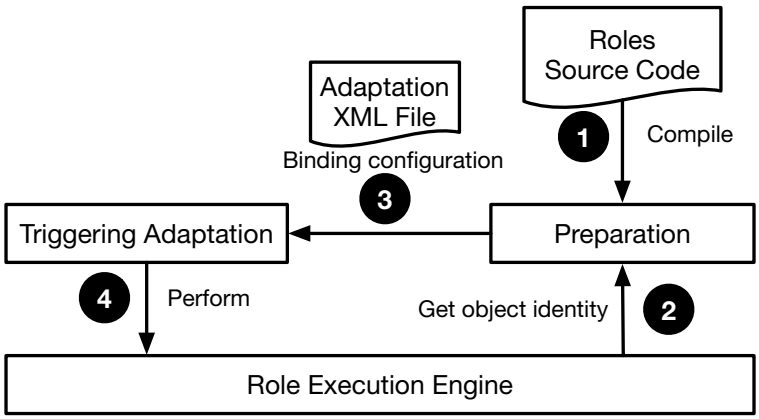


Figure 4.11: Process of unanticipated adaptation.

To realize unanticipated adaptation, there are four steps to be performed by application developers. First, new roles, which define the new behavior of certain objects, must be given and compiled ①. Second, the developers obtain current binding information from the lookup table, in which a tool to dump the lookup table is given ②. The information is necessary to construct the binding relation in an adaptation file in step ③. The adaptation file is an XML file specifying the information for constructing the binding relation such as an active compartment and new role types to be bound to the existing objects, i.e., cores or roles. Once the XML configuration is prepared, the file can be parsed by Application Programming Interface (API) of the runtime ④. In case the binding operations are specified, the prescribed role type is loaded into the runtime, and the binding relation is stored in the lookup table. Conversely, the unbinding process deletes the binding relation in the lookup table and destroys the role instance. This happens asynchronously because the parsing

process to manipulate the lookup table is triggered from another thread.

Snippet 4.6: An adaptation XML file.

```
<?xml version="1.0"?>
<adaptation>
  <compartment type="value" id="value">
    <bind [coreId="value" | roleId="value"] roleType="value" />
    <unbind [coreId="value" | roleId="value"] roleType="value" />
    <rebind [coreId="value" | roleId="value"] roleType="value" />
    <transfer fromCoreId="value"  toCoreId="value" roleType="value" />
    <[bind | rebind] [coreId="value" | roleId="value"] roleType="value" >
      <invoke method="method_name" returnType="void" />
    </[bind | rebind]>
  </compartment>
</adaptation>
```

Snippet 4.6 shows the elements composed in a configuration for unanticipated adaptation. The detailed description of each element is done as follows:

- **adaptation**: A top-level element for unanticipated adaptation. It contains at least one active compartment element for reconfiguration of core object binding. Several compartment elements can be described if developers want to adapt in multiple compartments executing in different threads. Note that there is only one active compartment per thread at a particular point in time.
- **compartment**: This element represents an active compartment where role operations are taking place. The `type` attribute is the fully qualified type name while `id` is the identity of the compartment which can be queried from the runtime. There are sub-elements inside a compartment element which are related to the role operations, i.e., binding, unbinding, rebinding, and transferring.
- **bind**: The element realizes the binding between either a core object or a role instance and a given role in the `roleType` attribute depending on whether a `coreId` or `roleId` is given. The value of both `coreId` and `roleId` is an object's identity which can be extracted from the runtime (i.e., the lookup table). In the process of parsing, the role type is loaded into the runtime (e.g., JVM), and a role instance of that type is instantiated.
- **unbind**: A given role specified in the `roleType` attribute is unbound from either core object or role instance depending on whether the `coreId` or `roleId` attribute is given. The role instance is destroyed and its binding relation is also removed from the lookup table.
- **rebind**: This element results from the combination of the `unbind` and `bind` element.
- **transfer**: This element detaches a role instance of a given type specified in the `RoleType` attribute from a core object (`fromCoreId` attribute), and attaches it to another core object as denoted in the `toCoreId` attribute within the same compartment. As the role instance is not destroyed, its states are preserved.
- **invoke**: Either `bind` or `rebind` element may contain single or several `invoke` elements which define a method in a given role type to be explicitly called.

4.3.9 Overcoming Object Schizophrenia

The loose binding of two or more objects which represent themselves as a single compound object sharing a common identity may lead to the problem of object schizophrenia [SR02]. Especially, it happens in the form of a broken identity when one of the participating objects changes its identity. This problem causes the dynamic method dispatch to be no longer valid among member objects.

In LyRT, when a compartment is activated, an object with a set of bound roles is just one compound object although those roles are physically separated. This activation minimizes the risk of broken identity as selecting roles is done only in an active compartment [HHM04]. However, the possibility of a broken identity of roles remains if those identities are duplicated or changed. In this regard, we rely on the `hashcode` method to generate a per-lifetime, unique identity for every object including its role instances. This `hashcode` is used as an object identity to store in and to query from the lookup table. Furthermore, the lookup table controls the object binding relation which can only be altered explicitly by means of adaptation; otherwise, it maintains a consistent binding relation. Therefore, we consider LyRT to be free of object schizophrenia problem.

4.3.10 Runtime Interaction with the Registry

The registry is a core component that allows programmers to interact with the runtime. To write role-based applications in LyRT, programmers need to understand the CROM [KBGA15] model and the necessary functions supported in the registry. The registry contains functions which deal with the creation of CROM elements, such as compartments, roles, and core objects, as shown in Table 4.2. These instantiations as well as the role binding process create a new record in the lookup table, which is later used for translating the method polymorphism. Table 4.3 displays the block constructs in LyRT to facilitate the initial binding, the subsequent adaptation, and consistency (Section 4.4). Table 4.4 shows the supporting functions of composition and method dispatch. There are other role-manipulated functions which are not discussed. Detailed instructions on how to use these functions in order to run role-based applications will be described in the case studies (Section 6.1).

Table 4.2: Creation of compartment and object and role binding functions.

Description	Method
Creating Compartment	`newCompartment(CompType, argsType, args)`
Creating Core Object	`newCore(ObjectType, argsType, args)`
Binding role	`bind(Object, RoleType, argsType, args)`
Unbinding role	`unbind(Object, RoleType)`
Rebinding role	`rebind(Object, RoleType, argsType, args)`
Transferring role	`transfer(fromObject, RoleType, toObject)`
Activating compartment	`activate()`
Deactivating compartment	`deactivate()`
Destroying role instances	`destroy()`

Table 4.3: Block constructs.

Description	Method
Initial Binding Block	`Compartment.InitBinding(){...}`
Adaptation Block	`AdaptationBlock ab=new AdaptationBlock(){...}`
Consistency Block	`Consistency c1=new Consistency(){...}`

Table 4.4: Method dispatching functions.

Description	Method
Lifting	`invoke(method, argsType, args)`
Lifting to a role	`invoke(RoleType, method, argsType, args)`
Lowering to player	`invokePlayer(method, argsType, args)`
Lowering to core	`invokeCore(method, argsType, args)`
Accessing to compartment	`invokeCompartment(method, argsType, args)`
Relation with role	`invokeRel(RoleType, method, argsType, args)`

4.4 Tranquility Controller

The main idea behind the tranquility controller is to determine whether a core object should promptly react to shift its behavior with respect to the adaptation, or whether it should wait if a series of ongoing methods is executing to achieve a common goal. While this problem remains unexplored in role-based software systems [KLG+14, SGP12a], in practice there are applications which require multiple method invocations which need to be executed atomically (i.e., consistent behavior). In the chat server example, chunks of a file should be transmitted with the same data format, e.g., all without compression or all with compression, to avoid corruption when assembling at the client side.

The tranquility controller relies on the tranquility concept [VEBD07], which divides a program into multiple *consistency blocks*[4] containing a set of method executions of different cores as shown in Figure 4.12. The cores executing inside a consistency block are not allowed to change their behavior regardless of adaptation. Regions outside the consistency block are called *tranquil states* where the adaptation can be performed safely. Our object-level tranquility concept extends the original tranquility concept to support the safe update of the node at instance level and to provide parallel execution. Section 6.1.4.3 discusses the differences between the two in a case study.

[4]Originally, the term *transaction* is used. To avoid confusion over database transactions, the term *consistency block* is used.

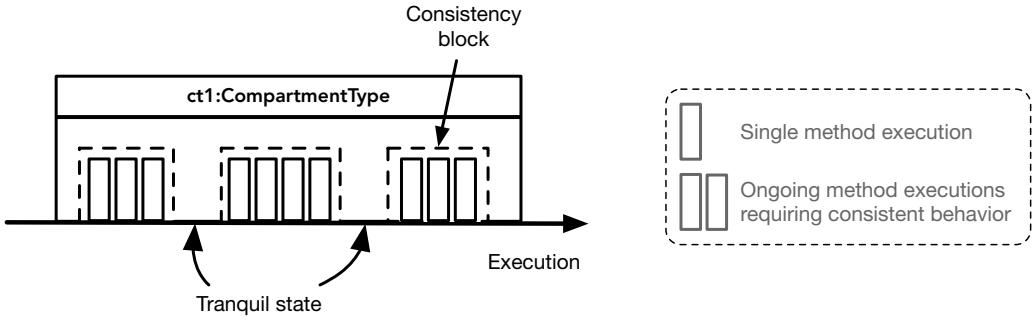


Figure 4.12: Tranquility concept.

4.4.1 Consistency Block

The adaptation in LyRT can happen either synchronously or asynchronously (see Section 4.3.6). Synchronous adaptation is defined in the flow of the program execution within an active thread. This strategy does not imply any system inconsistency because adaptation occurs sequentially under the control of the developers. Asynchronous activation, however, is triggered by a shared thread to adapt objects in the executing thread. It allows a flexible adaptation which can be easily derived from context reasoning in different program threads. Consequently, this crosscutting adaptation may affect system inconsistency as objects are engaging in a *transaction*[5] which requires uniform behavior from the start to the end of the transaction.

Kramer and Magee [KM90] define a transaction as "a sequence of messages that must be executed atomically". The messages in the definition refer to the method invocations of objects at the instance level. We assume that a transaction is an explicit concept which is known to the developer. It means that in some parts of the program code, there are transaction blocks surrounding the code to be executed atomically. The atomic property in this regard refers to the uniform behaviors which should not be changed as a result of an adaptation. The notion of this transaction is not related to the term transaction in the domain of databases, in which *atomicity*, *consistency*, *isolation*, and *durability* (ACID) properties are ensured. To avoid confusion, the term *consistency block* is used.

Snippet 4.7 shows a sample consistency block in a program where the `Channel` object in our chat server example (Chapter 2) is not allowed to change the way it formats the data before transmission, i.e., performing encryption, inside the block. Outside the block, the adaptation can be performed ordinarily.

Snippet 4.7: Application of `ConsistencyBlock`.

```
1  //Adaptation can be performed before entering the consistency block
2  (ConsistencyBlock cb1 = new ConsistencyBlock()){
3    while(!EOF(file)){ //not end of file
4      channel.invoke("send", getChunk(file)); //send a chunk
5    }
6  }
7  //Adaptation can be performed after the consistency block
```

[5]The term transaction here is defined in the concept of quiescence [KM90] and tranquility [VEBD07]. The term *consistency block* is used to avoid confusion over the database transaction.

```
8
9   //channel object may use different algorithms to transmit the data
10  (ConsistencyBlock cb2 = new ConsistencyBlock()){
11    while(!EOF(file)){ //not end of file
12      channel.invoke("send", getChunk(file)); //send a chunk
13    }
14  }
```

A consistency block is a scope of a *consistency object* (i.e., `cb1` and `cb2` in Snippet 4.7 in Line 2 and 10) that lives only in the block. The consistency object is instantiated when the block is activated and is destroyed when the block is ended. The consistency object, representing the consistency block, is registered with a starting time in an active compartment of a particular thread to anchor the appropriate role for method invocation during its lifecycle. Those roles will be maintained to ensure the uniform behavior during the block execution regardless of any adaptation demanding for binding or unbinding those roles. The consistency block and its associated consistency object are used interchangeably. Table 4.5 shows a sample registration of the first consistency blocks shown in Figure 4.13.

The consistency block must be provided by developers to guard the code block in which adaptation must not happen in order to ensure system consistency. This block is a thread-based execution meaning that it prevents any objects within the block from being adapted within its own thread while it is still possible to perform an adaptation of the same object instance executed in different threads.

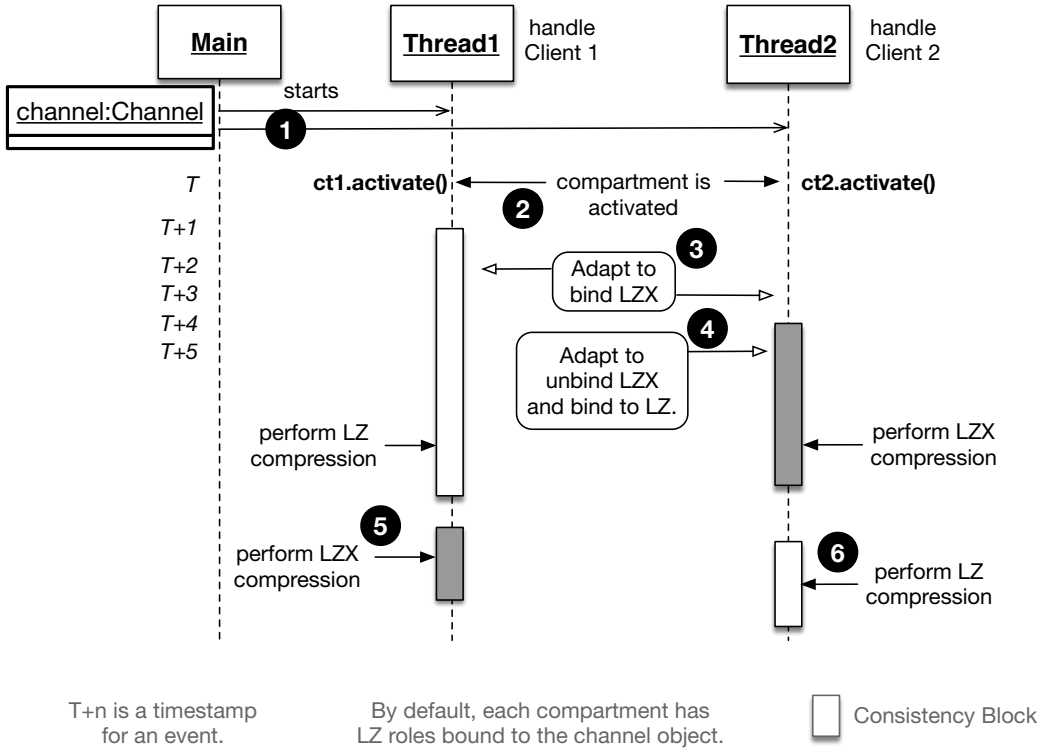


Figure 4.13: Consistent behavior of a shared object in a multi-threaded environment.

Considering our chat server example in a detailed process as depicted in Figure 4.13, the shared `Channel` object is used to transmit the data for different clients, i.e., `Thread1` and `Thread2` are for client 1 and client 2 respectively. This example demonstrates the change of the sending behavior of the `Channel` object with respect to the thread and consistency

block it is executing in. When a new client connects, a new thread is created to handle the client session (step ①). In each thread, there is an active compartment instance derived from the same type where a LZ role for data compression is the default binding role (step ②). In Thread1, the consistency block is started by performing the data transmission with the LZ compression role.

Soon after the start of this consistency block, the network condition presumably changes and demands binding to a LZX compression role (step ③). This adaptation is also applied to Thread2, handling another client session. Since Thread1 has already started the block, there is no effect to ensure the consistent behavior within the block. However, in Thread2, the new LZX behavior is applied to the Channel object because it enters the consistency block after the adaptation took place.

In step ④, another adaptation happens to unbind the LZX role which is currently executing in Thread2. In this regard, this LZX role must not be removed immediately from the Channel object. Instead, it is marked with the *unbound time*. The role with the unbound time will be deleted after the consistency block expires. This technique ensures that the execution of methods in the consistency block remains consistent regardless of the addition or the withdrawal of roles. After the expiration of the consistency block, the adaptation becomes effective for the next method call (step ⑤ and ⑥).

4.4.2 A Solution to Realize the Consistency Block

In order to achieve the consistency block as described in the previous section, the system relies on the dynamic instance binding mechanism to select the appropriate roles for method dispatching. The BoundTime and UnboundTime attributes in the Relation table of the lookup table (Section 4.3.4) help to determine which roles should be selected for invocation.

4.4.2.1 Dealing with Addition of New Roles

Whenever the adaptation occurs resulting in addition of a new role, the role and its binding relation are added to the runtime and the lookup table respectively. The runtime stores the time when the consistency block started in each thread. There is only one consistency block which can be active in each thread at a specific point in time. Table 4.5 shows the registration of the first consistency blocks which resembles the process described in Figure 4.13, denoted as cb11 and cb21 in Thread1 and Thread2 respectively. The StartTime column shows the value of the registered timestamp of each consistency block belonging to a particular thread.

If there is a running consistency block, the method dispatch (i.e., lifting operation) compares the times when the consistency block was started and the time when the new role is bound. If the BoundTime is greater than the StartTime of the consistency block, the role is excluded from the method invocation. Consequently, the adaptation does not affect the execution of methods inside the current consistency block as shown in step ③ of Thread1 in Figure 4.13. The adaptation will become effective with the next method calls after the block has expired (i.e., step ⑤ of Thread1). Algorithm 4.4 is a pseudo-code implementation for method dispatching (lifting) of a core executing when a consistency block is present.

Table 4.6 shows the snapshot of the lookup table from step ② to ④ in Figure 4.13. The consistency block cb11 locates in the compartment ct1 where two roles instances are bound

Algorithm 4.4: Lifting algorithm when a consistency block is present.

```
Input  : core, method
Output : object
1  compartment := GetActiveCompartment(CurrentThreadId)
2  if compartment is not null then
3      if HasConsistencyObject (CurrentThreadId) then
4          startTime := GetStartTimeOfConsistencyObject(CurrentThreadId)
5          if HasBoundRoleBeforeConsistencyBlock (compartment, core, startTime)
            then
6              role := GetDeepestRole(compartment, core, method, startTime)
7              return role
8          else
9              return core
10         end
11     else
12         return core
13     end
14 else
15     return core
16 end
```

to the same `Channel` object with different timestamps as seen in the `BoundTime` attribute. According to the method dispatch rule, the `lzx1:LZX` role should be invoked when bound. However, since the `cb11` block starts at time $T + 1$, `lz1:LZ` is used for invocation instead because `lz1:LZ` is bound earlier at time T. While `cb11` is executing, `lzx1:LZX` is bound at time $T + 2$. Hence, `lzx1:LZX` will be used in subsequent method calls either with or without a consistency block.

Table 4.5: Sample consistency block registration in each thread conforming to Figure 4.13.

ThreadId	CompartmentId	ConsistencyBlockId	StartTime
Thread1	ct1	cb11	T + 1
Thread2	ct2	cb21	T + 4

Table 4.6: Sample data of the chat server runtime in the `Relation` table as depicted in Figure 4.13. `ct1` and `ct2` are active compartments in `Thread1` and `Thread2` respectively.

Id	Comp.Id	CoreId	PlayerId	RoleId	BoundTime	UnboundTime
1	ct1	channel	channel	lz1:LZ	T	nil
2	ct1	channel	channel	lzx1:LZX	T + 2	nil
3	ct2	channel	channel	lzx2:LZX	T + 3	T + 5

4.4.2.2 Dealing with Removal of Bound Roles

Similar to adding a role, if the adaptation happens as a role removal, the role to be removed is not deleted immediately but it is marked as *unbound* with the timestamp specified in the `UnboundTime` attribute in the lookup table. This technique allows the objects to interact with the role to be removed in a consistency block as shown in step ④ of `Thread2` in Figure 4.13. The role to be removed is permanently deleted from the runtime as soon as the consistency block has expired. Consequently, the adaptation affects the method calls subsequent to the expiration of the consistency block, i.e., step ⑥ in `Thread2`. Similarly, if the bound role is transferred from one core object to another, it is considered as the role to be removed. Therefore, the same process explained above is applied.

As an example for removing a role during the execution of a consistency block, the `lzx2:LZX` role in the `ct2` compartment (Row 3 in Table 4.6) is marked *unbound* at time $T + 5$ but this role is part of the `cb21` consistency block. Therefore, it is kept until the `cb21` has expired.

To sum up, in order to realize consistent adaptation over ongoing multiple method invocations, we introduced the consistency block, which is specified by the developers in the program code. The `BoundTime` and `UnboundTime` attributes are updated as a result of adaptation to realize the consistency block. Finally, the runtime excludes roles to be bound or unbound from the method dispatch inside a consistency block in order to prevent the system from inconsistencies.

4.5 Rollback Recovery Controller

This component is designed to embrace run-time failures caused by bugs as a result of role composition which is triggered by adaptation. This composition is very dynamic in highly adaptive systems, and the developer cannot eliminate all the bugs during the testing phase. Simple bugs such as `DivideByZero` or `ArrayOutOfBound` can crash the runtime limiting the high availability of the runtime. These simple bugs are normally not caught by the compiler, e.g., Java compiler. Therefore, such bugs may appear during the execution.

There are many kinds of software bugs. The scope of this dissertation focuses on software bugs which might cause run-time failures during execution. While some of these bugs can normally be caught by the application exception handling during execution, others, like `ArithmeticException`, are not.

4.5.1 Failures Resulting from a Role Composition

Let us recall the activation strategy supported in LyRT. LyRT provides two levels of adaptation. First, the adaptation results from switching from one compartment to another at the coarse-grained level. Since roles are located inside the compartment, core objects are adapted according to those binding relations. Second, inside an active compartment, an object may change roles several times because of context change. The latter case is illustrated by a `DivideByZero` error, which is likely to occur, in our chat server example described in Chapter 2.

Figure 4.14 demonstrates a `DivideByZero` failure which may happen in a particular role composition triggered by three adaptations consecutively. First, the `Channel` object binds to the `LZ` role denoted in Figure 4.14a. According to the rule of dispatching, the invocation of `invoke("format", data)` in step ① is dispatched to the `format` method of the `LZX` role in step ②. In turn, the `factor` method of the `LZX` itself is called in step ③. Although the `factor` method may generate the value 0, there is no `DivideByZero` failure happening because any number to the power of 0 is 1.

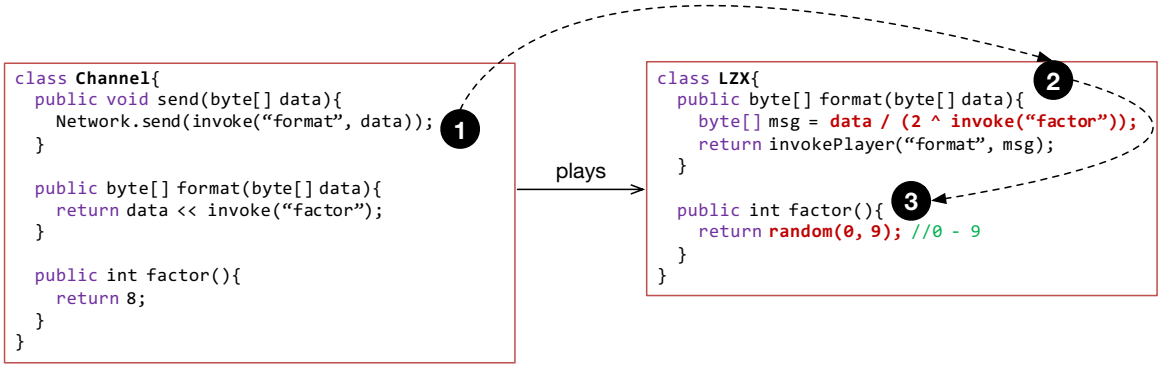
Second, the system adapts to play the `AES` role as shown in Figure 4.14b. The implementation of the `AES` role has no `factor` method as it relies on the `factor` method of the player, that is the `Channel` object (step ②). So, the `DivideByZero` failure never occurs since the implementation of the `factor` method returns the value 8 (step ③).

The composition in the third configuration, Figure 4.14c, may cause a `DivideByZero` failure. The configuration is composed of the `Channel` object, the `LZX` role, and the `AES` in the form of deep-play-relation. In step ① the `invoke("format", data)` method is dispatched to the `format` method of the `AES` role in regard to the dispatching rule. The `factor` method in step ② is dispatched to the implementation of the `factor` method of the `LZX` role because the `LZX` role is the player of the `AES` role. Therefore, whenever the `factor` method in step ③ returns the value 0, the call of the `format` method in `AES` results in a `DivideByZero` error.

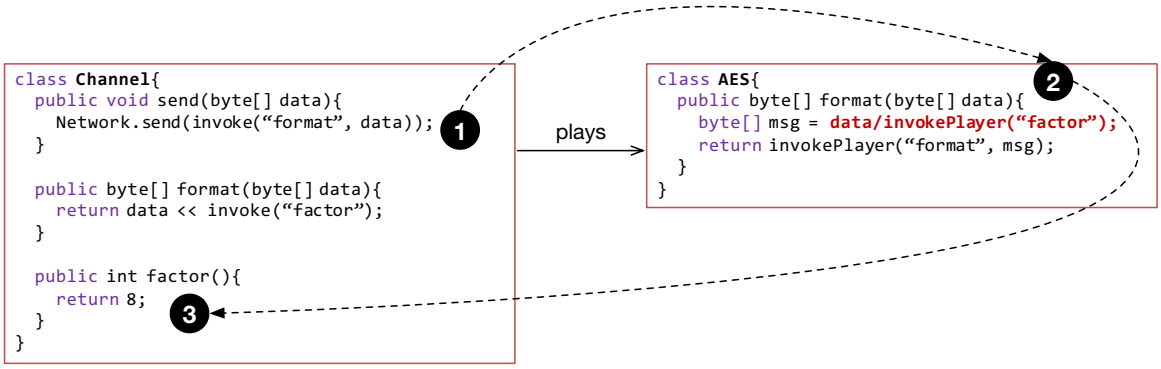
In the example above, while the composition of a role in isolation does not cause a failure, a particular relation of multiple roles might cause a run-time failure. Additionally, the order of the composition matters. For instance, in Figure 4.14c, there will be no failure if the `Channel` object first plays the `AES` role which then plays the `LZX` role because the `invokePlayer("factor")` of the `AES` role will be lowered to the `factor` method of the `Channel` object.

4.5.2 Rollback Recovery Mechanism

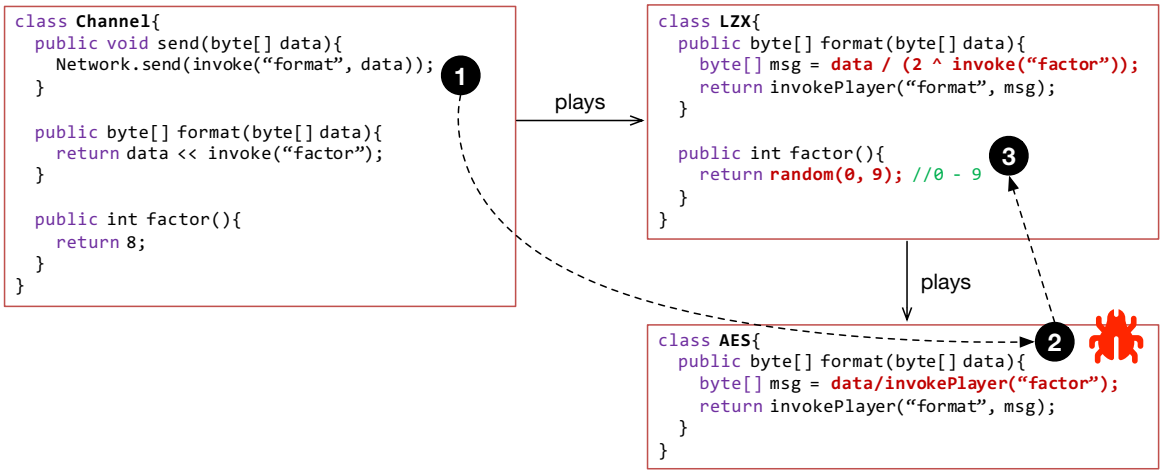
Adaptation results in changing a composition of roles to a core object (i.e., binding and unbinding). A method invocation of an object from a particular composition may contain an error leading to a failure. Therefore, the main idea of the approach is to embrace failures caused by bugs if we cannot avoid them, and enable the runtime to recover by rolling back to a recent checkpoint. The system generates a checkpoint before initiating a new adaptation. A *checkpoint* is a serialized representation of the current *application configuration* (AC), i.e., instances of roles and their associating compartment, including their states and binding information, reflecting the currently active play-relations between players and roles within a compartment.



(a) Channel object plays the LZX role.



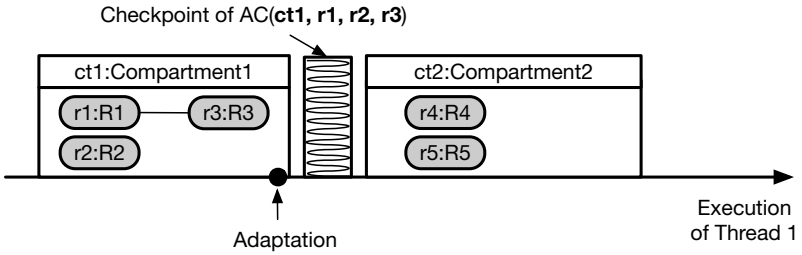
(b) Channel object plays the AES role.



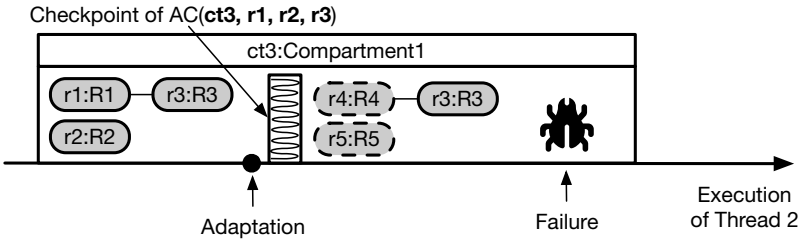
(c) Channel object plays the LZX role, and the LZX role plays the AES role. There is a chance of a DivideByZero failure when calling the factor method.

Figure 4.14: Composition leading to a DivideByZero failure.

After the checkpoint is created, the system performs the specified adaptation and the runtime reacts accordingly. The program may encounter bugs introduced by newly installed or updated role implementations. The system has a specialized bug sensor to detect bugs and to signal the runtime to roll back to the previous configuration by restoring the most recent checkpoint. That previous configuration is assumed to be error-free because bugs had not been caught within that application configuration. Meanwhile, the runtime records the defective configuration to prevent it from being re-activated. The system also generates a notification to the developer responsible for the bugs. The application configuration can be reapplied after the bug has been fixed and the new code version has been shipped.



(a) Checkpoint of an AC as a new compartment activation.



(b) Checkpoint of an AC as a change of roles in an active compartment.

Figure 4.15: A checkpoint of different activation strategies.

The adaptation in LyRT is done on a per-thread and per-instance basis in which it allows two or more instances of a compartment to live in different threads, and the activation of those compartments adapts core objects independently. Hence, the rollback recovery mechanism is applied on a per-thread basis to capture an independent rollback recovery process for each thread. Depending on the adaptation specified in each thread, a failure may happen in a particular thread which contains a defective role composition while the rest keeps running properly. Figure 4.15 illustrates the mentioned explanation by showing different styles of behavior activation and how checkpoints are made in isolation. Figure 4.15a shows that `Thread1` adapts the core object behavior by means of deactivating the current active compartment and activating another compartment without any sign of failure. The checkpoint is made before this transition. In contrast, Figure 4.15b depicts a role change within an active compartment of `Thread2` where a checkpoint of the current configuration needs to be made. Consequently, this change causes a failure, which needs to be recovered to keep the application running in a valid configuration. This recovery process is done independently from `Thread1`.

4.5.3 Rollback Recovery Architecture

The runtime architecture had to be extended as depicted in Figure 4.16 to incorporate the rollback recovery mechanism as part of LyRT. The extension comprises four components,

namely a bug sensor, a control unit, a checkpoint manager, and a rollback unit. With our approach, the dynamic adaptation cycle is completed by ① making a checkpoint of the current system configuration, ② adapting the application configuration, ③ detecting failures, ④ recovering from a software failure by rolling back to the previous application configuration and ⑤ notifying the developer about the failure.

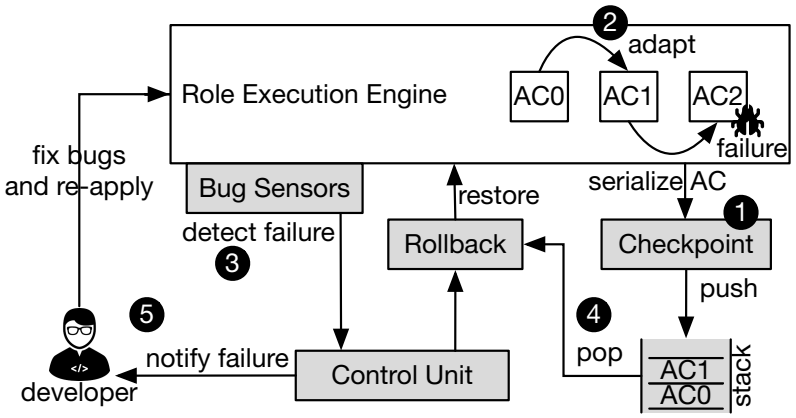


Figure 4.16: Rollback recovery architecture (AC: Application Configuration).

4.5.4 Checkpoint

In pursuit of supporting rollback, the runtime takes system snapshots as checkpoints, before performing an adaptation transaction. Given that the currently active application configuration is persisted in the lookup table, creating checkpoints consists of serializing the records containing all the role instances and binding information, pushing them to the stack (Step ① in Figure 4.16). Depending on the way we persist the data, serialization may be time-consuming, however, this is an effective method to preserve the objects' states and their dependencies, as opposed to shallow copying or cloning.

Besides the state stored in attributes of objects that represent roles, the roles' behavior might manipulate externally stored data, e.g., by utilizing file or database operations. For a complete rollback, such external state needs to be considered for checkpoint and rollback. While file versioning systems can be applied to deal with the rollback of files, database versioning is also on the verge of realizing this concept [HVBL15]. However, for the sake of simplicity, supporting this external state is considered out of scope.

The process of the checkpoint is embedded in the adaptation process. In case of performing the adaptation by switching from one compartment to another, the checkpoint is done when the former compartment is deactivated to serialize the current configuration of itself (Figure 4.15a). If adaptation happens by unbinding and binding new roles in an active compartment, these operations are placed in the adaptation block where the checkpoint can be made when the block is activated. Snippet 4.8 resembles the adaptation process as depicted in Figure 4.15b in coding style. Initially, roles `R1`, `R2`, and `R3` are bound to different objects. The subsequent adaptation is asked to replace `R1` and `R2` with `R4` and `R5` respectively. Therefore, the adaptation block (Lines 13-16) is necessary to let the runtime makes a checkpoint, which contains the instances of `R1`, `R2`, and the current compartment `ct3`, before applying new adaptation.

Snippet 4.8: Checkpoint is made when `AdaptationBlock` is activated.

```
1  //Initial binding
2  ct3.initBinding(){
3    obj1.bind(R1.class);
4    obj2.bind(R2.class);
5    obj3.bind(R3.class);
6  }
7
8  ct3.activate(); //Compartment activation
9
10 //Executing....
11
12 //Triggers adaptation
13 (AdaptionBlock ab1 = new AdaptationBlock()){ //checkpoint is created
14   obj1.bind(R4.class); //new binding to replace R1
15   obj2.bind(R5.class); //new binding to replace R2
16 }
```

For a large server application which involves several adaptations, keeping the checkpoint for all the executing threads consumes more memory if they are stored in memory. Although checkpoints can be stored on disk, the rollback process would take much longer due to slower disk access. Therefore, the maximum number of possible checkpoints should be limited by the developer who can estimate the frequency of adaptation in the system. LyRT sets the threshold value of the checkpoint to be 10 for each thread by default.

4.5.5 Bug Sensors

Bug sensors are run-time components designed to detect software failures that are uncaught during testing or compilation. Bug sensors are used to signal the Control Unit for recovery, as depicted in step ③ of Figure 4.16. Inspired by the software fault tolerance domain [QTSZ05], two types of bug sensors can be deployed during run time. The first type of sensors utilizes the exception handling system used by the application. The second type of bug sensors handles memory-related bugs, by detecting buffer overflow, memory leaks, etc. However, garbage-collected languages, such as Java, resolve most of the memory-related issues. Therefore, we consider only the first type of bug sensors to monitor the run-time failures. This technique tackles only the failures raised by run-time exceptions. Bug sensors are installed either globally in the main program to intercept all exceptions raised from all threads, or locally to each thread, capable of catching only a particular thread's exceptions. In a scenario like our chat server example, the sensor should be installed in each thread because variability and potential failures exist on the thread level. Although we provide the default implementation of a bug sensor, developers can customize it to suit their needs.

4.5.6 Rollback

The runtime rolls back by destroying the current defective configuration and restoring the latest checkpoint from the stack (step ④ in Figure 4.16). This process involves deleting the current records in the lookup table and inserting the serialized records from the checkpoint. The lookup table manages instance relations and their executing thread information. Therefore, the rollback, which is performed on a per-thread level, is relevant to the threads in

which the failure occurred. This design minimizes the overall system disruption and data loss.

Although clients do not experience disconnection, the rollback probably incurs data loss during its recovery process. We leave this problem to the application protocol to keep the executing messages in the cache and to resend the cached messages once the recovery is done.

Another scenario which does not limit the applicability of this rollback is to apply a new feature to a testing user group. Considering our chat server example, each user in the testing group has their own representation of the data transmission (e.g., without compression or with compression). The new encryption feature (`AES`) is applied to particular users in the testing group with different composition configurations before system-wide adoption. A bug in the specific encryption composition has only an impact on the testing group while it has no effect on the regular clients. Since LyRT allows unanticipated adaptation, the bug can be fixed, and the fixed configuration can be reapplied subsequently.

4.5.7 Control Unit

It is a central component handling checkpoints, rollback, and bug notifications. The Control Unit communicates with the adaptation activation to generate a checkpoint between each adaptation and listens to the signal from the bug sensors to execute a rollback. The Control Unit records the defective adaptation configurations in order to avoid the same configuration from being activated at a later stage unless fixes have been made. Meanwhile, the Control Unit notifies developers about the failure and reapplies the configuration once the bug is solved, by means of unanticipated adaptation as illustrated in step ⑤ of Figure 4.16.

4.6 Required Features of Host Languages to Implement LyRT

The concept behind LyRT is introduced to be generic; so it can be implemented in any modern object-oriented technology although we chose Java as a host language. However, LyRT still requires some language features such as *reflection*, *dynamic class loading*, and *exception handling* to support its variability.

Reflection. Reflection provides a low-level API, which can be used to query methods and their accepted arguments for a specific class. This feature is used heavily in our dynamic method dispatch as the methods of roles are queried and cached for later invocation. A method call from a core object is intercepted and compared to the cached methods of the bound roles. If the signature of a calling method is matched, technically the identical cached method of a role is invoked. The invocation process also relies on reflection due to the late method binding as it is determined during run time, especially in the case when a new role instance is bound dynamically.

Dynamic Class Adaptation. Another feature, which should be supported in the host language to implement LyRT, is the dynamic class adaptation which allows the new implementation of an existing class or a new class to be reloaded and loaded respectively. This feature is essential for supporting unanticipated adaptation in which roles are required to be adapted during execution.

Exception Handling. Besides, an exception handling mechanism must be supported in the host language as well. This mechanism can be used to implement the bug sensor.

The mentioned features are typically supported in any modern OOP languages. Therefore, the concept to design LyRT can be implemented in those languages without issues. Next chapter, we will show how this concept can be implemented in Java as a framework.

4.7 Chapter Summary

This chapter illustrated the conceptual work of LyRT which introduced three main essential components, namely the *Role Execution Engine*, the *Tranquility Controller*, and the *Rollback Recovery Controller*, in a single run-time architecture for dynamic variability. The Role Execution Engine is designed based on a dynamic instance binding mechanism developed to support loosely coupled bindings between instances to achieve adaptation on a per-thread and per-instance basis. Additionally, unanticipated adaptation has been designed from the ground up and embedded in the dynamic instance binding mechanism. LyRT allows programmers to interact with the runtime by means of accessing to functions of the *registry*.

The Tranquility Controller consists of a consistency block mechanism to surround application code containing a series of method invocations of objects which are required to be executed consistently. Therefore, those objects are not allowed to change their behavior in the presence of the consistency block.

The Rollback Recovery Controller is an extension to the Role Execution Engine to make a checkpoint of the current application configuration, i.e., role instances and their compartment instances, before applying a new adaptation. A specialized sensor utilizing the application exception handling is installed to embrace run-time failures. Once a failure is detected, the runtime rolls back to the previously saved checkpoint in order to reinstate the thread in which the failure occurred. Thanks to the support of adaptation on a per-thread and per-instance basis, which allows us to make a checkpoint and roll back on the thread level, it minimizes the overall run-time disruption and data loss. Hence, it improves run-time stability.

In order to implement LyRT, host languages require a few features namely reflection, dynamic class adaptation and exception handling. Modern object-oriented programming languages support those. In the next chapter, we will show the prototype implementation of LyRT in Java.

Chapter 5

LyRT Implementation

An important part of demonstrating the feasibility of LyRT involves building a realistic prototype which we can use to assess the features of the presented approach with respect to the requirements of run-time variability and to also evaluate its performance aspects. In this chapter, we describe the implementation of LyRT as a library framework in Java. Afterwards, we will discuss the trade-offs of using Java, as opposed to other dynamically typed languages.

5.1 An Implementation Overview

A prototype of LyRT is implemented in Java as a library in order to demonstrate the generality of the proposed approach. Due to the fact that Java is a statically and strongly typed language it is more challenging to implement an adaptive run-time system than to rely on dynamically typed languages. As a library, the prototype runs on pure JVM and requires no additional development tooling.

The essential ingredients of the prototype implementation are depicted in Figure 5.1 and briefly described as follows:

Registry and Lookup Table. This is the core run-time component where most of the implementation logic of the dynamic instance binding mechanism and the lookup table are located. The `Registry` is a singleton class which is used by other components.

Dynamic Method Invocation. This part contains an implementation of cached method tables which are stored in each compartment instance for handling the lifting, lowering and collaboration functions. In order to improve performance, the `invokedynamic` opcode instruction is used for method invocation.

Dynamic Class Reloading. In order to incorporate the unanticipated adaptation support, a role class has to be dynamically loaded and reloaded if its implementation is altered. The XML helper is needed to parse the unanticipated adaptation script.

Block Constructs. This component describes the implementation of the three distinct block constructs including the `ConsistencyBlock` to realize the tranquility controller.

Rollback Recovery. This component handles checkpoints, and rolls back if the runtime gets a signal from the bug sensor reporting an error caused by role composition.

Compartment, Role and Core Object. These are required entities which programmers use to implement role-based adaptive systems running on LyRT.

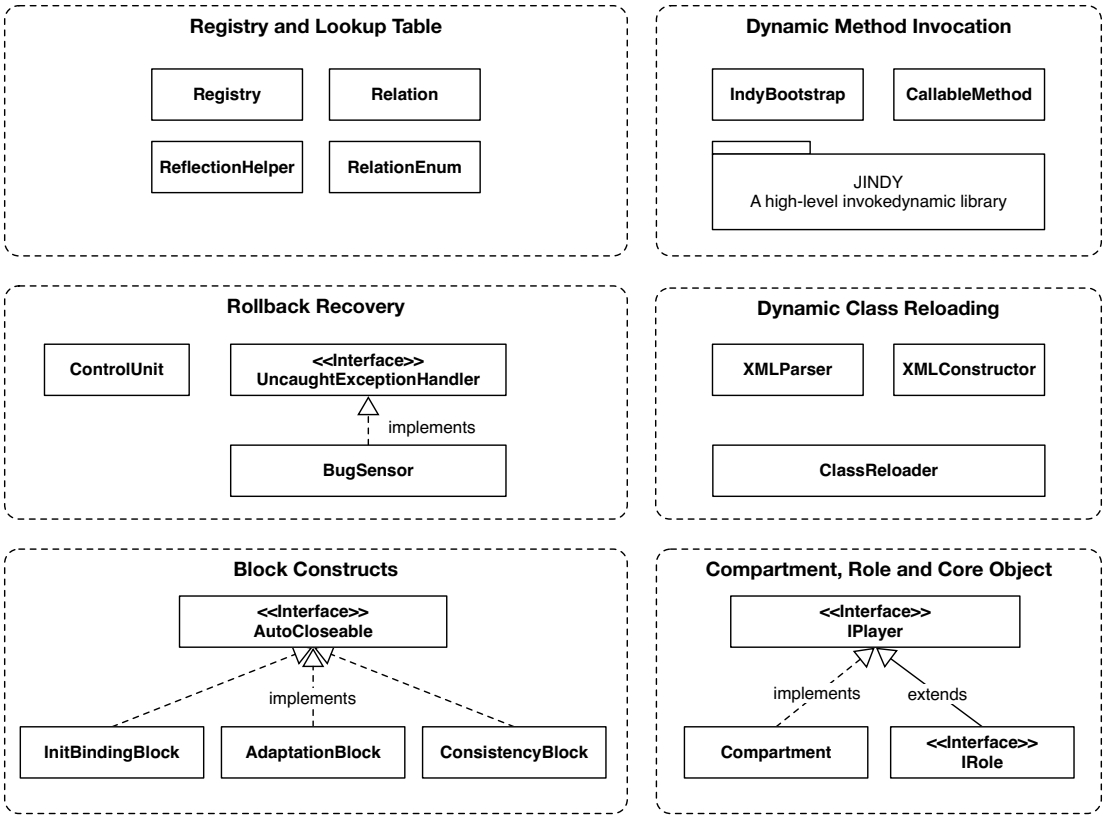


Figure 5.1: An overview of LyRT implementation and its class diagram.

5.2 Registry and Lookup Table

The registry is the core function of LyRT which allows programmers to develop a role-based application following the CROM [KBGA15]. The registry implements the dynamic instance binding mechanism that manipulates the lookup table in order to manage the instances of compartment, role and core objects. The class diagram related to this registry can be found in Figure 5.2.

The lookup table is an `ArrayDeque<Relation>` which is an efficient generic list structure in Java. The lookup table is stored locally in each compartment object. The `Relation` contains the object references as follows:

- **Compartment**: an `Object` represents a compartment object.
- **Core**: an `Object` represents a core object.
- **Player**: also an `Object`. If the relation kind is a *play*-relation, the player object is the core object. Otherwise (i.e., *deep-play*-relation), this player is a role. If a compartment acts as a core object to play a role, this player is the compartment object.
- **Role**: an `Object` represents a role object.
- **Depth**: an `int` denotes the distance of a role with respect to the core object.

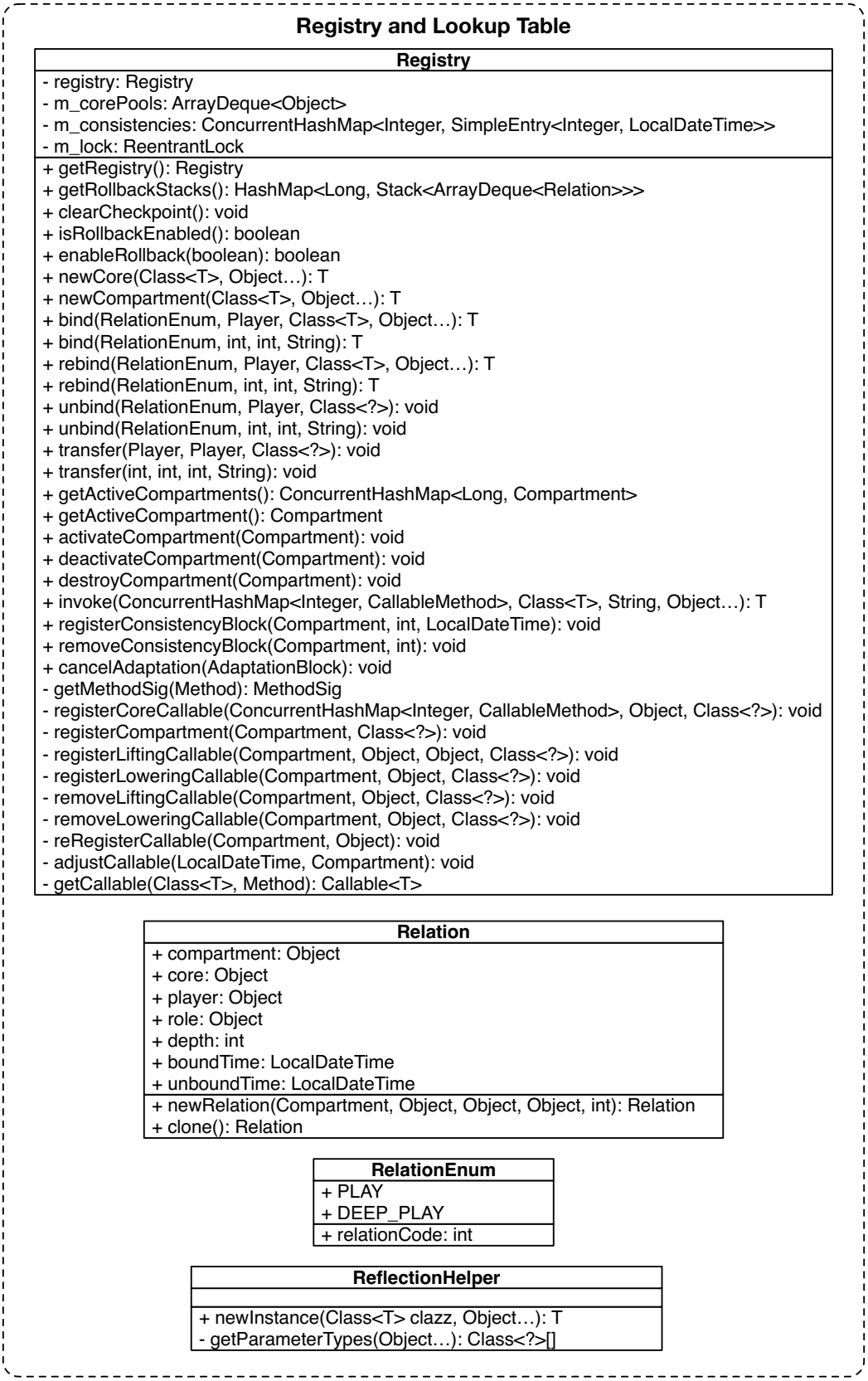


Figure 5.2: A class diagram of registry's components.

- **BoundTime**: a `java.time.LocalDateTime` type represents the time when a role got bound.
- **UnboundTime**: a `java.time.LocalDateTime` type represents the time when a role got unbound. This meta-data is valid only when a consistency block is present.

The lookup table presented in Chapter 4 is a relational database schema in which we can rely on an in-memory database to store the *play*-relation and the *deep-play*-relation. We stick to the `ArrayDeque<Relation>` data structure because of two reasons. First, it is easy and fast for a reasonable amount of relations. Additionally, Java 8 provides the `java.util.stream` API[1], which enables functional-style operations on streams of elements, in order to manipulate the searching process of the list data type with ease. Second, there is no need for a separate list structure to implement instance pools which store the references of compartments, roles, and core objects respectively. However, the manipulation of millions of objects using `ArrayDeque<Relation>` is more expensive than that of the in-memory database. An application might have millions of objects but the lookup table is kept separately in each compartment instance. So, it is unlikely in practice that a compartment would hold millions of roles.

5.3 Dynamic Method Invocation

In order to achieve dynamic application behavior, a method should be resolved dynamically at run time. Reflection is a suitable yet time-consuming approach. Thus, in order to compensate the overhead while maintaining the adaptability, we implement a table for caching methods of a core and its bound roles. This caching follows our method dispatching rule as explained in Section 4.3.5. Each method in the table has a reference to a `Callable<?>` interface whose concrete implementation is dynamically generated and contains the new `invokedynamic` instruction. Hence, a method is invoked utilizing this new opcode instruction instead of using reflection. The class diagram for this component can be found in Figure 5.3.

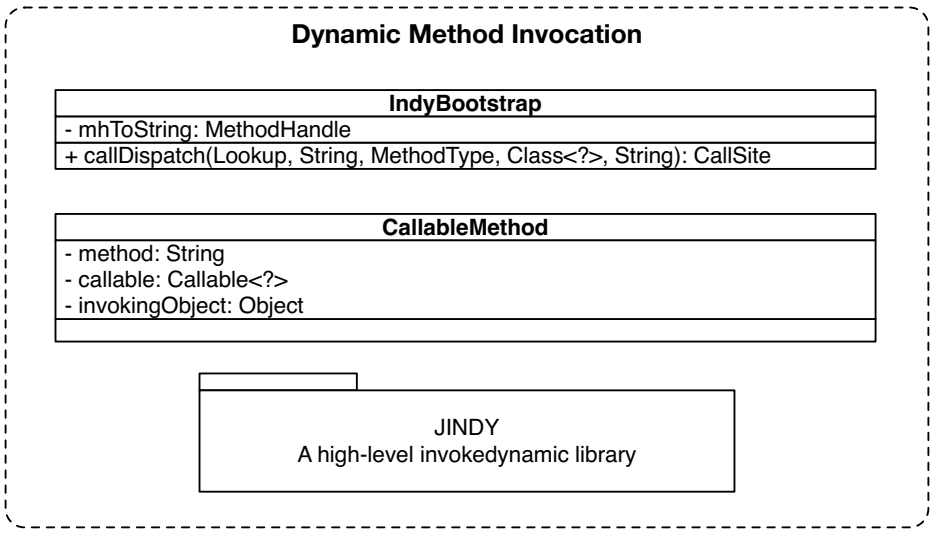
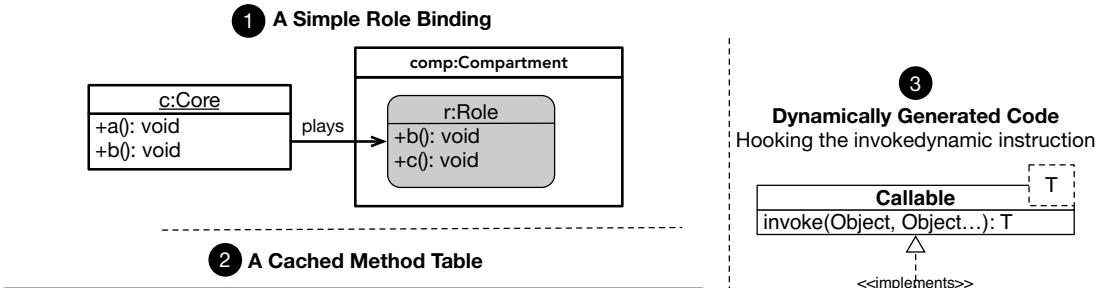


Figure 5.3: A class diagram of dynamic method invocation.

[1]Java Stream API accessed on July 15, 2017: `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

5.3.1 A Cached Method Table

All defined methods of a core object are sliced and stored in a cached method table with a `ConcurrentHashMap<Integer, CallableMethod>` data structure. The `Integer` is a hashcode of a method signature denoted as a key whose value is a `CallableMethod` data type reflecting the execution model of the method. The `CallableMethod` class contains a `Callable<?>` interface and `invokingObject` referring to either a core object or a role (Figure 5.4 ②). The `Callable<?>` interface has a concrete `CallableImp` class dynamically generated at run time (Figure 5.4 ③). The function of the `CallableImp` is to link a symbolic method name and its concrete implementation by the `invokedynamic` opcode instruction that will be explained in Section 5.3.2.



Method (Hashcode)	CallableMethod (Callable<?>, invokingObject)
a (1111)	Callable<?>, c:Core
b (2222)	Callable<?>, r:Role
c (3333)	Callable<?>, r:Role

Figure 5.4: A process to build a cached method table for dynamic invocation.

When the core object is bound to a role type, the relation is created in the lookup table. A role object is instantiated and its declared methods are sliced and stored in the cached method table. If the role object contains methods with the exact same signature as those of the core object, it overrides the existing one of the core. As a result, the invocation of the core object's method is delegated to that of the role object. If the signatures are not matching, the core object can invoke the role methods by providing the concrete method signature which is then translated to a hashcode that can be looked up in the method table in order to find the appropriate receiver.

Figure 5.4 shows a simple binding between a core and a role as illustrated in ①. There are only three methods (`a`, `b`, and `c`) stored in the cached method table because method `b` of the core object is overridden by method `b` of the role specified by the lifting mechanism illustrated in Section 4.3.5. The associated `Callable<?>` interface contains the `invokedynamic` instruction for invocation.

The cached method table is readjusted when there is a change in the role binding process. The procedure explained above not only conforms to the lifting function but also applies to the lowering and the collaboration functions as described in Section 4.3.5. Each function has its own cached method table, and all of them are stored in each compartment instance.

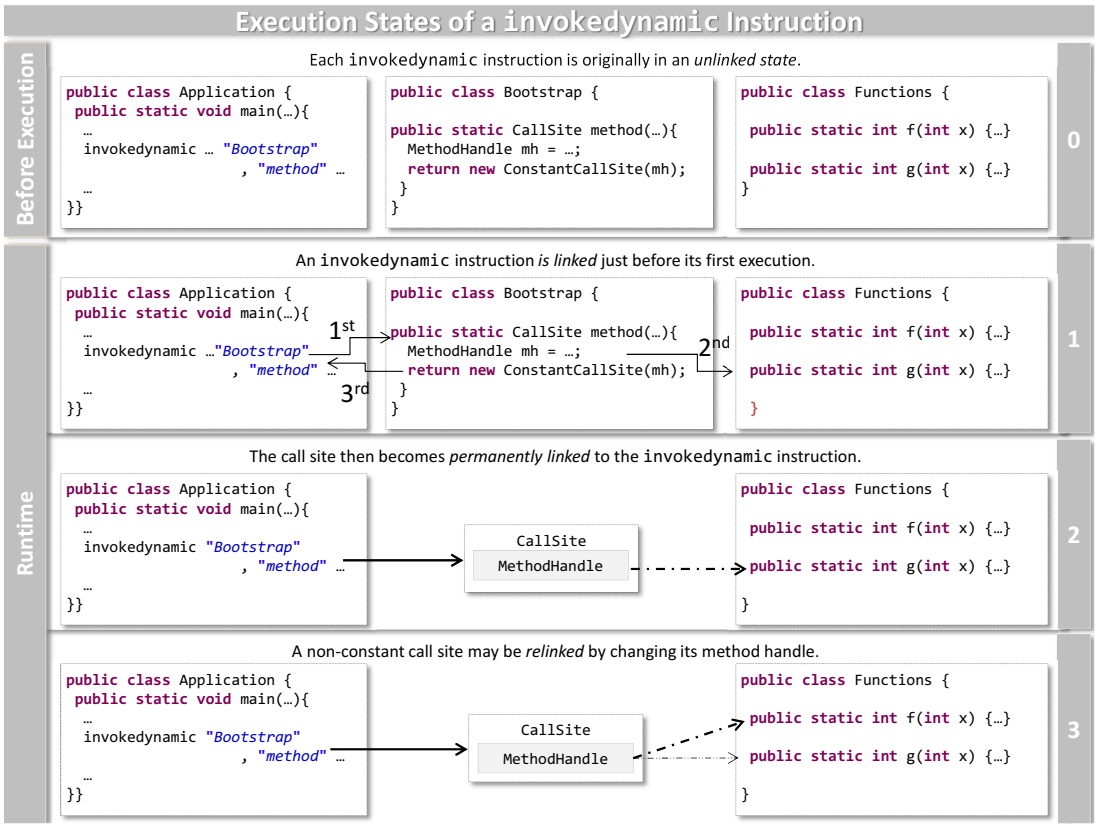


Figure 5.5: Run-time execution of the `invokedynamic`, an excerpt from JINDY [CO14].

5.3.2 The `invokedynamic` opcode instruction

Since Java 7, the `invokedynamic` opcode instruction has been added to the JVM in order to provide a new dynamic linking and to shift the type checking of the method invocation to run time [CO14]. Once the link is established, the JVM performs a common optimization in order to provide a better run-time performance compared to the reflection [CO14].

Figure 5.5 shows the run-time execution of `invokedynamic` operation. First (Step 0), the method to be invoked in the `Application` class is passed as a string and its parameters are optional. This method is in *unlinked* state that it supposes to call the `Bootstrap` class returning a `CallSite`. The `CallSite` contains a `MethodHandle` of a target method to be resolved at run time (i.e., either `f` or `g` method of the `Functions` class). During execution (Step 1), an `invokedynamic` instruction is *linked* just before the first execution. After that (Step 2), the link becomes permanent. The JVM then can optimize that link in order to improve run-time performance. If necessary, the link can be reestablished to another method for adaptation (Step 3). A further detailed illustration can be found in JINDY [CO14].

Despite the optimistic performance gain, the support of the `invokedynamic` for high-level application is infeasible for a typical programmer due to the lack knowledge of bytecode specification. The bytecode, i.e., `Callable<?>`, must be regenerated at run time in order to hook the `invokedynamic` opcode instruction into the method attaching to the specific `CallSite`. For the sake of simplicity, we use JINDY [CO14] library which offers a high-level support of the `invokedynamic` instruction relying on the efficient dynamic bytecode

generation library ASM [BLC02]. JINDY proved that the overhead of the `invokedynamic` is minimal compared to the static method invocation.

Snippet 5.1 shows the implementation of a generic `IndyBootstrap` class which is associated with each method of a core object or a role (Line 3 of Snippet 5.2). The implementation of the `Callable<?>` interface of each method is dynamically generated by JINDY's `ProxyFactory.generateInvokeDynamicCallable` method as depicted in Line 6 of Snippet 5.2. Each method is registered in the cached method table stored in each compartment instance as illustrated in Snippet 5.3.

Snippet 5.1: An implementation of the `invokedynamic` bootstrap that returns `ConstantCallSite`.

```
1 public class IndyBootstrap {
2   private static MethodHandle mhToString = null;
3
4   public static CallSite callDispatch(MethodHandles.Lookup lookup, String
        name, MethodType methodType, Class<?> paramClass, String methodName)
        throws Throwable{
5
6     mhToString  = lookup.findVirtual(paramClass, methodName, methodType.
          dropParameterTypes(0, 1));
7
8     return new ConstantCallSite(mhToString);
9   }
10 }
```

Snippet 5.2: Generating a `Callable<?>` of the `invokedynamic` for a particular method of role. This snippet is a method implementation in the `Registry` class.

```
1 private <T> Callable<T> getCallable(Class<T> clazz, Method method) {
2   try {
3     Bootstrap bootstrap = new Bootstrap(Cache.Save, "net.lyrt.IndyBootstrap",
          "callDispatch", clazz, method.getName());
4     MethodSignature sig = new MethodSignature(method.getReturnType(), clazz,
          method.getParameterTypes());
5
6     Callable<T> callable = ProxyFactory.generateInvokeDynamicCallable(
          bootstrap, sig);
7
8     return callable;
9   } catch (Throwable t) {
10    t.printStackTrace();
11  }
12
13  return null;
14 }
```

Snippet 5.3: Registering a `Callable<?>` of the `invokedynamic` instruction into the method table of the lifting function. This snippet is a method implementation in the `Registry` class.

```
1 private void registerLiftingCallable(Compartment compartment, Object core,
      Object role, Class<?> roleType){
2
3   Method[] methods = roleType.getDeclaredMethods();
4
5   for (Method method : methods) {
6     Callable<?> c = getCallable(roleType, method);
```

```
 7     MethodSig ms = getMethodSig(method);
 8     String m = String.format("%d:%s", core.hashCode(), ms.toString());
 9
10     CallableMethod cm = new CallableMethod(m, role, c);
11
12     compartment.liftingCallable.put(m.hashCode(), cm);
13   }
14 }
```

5.4 Dynamic Class Reloading

Support for unanticipated adaptation does not require any change on the overall implementation as the dynamic instance binding mechanism was proposed for this purpose. The remaining requirement is to load and reload the role type or class dynamically at run time. Once the role class is loaded into the JVM, it is initialized and its methods can be cached in the existing cached method table explained Section 5.3. As a result, the later invocation from a core object will be dispatched to the newly loaded role instance.

By default, Java allows a class to be dynamically loaded at run time. However, once loaded, the class template is cached in the JVM. Subsequent changes are prohibited, thus, classes cannot be reloaded and the implementation cannot be changed at run time. In order to overcome this limitation, we provide a dynamic class reloader (`ClassReloader`), which extends the default class loader, provided by Java. In contrast to the default class loader that reads from the cache, our class reloader reads the class's bytecode every time when it is loaded. Thus, the new implementation is reflected. This technique is adopted in all major software frameworks, such as Spring[2], that needs the dynamic class reloading capability. The class diagram related to this section can be found in Figure 5.6.

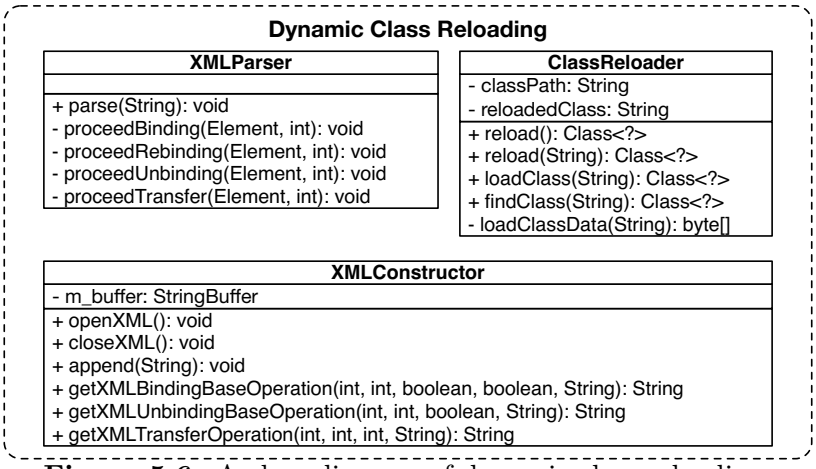


Figure 5.6: A class diagram of dynamic class reloading.

Snippet 5.4 shows the implementation of the dynamic `ClassReloader`. In Line 22 the bytecode of the role class is read by the `loadClassData` method (Lines 37-51). In Line 23, the loaded bytecode is redefined into a class using the `defineClass` method of the super class' `ClassLoader`.

[2]Spring framework accessed on July 15, 2017: `https://spring.io`

Snippet 5.4: An implementation of the dynamic class reloader.

```
1 public class ClassReloader extends ClassLoader {
2     String classPath = "target" + File.separator + "test-classes";
3     String reloadedClass;
4
5     public ClassReloader(){}
6
7     public ClassReloader(String classPath){ this.classPath = classPath; }
8
9     public Class<?> reload(){ return loadClass(reloadedClass); }
10
11    public Class<?> reload(String clazz){
12        reloadedClass = clazz;
13        return reload();
14    }
15
16    @Override
17    public Class<?> loadClass(String s) { return findClass(s); }
18
19    @Override
20    public Class<?> findClass(String s) {
21        try {
22            byte[] bytes = loadClassData(s);
23            return defineClass(s, bytes, 0, bytes.length);
24        } catch (IOException ioe) {
25            try {
26                return super.loadClass(s);
27            } catch (ClassNotFoundException ignore) { }
28            ioe.printStackTrace(System.out);
29            return null;
30        }
31    }
32
33    public Class<?> loadClass(Class<?> clazz){
34        return loadClass(clazz.getName());
35    }
36
37    private byte[] loadClassData(String className) throws IOException {
38        String userDir = System.getProperty("user.dir");
39        String fullDir = userDir + File.separator + classPath + File.
              separator;
40        File f = new File(fullDir + className.replaceAll("\\.", File.
              separator) + ".class");
41
42        if(!f.exists()) throw new IOException(); //force to use super (
              ClassLoader) to load class
43
44        int size = (int) f.length();
45        byte buff[] = new byte[size];
46        FileInputStream fis = new FileInputStream(f);
47        DataInputStream dis = new DataInputStream(fis);
48        dis.readFully(buff);
49        dis.close();
50        return buff;
51    }
52 }
```

The ability to reload the class at run time may affect the semantics of the already running instances whose class template has changed. Since LyRT operates on the instance level and

the bound role is conceptually considered a part of the core object in which the role instance is never directly accessed by the programmer, this problem has no effect on our concept and implementation.

For example, `Alice` and `Bob` are the instances of a `Person` class denoting core objects. Both of them play the same `Student` role type with different instances, `student1` and `student2`. At run time, the `Student` role type's implementation is changed and changes should only apply for `Alice` but not for `Bob`. In this scenario, the programmer must provide the new binding to the new implementation of the `Student` role for `Alice` through an XML configuration (see Section 4.3.8 of Chapter 4). As a result, a `student3` instance with the new implementation is initialized and bound to `Alice` while the existing `student2` role, which is still bound to `Bob`, remains the old one.

5.5 Block Constructs

According to our concept described in Chapter 4, there are three kinds of block constructs supported in LyRT—`InitBindingBlock`, `AdaptationBlock`, and `ConsistencyBlock`. These blocks rely on the `AutoCloseable` interface, which is known as *try-with-resources* available since Java 7. It is a specialized exception handling technique, which automatically closes the opened resources, such as files, databases, sockets, when the `try-catch` block reaches its end. The class diagram associated to this section can be found in Figure 5.7.

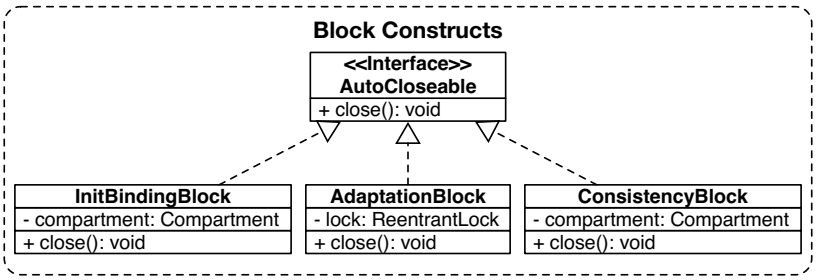


Figure 5.7: A class diagram of the block constructs.

5.5.1 The `InitBindingBlock` Construct

The `InitBindingBlock` is used only one time to bind certain roles to cores immediately after a compartment object is initialized. LyRT performs role binding at run time and roles must be virtually resided in a compartment object. The implementation of the `InitBindingBlock` is shown in Snippet 5.5. Since roles can only be bound in an active compartment, when the block is initialized, the passing compartment object reference is activated and deactivated when the block has expired as described in the overridden `close` method (Lines 10-12).

Snippet 5.5: `InitBindingBlock` implementation.

```
1 public class InitBindingBlock implements AutoCloseable{
2     Compartment compartment;
3
4     public InitBindingBlock(Compartment compartment){
5         this.compartment = compartment;
6         this.compartment.activate();
7     }
```

```
8
9       @Override
10      public void close() {
11          compartment.deactivate(false); //No need to do check point
12      }
13  }
```

In order to use the `InitBindingBlock`, a compartment object is required because it is embedded in the `initBinding` method of a compartment. The implementation of this method returns a `new InitBindingBlock(this)` where `this` is the compartment object. Similar to the usage of the `try-catch` block, developers need to provide the construct as shown in Lines 7-9 in Snippet 5.6. Note that there is no catch block required.

Snippet 5.6: `InitBindingBlock` usage.

```
1 //Main code. Initializing a compartment and a core.
2 Registry reg = Registry.getRegistry();
3 Compartment compartment = reg.newCompartment(Compartment.class);
4 Person p = reg.newCore(Person.class);
5
6 //Init binding block
7 try(InitBindingBlock ib = compartment.initBinding()){
8   p.bind(Student.class);
9 }
```

5.5.2 The `AdaptationBlock` Construct

The `AdaptationBlock` provides a mechanism to change roles from cores within an active compartment. Unlike the `InitBindingBlock`, which is called immediately after the compartment object is initialized, the `AdaptationBlock` can be called several times responding to the required adaptations happening over time. The implementation of this block also relies on the try-with-resources. The important part of this implementation is to perform the checkpoint process of current application configuration before transferring to the new one (see Section 5.6).

When the block has expired, the `close` method is called to check whether the adaptation contains defective roles which cause the runtime to crash. In that case, the adaptation is canceled meaning that the cached method table is reverted because binding and unbinding roles change the data in the cached method table. The implementation of the `AdaptationBlock` is shown in Snippet 5.7.

Snippet 5.7: `AdaptationBlock` implementation.

```
1  public class AdaptationBlock implements AutoCloseable{
2      public AdaptationBlock(){
3        Registry reg = Registry.getRegistry();
4          Compartment compartment = reg.getActiveCompartment();
5          if (compartment == null) throw new RuntimeException("No active
               compartment was found");
6
7          ControlUnit.checkpoint(compartment);
8      }
9
10     @Override
11     public void close(){
```

```
12     if(ControlUnit.hasDefectiveRoles()){
13       Registry.cancelAdaptation(this);
14     }
15     }
16 }
```

The usage of the AdaptationBlock requires a call inside an active compartment. Snippet 5.8 depicts two adaptations which require a core person to bind to different roles by using two AdaptationBlocks.

Snippet 5.8: AdaptationBlock usage.

```
1 compartment.activate();
2
3 //Adaptation requires a person to bind to new role
4 try(AdatpationBlock ab = new AdaptationBlock()){
5   person.bind(Professor.class);
6 }
7
8 //Another adaptation of the person
9 try(AdaptationBlock ab = new AdapationBlock()){
10   person.bind(CEO.class);
11 }
12
13 compartment.deactivate();
```

5.5.3 The ConsistencyBlock Construct

The ConsistencyBlock is used in order to prevent the behavior of core objects from being changed while executing. It is essential to detect an active compartment and to register a consistency object associated with the active compartment and a starting timestamp (Snippet 5.9). This timestamp is required to prevent the sliced methods of the newly bound roles from being registered in the cached method table. In other words, the method table of each core object is maintained and thus the behavioral consistency of the cores is ensured.

In each thread, there is only one ConsistencyBlock running anchored with the compartment object. Therefore, multiple ConsistencyBlocks execute in parallel in a multi-threaded environment. Those ConsistencyBlocks are stored in a ConcurrentHashMap<Integer, SimpleEntry<Integer, LocalDateTime>> data structure where the first Integer refers to the associated compartment hashcode and the SimpleEntry<Integer, LocalDateTime> is a key-value pair of the hashcode of the consistency object and the starting timestamp of the ConsistencyBlock.

Any new role binding or unbinding occurring during the presence of the ConsistencyBlock will be marked with the timestamp attributed to BoundTime and UnboundTime respectively in the Relation data type. Therefore, when the ConsistencyBlock expires as defined in the close method of Snippet 5.9, the cached method table is adjusted with respect to the bound or unbound roles. Furthermore, the registered consistency object is also removed by calling removeConsistencyBlock of the Registry class.

Snippet 5.9: `ConsistencyBlock` implementation.

```
1 public class ConsistencyBlock implements AutoCloseable {
2     private Registry reg;
3     private Compartment compartment;
4
5     public ConsistencyBlock(){
6         reg = Registry.getRegistry();
7         compartment = reg.getActiveCompartment();
8         if(compartment==null) throw new RuntimeException("No active
              compartment was found");
9
10        LocalDateTime time = LocalDateTime.now();
11        reg.registerConsistencyBlock(compartment, this.hashCode(), time);
12    }
13
14    @Override
15    public void close() {
16        reg.removeConsistencyBlock(compartment, this.hashCode());
17    }
18 }
```

Snippet 5.10 demonstrates how the `ConsistencyBlock` is used for a `Channel` object of the file transfer application, explained in Chapter 2, in which the behavior of the `Channel` object must not be changed while sending file chunks. Any adaptation happening asynchronously and changing the behavior of the channel object is not permitted, but the adaptation will be performed as soon as the `ConsistencyBlock` expires for the current executing thread.

Snippet 5.10: `ConsistencyBlock` usage.

```
1 compartment.activate();
2
3 //Consistent behavioral invocation
4 try(ConsistencyBlock cb = new ConsistencyBlock()){
5   while(EOF(file)){
6     channel.invoke("send", getChunk(file));
7   }
8 }
9
10 compartment.deactivate();
```

5.6 Rollback Recovery

Checkpoint and rollback are developed inside a `ControlUnit` class containing two additional functions. First, the `reportBugs()` method logs the bugs found in a configuration for developers to fix. Second, the `hasDefectiveRoles()` method checks and prevents the defective roles from being reactivated. We attach the checkpoint execution to the implementation of the `AdaptationBlock` mentioned in Section 5.5.2, while the rollback process is handled in the bug sensor implementation. A class diagram related to this process can be found in Figure 5.8.

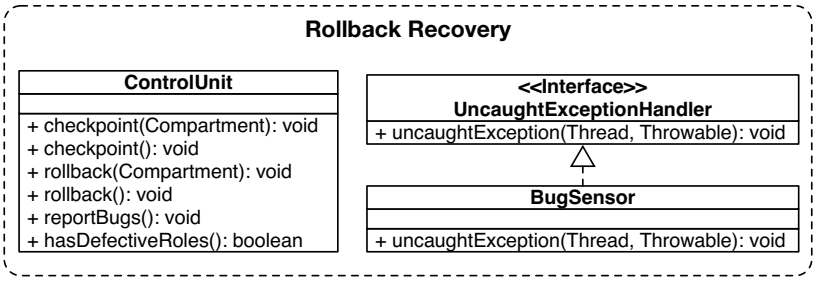


Figure 5.8: A class diagram of rollback and recovery component.

5.6.1 Checkpoint and Rollback

Serialization can be used to implement a checkpoint. In Java, serialization requires to implement the `Serializable` interface. However, not all objects can be serialized, such as file pointer, socket, database connection. Due to backward compatibility, performance is a serious issue in the Java serialization library. Therefore, we use a third-party serialization library called Kryo[3]. Kryo is a fast and reliable object graph serialization framework for Java that provides many benefits compared to the default Java serialization. First, the serializing object does not require to implement the `Serializable` interface. Second, it can perform a both deep and shallow copying. Third, it is incorporated in many well-known open source projects such as Apache Spark[4] and Storm[5].

In this prototype, we use a copy method in order to build checkpoints by copying the relations and role instances in the lookup table and pushing them to the stack. The deep copy ensures the role instances and their dependencies to be copied by using direct assignment from object to object, opposed to the serialization that transforms from object to bytecode and from bytecode to object. As a result, the copying technique consumes more memory for large objects, but it is faster than the serialization. Kryo supports both techniques. We use a `HashMap<Long, Stack<ArrayDeque<Relation>>>` data structure in order to capture the saved checkpoints of a particular thread. The key (`Long`) is a thread identity while the `Stack<ArrayDeque<Relation>>` holds a stack of multiple checkpoints of the lookup table represented as an `ArrayDeque<Relation>` as explained earlier in Section 5.2.

In order to roll back, we first remove the current configuration (i.e., binding information and role instances) from the lookup table (`ArrayDeque<Relation>`). Then, we pop the recent checkpoint from the stack and reinstate it in the lookup table. Finally, the cached method table is reevaluated.

5.6.2 Utilizing an Exception Handling for the Bug Sensor

By taking advantage of the application-level exception handling, we can hook in an exception handler for all threads, a group of threads, or a particular thread in order to intercept

[3]Kryo accessed on July 16, 2017: `https://github.com/EsotericSoftware/kryo`

[4]Apache Spark accessed on July 16, 2017: `http://spark.apache.org`

[5]Storm accessed on July 16, 2017: `http://www.storm-project.net`

the raising error for handling a recovery process. The bug sensor implements the `UncaughtExceptionHandler` interface by overriding the default method, i.e., `uncaughtException()`, where the logic of the recovery process takes place. We can then set the bug sensor for any uncaught exception. Snippet 5.11 depicts the bug sensor implementation which is installed at the beginning of the main code as shown in Line 4 of Snippet 5.12, or attached to each thread (Line 7 of Snippet 5.12).

Snippet 5.11: Bug sensor implementation.

```
1 public class BugSensor implements Thread.UncaughtExceptionHandler {
2   private Socket client;
3
4   public BugSensor(Socket client){
5     this.client = client;
6   }
7
8   @Override
9   public void uncaughtException(Thread t, Throwable e) {
10    Registry reg = Registry.getRegistry();
11    Compartment comp = reg.getActiveCompartments().get(t.getId());
12
13    //Report bugs
14    ControlUnit.reportBugs(comp, e);
15
16    //Rollback
17    ControlUnit.rollback(comp);
18
19    //Restart the client thread
20    ServiceHandler handler = new ServiceHandler(client);
21    handler.start();
22  }
23 }
```

Snippet 5.12: Bug sensor installation.

```
1 public static void main(String... args){
2   //Globally attach the bug sensor
3   BugSensor sensor = new BugSensor(client);
4   Thread.setDefaultUncaughtExceptionHandler(sensor);
5
6   //Locally attach the bug sensor to specific thread
7   Thread.currentThread().setUncaughtExceptionHandler(sensor);
8
9   //The rest of program
10 }
```

The current implementation of the bug sensor catches only the uncaught exceptions which are subclasses of the `RuntimeException`, such as `DivideByZeroException`, `ArrayOutOfBoundException`, etc., raised by the JVM. For the caught exceptions such as `IOException`, there are two options to handle and detect them with the bug sensor. First, application developers need to manually rethrow all the caught exceptions so that the JVM can perform a stack trace of the caught exceptions until it reaches the exception block in the bug sensor. The second option requires no application developers' involvement, but still, it needs to rethrow caught exceptions. Exception rethrowing can be injected automatically during class loading by using a bytecode rewriting framework such as ASM [BLC02]. For the current implementation, we chose the first option.

The effectiveness of the recovery process lies under the power of the bug sensor. Our current implementation relies on the exception handling mechanism to detect the uncaught and caught exception handling at the application level. Nonetheless, the JVM-related issues such as `OutOfMemoryError` that shuts down the JVM cannot be handled. In this regard, implementing a bug sensor at the JVM level could be more efficient, but it may face compatibility issues.

If an exception is raised in a particular thread, that thread is killed. Although respawning the thread is purposeful in the bug sensor implementation, some service disruptions are unavoidable during the thread restarting process. We leave this problem to the application developers for customizing the bug sensor to handle an application-specific recovery process.

5.7 Compartment, Role, and Core Object

In order to implement role-specific functionality in LyRT, programmers have to rely on the given `Compartment` class, `IRole` and `IPlayer` interfaces. `IPlayer` and `IRole` are Java interfaces containing default methods. Normally, Java does not allow to have a concrete implementation in the interface, but this is possible in Java 8[6]. A method with concrete implementation is called *default method* which must be defined with a `default` keyword. The default method shares similar semantics with static traits [BMN14], and allows us to add role operations easily.

To be a core object, a user-defined class must implement the `IPlayer` interface making it capable of performing role operations such as binding, unbinding, and method invocation. A method to be invoked is given as a `String` and matches a method signature stored in the cached method table. These player's functionalities are implemented to call the associating `Registry`'s methods comprising the respective logics. A class diagram showing a relationship between compartment, role and object can be found in Figure 5.9.

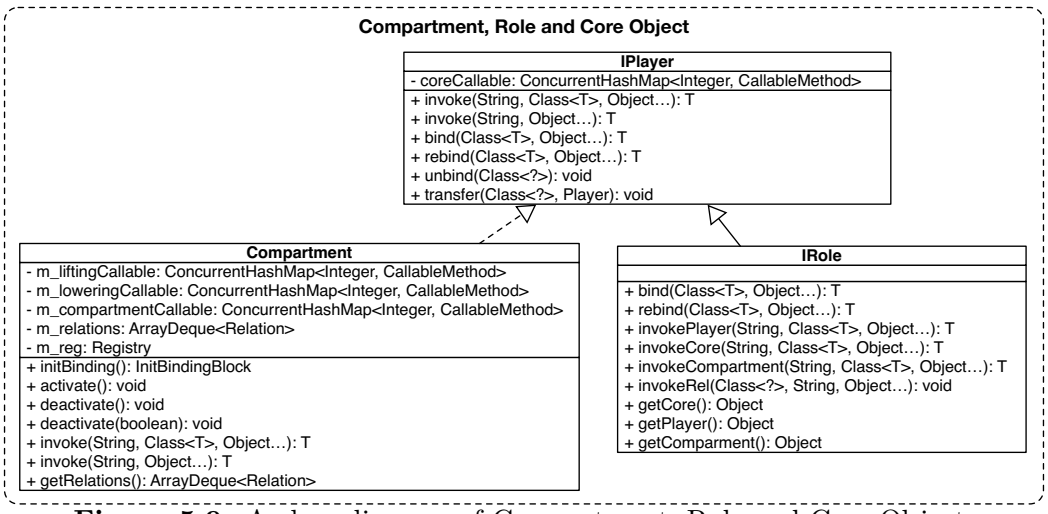


Figure 5.9: A class diagram of Compartment, Role and Core Object.

The `IRole` interface extends the `IPlayer` with additional binding and rebinding operations

[6]An official discussion is available at: `https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`, accessed on July 17, 2017

in order to support the *deep-play*-relation. The `IRole` interface contains also default methods dealing with lifting, lowering (i.e., `invokeCore` and `invokePlayer`), and collaboration (i.e., `invokeCompartment` and `invokeRel`) functions as explained in Section 4.3.5. Besides, we can add additional methods in order to get a reference of the core, player or even compartment objects with `getCore`, `getPlayer`, and `getCompartment` method respectively. Similar to `IPlayer`, the role's default methods then call the respective `Registry`'s functions. In order to be a role, a user-defined class needs to implement this `IRole` interface.

Contrarily, `Compartment` is a class implementing the `IPlayer` interface in order to play roles. A user-defined compartment has to be a subclass of this `Compartment` class. As explained earlier in Section 5.3, each compartment instance contains different cached method tables, whose data structure is a `ConcurrentHashMap<Integer, CallableMethod>`, for lifting, lowering and collaboration functions as well as for managing its own lookup table with `ArrayDeque<Relation>`. Having separate tables greatly minimizes the global locking in the entire application when executing in a multi-threaded environment. Thus, the performance is improved. However, the `ConcurrentHashMap` is still used because there are cases in which different roles in the same compartment instance are executed concurrently.

5.8 Decision to Use Java

The reasons why Java was used as a host language for the LyRT prototype are twofold. On the one hand, we want to tackle the challenge of implementing adaptive behavior in a strongly and statically typed language. Hence, if we can demonstrate that the concept of LyRT can be developed in Java, it may be feasible for other languages as well. Evidently, any list data structures can be used to represent a lookup table. Similarly, a cached method table can be implemented in any dictionary data structure. Instead of `invokedynamic` opcode, reflection is always a last resort. For consistency, checkpoint, and rollback, they require no additional toolkits. Implementing a bug sensor is also viable since these dynamic languages fully support exception handling. On the other hand, Java is still one among the most popular languages in industry and academia. Furthermore, most of the existing ROPs are based on Java although some of them are no longer accessible at the moment.

Through code snippets, we have shown that the method invocation is passed by a string to avoid statically type checking at compile time. Yet, this technique limits code readability. Code readability can be exceedingly improved if LyRT is implemented in those dynamic languages (e.g., Ruby, Python, Javascript, etc.) because they delay the type checking until run time. Their open implementation further enhances the vocabularies of the framework. For example, our `try-block` can be cleanly replaced by a `with` statement in Python. Additionally, dynamic method dispatch can be implemented by hooking an interception logic in a `method_missing(method, *args, &block)` in Ruby. Python offers a similar function by defining inside a `__getattr__(self, name)` method. In Java, we can develop an interpreter or a compiler for language extension in order to enhance code readability. However, the resulting development environment would lack the tool support such as IDE and debugging tool. Since developing a new language is out of scope, LyRT is prototyped as a library for better integration into the mainstream language. LyRT is publicly available at `https://github.com/nguonly/lyrt`.

5.9 Chapter Summary

In this chapter, the most important parts of the LyRT implementation were described. Java is the chosen host language for exercising our concept in order to demonstrate the generality of LyRT which does not depend too heavily on the host language's features. The `ArrayDeque<Relation>` is used to implement the lookup table, while manipulating role relations relies on the `java.util.stream` API. The relations in the lookup table serve as information for building a cached method table for dynamic dispatch containing the new `invokedynamic` opcode instruction for efficient method invocation. We hope that this technique can minimize the runtime overhead of both searching a method and its invocation. The `InitBindingBlock`, `AdaptationBlock`, and `ConsistencyBlock` constructs are implemented with a *try-with-resource* construct available since Java 7.

The checkpoint is implemented per thread with a direct copy of the lookup table `ArrayDeque<Relation>` and is pushed to the stack `Stack<ArrayDeque<Relation>>`. In order to roll back, a copied `ArrayDeque<Relation>` is popped from the stack and is restored to the runtime. Despite limitations, application exception handling is adopted for initial implementation of the bug sensor. In the next chapter, we will evaluate LyRT based on this prototype as a proof of concept.

Chapter 6

Evaluation

This chapter demonstrates the practical feasibility of LyRT in terms of the requirements of run-time variability, set in Chapter 2, and performance. The evaluation is divided into two main parts. First, three case studies will be presented to systematically assess the criteria of the requirements. With these case studies, this dissertation further analyses features supported in LyRT against the classifying role feature list compiled by Kühn et al. [KLG+14]. Since CROM [KLG+14] influences the design of LyRT, we also discuss the relation between the two. Second, several micro-benchmarks are set up to critically evaluate the overhead of the runtime in terms of execution time.

6.1 Case Studies

With respect to our research questions asked in Chapter 1, we derived five requirements and added another one by observing the relationship between the high availability property and software deployment cost in Chapter 2. These requirements will be our primary validation points used to systematically assess the case studies. Let us briefly revisit those requirements:

R1: Modularity. Adaptive entities, also known as variants, should be declared separately from the base system and from each other in order to have a clean application logic.

R2: Dynamic Activation. The coupled variants can be functional by means of activation. Deactivation of the activated variants should be supported.

R3: Late Variants Adoption. As applications keep changing, foreseeing all possible variants beforehand is impossible. Unknown variants should be adopted at run time and dynamically activated in order to provide an additional unforeseen functionality.

R4: Object-Level Tranquility. Adaptation can be harmful if it happens at the wrong time. Having a proper mechanism to handle a consistent behavior of an object as a precaution is necessary.

R5: Failure Handling. While activation of a single variant executes flawlessly, variants that interact with each other may result in a bug and cause run-time failures.

R6: Continuous Deployment. This requirement is derived from the support of R3 and R5. The defective variants, which are captured (R5), should be incorporated to the runtime once again through the late adoption capability (R3) when they are fixed.

6.1.1 Validation Setup

Although the validation of the expressiveness of a system is not trivial [Fel90], we have developed a validation approach as systematically as possible. Table 6.1 shows how the case studies address the requirements. Additionally, these case studies are developed and targeted in different application domains to demonstrate LyRT's applicability. They are summarized as follows:

Tax Management System is a console application which intends to demonstrate the modularity and adaptability at the instance level.

Snake Game is a classic game with a Graphical User Interface (GUI) support that mainly targets unanticipated adaptation.

File Transfer Application is a client-server application executing in a multi-threaded environment that primarily focuses on consistent behavior and error handling. This case study focuses also on the derived feature of continuous software deployment.

Table 6.1: Case studies and their intended validating points. Both ■ and ⊞ symbols are denoted as requirement satisfaction, but the intended validation points are denoted by this ■ symbol.

Requirement	Tax Management System	Snake Game	File Transfer Application
R1	■	⊞	⊞
R2	■	⊞	⊞
R3		■	⊞
R4			■
R5			■
R6			■

The case studies share a common structure for discussion:

- A scenario describing a typical use of an adaptive software system.
- A description of expected functionality derived from the scenario.
- A description of the implementation highlighting the required functionality.
- A validation on how the case study fulfills the requirements shown in Table 6.1.

Limitations on the validation. The validation should be performed without bias. In order to do so, we could have implemented the same case studies in a comparable approach and drawn a comparative study. Unfortunately, we cannot follow this step because of two main reasons. On the one hand, there are no comparable approaches, which support all requirements as discussed in Chapter 3. Most of the existing approaches tackle only requirement R1 and R2. The state-of-the-art analysis was summarized in Table 3.9 in Chapter 3. On the other hand, doing a comparative study can be easily subjective and biased in the experiment due to the knowledge of the selected approaches and the case studies [Gon08, p. 141].

6.1.2 Tax Management System

This case study has been described in the concept chapter (Section 4.3.2) as an example to discuss the dynamic instance binding mechanism. Again, we use this example, with a slight change to demonstrate the modular design and the dynamic activation to achieve adaptation at instance level. This case study is motivated by the following scenario:

> A software firm is developing an integrated salary and tax payment system for companies, startups, and the state tax department. The main goal is to keep track of the tax payment of individuals which is based on their revenue. The tax for a company employee is calculated into the company revenue; so the company will pay the tax as a whole which accounts for 20% of its net income, while a self-employed person pays only 10%.
>
> The company generates revenue by the performance of its staff. Apart from day-to-day business, accountants are responsible for the monthly salary payment to all employees including themselves. The received salary is a net income and stored in the person's saving. Self-employees or freelancers manage their income and pay tax by themselves. A person who was previously an employee can be a freelancer. The system should be able to calculate the correct amount of tax for this person.

Besides showing dynamic activation, this case study also serves as a tutorial for implementing role-based applications using LyRT prototype.

6.1.2.1 Expected Functionality

Before discussing the code implementation, practical functionalities are highlighted. Figure 6.1 depicts the run-time model with the following functionalities:

Adaptability. There are several instances of a `Person` class, fulfilling different roles including `Developer`, `Accountant`, `Freelancer`, `TaxPayer`, and `TaxCollector`. These roles represent the respective behaviors to be performed by the engaging person. For example, while developers work to generate revenue, accountants issue a salary payment. For any reason, an `ely` instance, formerly a `Developer`, resigns from a company `abc` and starts as a `Freelancer`. As a result, `ely`'s behavior adapts accordingly.

Monthly Salary Generation. At the end of each month, `accountants` invoke a `paySalary` method in order to distribute the salary to respective persons employed in each position. The salary is a net income by withdrawing from a company revenue and is aggregated to the person's saving. The individual employee's income tax (20%) is withheld by the company, which then pays the overall tax for all employees.

Tax Collection. Both a company and a freelancer have to pay taxes at different rates. A `TaxCollector` collects taxes by invoking a `collectTax` method. This method looks for appropriate `TaxPayers` obligated to pay taxes. The collected taxes are added to the revenue of a `TaxDepartment`.

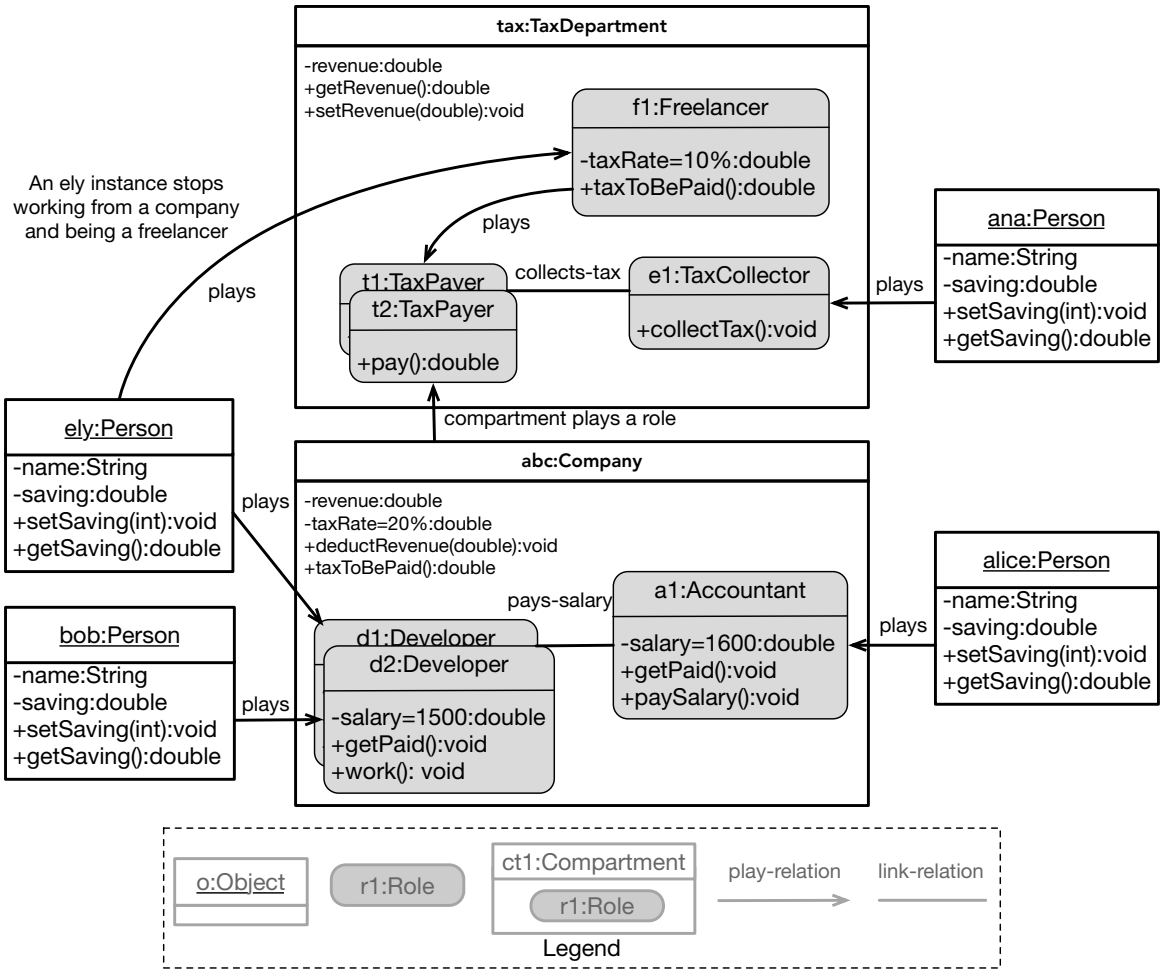


Figure 6.1: A class diagram for the Tax Management System, based on CROM notation. This diagram was illustrated once in Figure 4.5 to explain the concept of the dynamic instance binding mechanism.

6.1.2.2 Implementation

This section starts with discussing the definition of the relevant compartment, role and core object implementations. Afterwards, we explain how these objects are crafted in order to achieve dynamic behavioral activation.

Core Object. A `Person` class, illustrated in Snippet 6.1, is the main core object which will bind to different roles in order to perform certain tasks. Although it is not necessary, a core object is normally defined by implementing the `IPlayer` interface to hook into certain functionalities without relying on the `Registry` (see Section 5.2). The definition of the `Person` class contains two fundamental properties: `name` and `saving` with respective `get/set` methods. The `saving` property is used to store how much money that the person has earned.

Snippet 6.1: Implementation of the `Person` core object.

```
1 public class Person implements IPlayer{
2     private String name;
3     private double saving;
4
5     public Person(String name){ this.name = name; }
6
7     public String getName() { return name; }
8     public void setName(String name) { this.name = name; }
9
10    public double getSaving() { return saving; }
11    public void setSaving(double saving) { this.saving += saving; }
12 }
```

Developer Role. Similar to a core object, a role is defined by implementing the `IRole` interface. Snippet 6.2 shows the implementation of a `Developer` role which has a default salary of 1500 Euro. A `getPaid` method will be called by a person who plays an `Accountant` role allowing the person playing this `Developer` role to get monthly payment. This procedure will deduct the amount from the company revenue and will add the salary to the saved property of the person that binds to this role. Note that a company is implemented as a compartment (Snippet 6.7). The role calls a `paySalary` method of the company compartment using the `invokeCompartment` method. In order to save the paid salary to the person's saving attribute, a reference to its core object is required. A `getCore` method returns a core object's reference binding to this role (Line 8).

A `work` method simulates the revenue generation by calling a `addRevenue` method of a company compartment using the `invokeCompartment` method (Line 15). The revenue generated in each call is equivalent to the paid salary.

Snippet 6.2: Implementation of the `Developer` role.

```
1 public class Developer implements IRole{
2   private double salary=1500;
3
4   public Developer(Double salary) { this.salary = salary; }
5
6   public void getPaid(){
7     invokeCompartment("paySalary", salary);
8     Person core = (Person)getCore();
9     core.setSaving(salary);
10  }
11
12  public void work(){
13    IPlayer person = (IPlayer)getCore();
14    String name = person.invoke("getName", String.class);
15    invokeCompartment("addRevenue", salary); //increase company revenue
16    System.out.println(name + " generates a revenue of "+salary+" Euro");
17  }
18 }
```

Accountant Role. Being an employee, an `Accountant` role shares a similar `getPaid` method with the `Developer` role as implemented in Snippet 6.3. It has a `paySalary` method for the distribution of the salary to all persons based on the fulfilled position, i.e., `Developer`

and Accountant. With our collaboration functions (see Section 4.3.5), the invokeRel[1] method can be used to invoke the getPaid method of both the Developer and the Accountant role by means of a relationship between them as depicted in Figure 6.1. The invokeRel method triggers all the core objects who play the passing role type (i.e., Developer) in order to call the getPaid method (Lines 11-12).

Snippet 6.3: Implementation of the Accountant role.

```
1 public class Accountant implements IRole{
2   private double salary = 1600;
3
4   public void getPaid(){
5     invokeCompartment("paySalary", salary);
6     Person core = (Person)getCore();
7     core.setSaving(salary);
8   }
9
10  public void paySalary() {
11    invokeRel(Developer.class, "getPaid");
12    invokeRel(Accountant.class, "getPaid");
13  }
14 }
```

Freelancer Role. A Freelancer is a role that has to pay 10% of incoming tax independently. Snippet 6.4 shows its implementation. It collects revenue by invoking an earn method. Its taxToBePaid method which will be called by the TaxPayer role returns the amount of tax to be paid based on the current revenue and the tax rate.

Snippet 6.4: Implementation of the Freelancer role.

```
1 public class Freelancer implements IRole{
2     private double salary;
3     private double taxRate = 0.1; //10%
4
5     public void earn(double amount){ salary += amount; }
6
7     public double getMoney(){ return salary; }
8
9     public double taxToBePaid(){
10        double tax = salary * taxRate;
11        salary -= tax;
12        return tax;
13    }
14 }
```

TaxPayer Role. The TaxPayer role has the ability to calculate the correct amount of tax to be paid based on its players, either the Company compartment or the Freelancer role. According to its implementation in Snippet 6.5, it has a pay method which first queries the amount of the paid tax by the invocation of the taxToBePaid method (defined in both Company and Freelancer). The return value is accumulated to the revenue of the compartment (Line 6), which is the TaxDepartment.

[1] invokeRel refers to invoking a role's method participating in a relationship

Snippet 6.5: Implementation of the TaxPayer role.

```
1 public class TaxPayer implements IRole {
2   public void pay(){
3     IPlayer payer = (IPlayer)getCore();
4     double tax = payer.invoke("taxToBePaid", double.class);
5     String payerName = payer.invoke("getName", String.class);
6     invokeCompartment("addRevenue", tax);
7     System.out.println(payerName + " pays a tax of " + tax + " Euro");
8   }
9 }
```

TaxCollector Role. Snippet 6.6 presents the implementation of a TaxCollector role intended to be played by the Person core object. Its collectTax method invokes the pay method of all core objects binding to the TaxPayer role. Those core objects are the abc company compartment and ely who played the Freelancer role as shown in Figure 6.1.

Snippet 6.6: Implementation of the TaxCollector role.

```
1 public class TaxCollector implements IRole{
2   public void collectTax(){
3     invokeRel(TaxPayer.class, "pay");
4   }
5 }
```

Company Compartment. A compartment is defined by subclassing the Compartment class, given by LyRT. A Company class is coded in Snippet 6.7. The addRevenue method is called by the work method of the Developer role (Line 13 of Snippet 6.2) to generate a company revenue. Similarly, the paySalary method is called by the getPaid method of the Developer role to subtract the revenue for salary payment. The taxToBePaid method is used to calculate the tax amount to be paid with respect to the applied tax rate and current revenue.

Snippet 6.7: Implementation of the Company compartment.

```
1 public class Company extends Compartment{
2   private double revenue = 0;
3   private String name;
4   private double taxRate = 0.2; //20%
5
6   public Company(String name){ this.name = name; }
7
8   public String getName() { return name; }
9
10  public void addRevenue(double amount){ revenue += amount; }
11  public double getRevenue(){ return revenue; }
12
13  public double taxToBePaid(){
14    double tax = revenue * taxRate; //tax to be paid
15    this.revenue -= tax; //deduct from revenue
16    return tax;
17  }
18
19  public void paySalary(double amount){ revenue -= amount; }
20 }
```

TaxDepartment Compartment. A `TaxDepartment` is a compartment to scope another application block where the functionality of a tax payment exists. Its implementation, depicted in Snippet 6.8, defines `addRevenue` and `getRevenue` methods derived from the collected tax.

Snippet 6.8: Implementation of the `TaxDepartment` compartment.

```
1 public class TaxDepartment extends Compartment{
2     private double revenue;
3     private String name;
4
5     public TaxDepartment(String name){ this.name = name; }
6
7     public double getRevenue() { return revenue; }
8     public void addRevenue(double amount) { this.revenue += amount; }
9 }
```

Main Program. The main program is shown in Snippet 6.9 which resembles the run-time model depicted in Figure 6.1. The output resulting from the execution of this main program is self-explanatory and shown in Snippet 6.10. The following paragraphs explain the execution of the constructed main program.

Initialization of Compartment, Role and Core Object. The intended compartments and core objects can be instantiated by the `Registry` functions as shown on Lines 4-7 of Snippet 6.9. The initialization of roles is done implicitly in the binding process that takes place in the `InitBindingBlock` (Lines 10-14 and Lines 42-46 of Snippet 6.9).

Dynamic Activation. Conceptually, once bound, a role adapts or provides additional behaviors to its core object. To enable this feature, `abc`, an instance of the `Company` compartment, must be activated (Line 16) because it contains the role binding configuration as done in the `InitBindingBlock`. Then, `bob` and `ely`, instances of the `Person`, start working in order to generate a revenue for the company whose revenue initially was zero. This can be done by invoking a `work` method, defined in the `Developer` role. The company revenue gains 7000 Euro (2*1500 + 2*2000), resulting from four executions of the `work` method (Lines 18-21) and `bob` and `ely` having different salary scales (see implementation of `Developer` role in Snippet 6.2).

`Alice` then issues a monthly payment to all company's employees, who are `bob`, `ely` and herself, by executing the `paySalary` method of the `Accountant` role (Line 29). Despite the same core person, neither `bob` nor `ely` can invoke the `paySalary` method because they play the `Developer` role. The produced output is shown in Snippet 6.10. After invoking the `paySalary` method, the balance of the company revenue is 1900 Euro because the salaries of 1500 Euro, 2000 Euro, and 1600 Euro have been given to `bob`, `ely`, and `alice` respectively.

Let us investigate the tax payment process. The `abc` company is deactivated in order to disable the current behavior at least for `ely` while the `tax` compartment, an instance of the `TaxDepartment`, becomes active. In this configuration `ely` drops the role of the `Developer` and starts as a `Freelancer` (Line 44). The `Freelancer` role, currently bound to `ely`, also plays a `TaxPayer` role enabling `ely` to execute the tax payment by calling the `pay` method. Similarly, the `abc` company also plays the `TaxPayer` role for the same reason (Line 45), but

ely and abc company pay a tax with different rates according to the specification (10% for a Freelancer and 20% for a Company). Once ana, a Person playing a TaxCollector role, invokes a collectTax method, the taxes are collected from both the abc company and ely because they are the core objects of the TaxPayer role.

Since ely is now a freelancer and has earned 2000 Euro (Line 50), she has to pay 200 Euro for 10% tax. The abc company, however, pays 380 Euro (1900*0.2) for its 20% tax of the net income. The collected tax, in total 580 Euro, is debited to the tax department revenue.

Snippet 6.9: The main program of the Tax Management System.

```
1 public static void main(String... args){
2   Registry reg = Registry.getRegistry();
3   //Initialization of compartment, role and core object
4   Company abc = reg.newCompartment(Company.class, "ABC");
5   Person alice = reg.newCore(Person.class, "Alice");
6   Person bob = reg.newCore(Person.class, "Bob");
7   Person ely = reg.newCore(Person.class, "Ely");
8
9   //Construct initial binding
10  try(InitBindingBlock ib = abc.initBinding()){
11    alice.bind(Accountant.class);
12    bob.bind(Developer.class, 1500.00);
13    ely.bind(Developer.class, 2000.00);
14  }
15
16  abc.activate(); //a compartment is activated
17
18  bob.invoke("work"); //bob works to generate company's revenue
19  bob.invoke("work"); //bob works to generate company's revenue
20  ely.invoke("work"); //ely works to generate company's revenue
21  ely.invoke("work"); //ely works to generate company's revenue
22
23  System.out.println("====== Before Salary Payment ======");
24  System.out.println("Company revue is " + abc.getRevenue() + " Euro");
25  System.out.println("Bob saving is " + bob.getSaving() + " Euro");
26  System.out.println("Ely saving is " + ely.getSaving() + " Euro");
27  System.out.println("Alice saving is " + alice.getSaving() + " Euro");
28
29  alice.invoke("paySalary"); //alice generates salary
30
31  System.out.println("====== After Salary Payment ======");
32  System.out.println("Company revenue is " + abc.getRevenue() + " Euro");
33  System.out.println("Bob saving is " + bob.getSaving() + " Euro");
34  System.out.println("Ely saving is " + ely.getSaving() + " Euro");
35  System.out.println("Alice saving is " + alice.getSaving() + " Euro");
36
37  abc.deactivate(); //deactivate the abc compartment
38
39  TaxDepartment tax = reg.newCompartment(TaxDepartment.class,"Tax Dept.");
40  Person ana = reg.newCore(Person.class, "Ana");
41
42  try(InitBindingBlock ib = tax.initBinding()){ //role binding
43    ana.bind(TaxCollector.class);
44    ely.bind(Freelancer.class).bind(TaxPayer.class);
45    abc.bind(TaxPayer.class);
46  }
47
48  tax.activate(); //activate the tax compartment
49
```

```
50   ely.invoke("earn", 2000.0); //ely earns
51
52   ana.invoke("collectTax"); //Ana collects tax from ely and abc company
53
54   System.out.println("====== After Tax Payment ======");
55   System.out.println("Collected tax is " + tax.getRevenue() + " Euro");
56   System.out.println("Company revenue is " + abc.getRevenue() + " Euro");
57   System.out.println("Ely saving is " + ely.getSaving() + " Euro");
58 }
```

Snippet 6.10: Output produced by the main program.

```
1   Bob generates a revenue of 1500.0 Euro
2   Bob generates a revenue of 1500.0 Euro
3   Ely generates a revenue of 2000.0 Euro
4   Ely generates a revenue of 2000.0 Euro
5   ====== Before Salary Payment ======
6   Company revue is 7000.0 Euro
7   Bob saving is 0.0 Euro
8   Ely saving is 0.0 Euro
9   Alice saving is 0.0 Euro
10  ====== After Salary Payment ======
11  Company revenue is 1900.0 Euro
12  Bob saving is 1500.0 Euro
13  Ely saving is 2000.0 Euro
14  Alice saving is 1600.0 Euro
15  Ely pays a tax of 200.0 Euro
16  ABC pays a tax of 380.0 Euro
17  ====== After Tax Payment ======
18  Collected tax is 580.0 Euro
19  Company revenue is 1520.0 Euro
20  Ely saving is 3800.0 Euro
```

6.1.2.3 Validation

Concerning the requirements, the points to be validated are the following:

R1: Modularity. The main purpose of the case study is to demonstrate the dynamic adaptation at the instance level. There are several `Person` instances having totally different behaviors according to the played roles, i.e., `Developer`, `Accountant`, `TaxPayer`, etc. Those roles are clearly separated from the core `Person` as shown in Snippet 6.2, Snippet 6.3, and Snippet 6.5 respectively. Although a compartment contains roles from the model perspective, its concrete implementation is defined as a separate class. Snippet 6.7 shows how the `Company` compartment is implemented separately from its participating roles. In short, the definition of the core objects, roles and compartments is loosely coupled at design time while they merge together at run time with a binding operation. Therefore, this case study shows that building role-based applications on top of LyRT achieves modularity.

R2: Dynamic Activation. Those separated roles in the case study can be dynamically bound to the respective cores in order to perform certain tasks at run time as shown in Snippet 6.9. While `bob` instance invokes the `work` method as a `Developer` (Lines 18-21 of Snippet 6.9), `alice` instance executes the monthly payment via a `paySalary` method as an `Accountant` (Line 29 of Snippet 6.9). These new behaviors, encapsulated in roles,

are activated only when their associated compartment becomes active (i.e., `Company` compartment is activated in Line 16 of Snippet 6.9) because roles are configured to reside within a compartment during the binding process (i.e., `InitBindingBlock` as shown in Lines 10-14 of Snippet 6.9). When their compartment is no longer active, the behavior of the core objects reverts to the original. Another compartment with different binding configurations of the same cores can be activated to bring different behaviors to those cores. For instance, `ely` instance becomes a `Freelancer` in a `TaxDepartment` compartment soon after her previous compartment `Company` is deactivated and her new compartment `TaxDepartment` is activated (Lines 37-48 of Snippet 6.9).

6.1.3 Arcade Snake Game

In order to demonstrate the effectiveness of unanticipated adaptation, we pick up an arcade snake game in which the snake object and the game interface can dynamically be adapted to unforeseen functionality during run time. The case study is motivated by the following scenario:

> A developer is tasked to implement a classic snake game with a GUI laid out on a fixed square board in which the snake can move around. The goal is that the snake has to eat in order to extend its body length and level up. Once the food is eaten, another item is placed randomly in the square board. The challenge is that the snake is not allowed to neither hit its body nor the walls surrounding; otherwise the game is over. The game should be designed to address new requirements at run time without shutting it down.

In this case study we focus on unanticipated adaptation. Additionally, we explore the applicability of adapting the GUI applications.

6.1.3.1 Expected Functionality

Before discussing the underlying implementation of the unanticipated adaptation support, we first describe the technical functionality of the snake game and the expected scenarios for which unanticipated adaptation is required.

The class diagram of the snake game is illustrated in Figure 6.2. The snake game works on the `Board` class which consists of a two-dimensional array of cells. Each `Cell` contains rows and columns to identify its position on the `Board`. The `Cell` has different types such as empty cell, snake body, obstacle and food. The `Snake` class refers to the parts of the snake body which contains the reference to cells in the `Board`. The snake moves to the next cell based on a direction, maneuvered by players, specified in a `Router` class. The `Router` defines the position and the movement parameters of the snake and checks whether the snake collides with obstacles, i.e., its own body, walls, and other barriers. The food is a special cell (`Cell.CELL_TYPE_FOOD`) eaten by the snake when its head moves to the food cell position.

The `SnakeGame` class contains the main game settings as well as the main class to be executed. Additionally, there are classes dealing with user interfaces. `BoardPanel` is the main core interface where the game is painted and animated. `StatusPanel`, located at the bottom of the GUI (Figure 6.3), shows only the information about the command keys to perform

certain tasks whereas the `StatisticsPanel`, positioned at the top of the GUI, displays the movement speed of the snake and the number of items eaten.

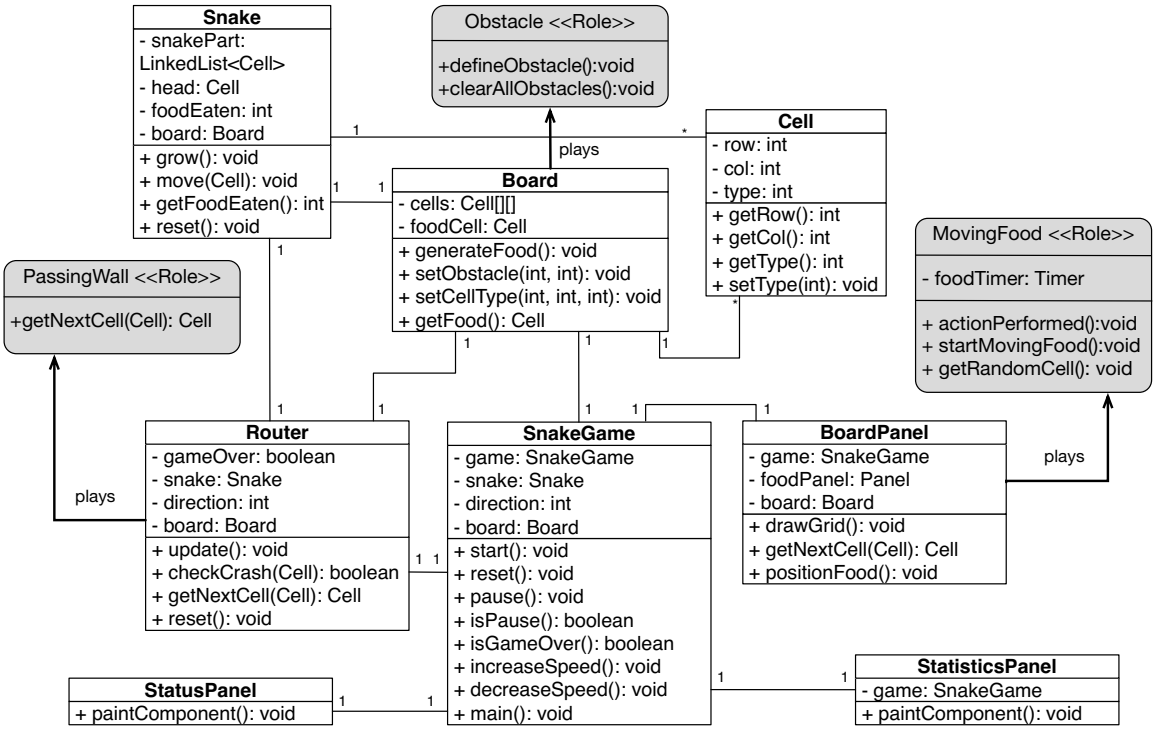


Figure 6.2: Class diagram of the Snake Game.

Roles, such as `PassingWall`, `Obstacle`, and `MovingFood`, are not given during compile time. They will be discussed in each unanticipated adaptation scenario. For common understanding, we attach these roles to their respective core objects as shown in Figure 6.2.

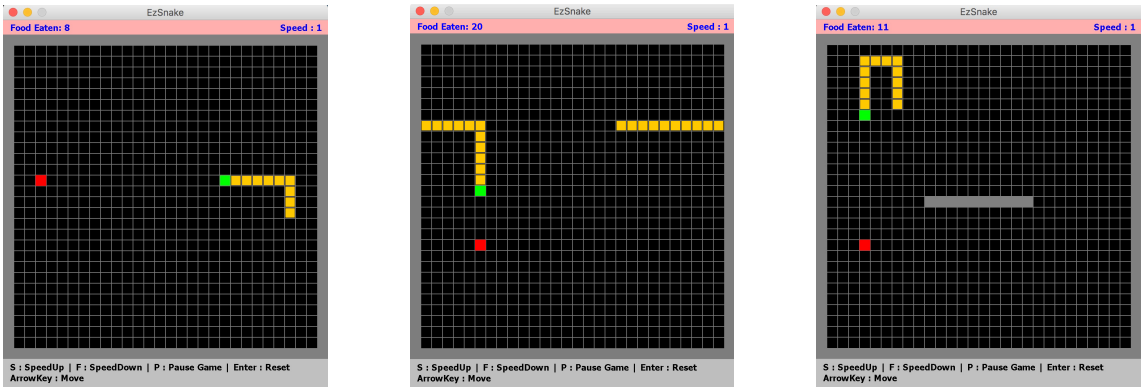
In order to satisfy players, a game is typically evolved over time to add new functionalities. While some functionalities are added statically, some are incorporated dynamically and unexpectedly. In the Snake Game, we present three scenarios of adding new functionalities dynamically in terms of unanticipated adaptation. Each of them has a purpose to demonstrate various settings of applying new behavior in LyRT. The three scenarios are:

1. The snake is allowed to pass through walls. For example, when it hits the wall on the right side, it will appear on the opposite side and vice versa.

2. An obstacle is added in the square board to impose an additional challenge in a higher level of the game. The added obstacle can also be removed.

3. The food, which was initially in a static position, is moving randomly at a slower speed than that of the snake to be sure that the snake can catch the moving food. While playing, the moving food can be reverted to the fixed position in response to the game settings.

Figure 6.3a presents the default game settings without applying any unanticipated adaptation. While Figure 6.3b shows the behavior of the snake when adapting to allow passing through a wall, Figure 6.3c exhibits the addition of obstacles on-the-fly.



(a) Original functionality. **(b)** Passing through walls. **(c)** Adding/removing obstacles.

Figure 6.3: User interfaces of the snake game.

6.1.3.2 Implementation

This section highlights the implementation of each unanticipated adaptation scenario. Only the implementation of relevant classes and new roles responsible for the scenario are discussed.

Let the game run, and we supposedly start applying unanticipated adaptation in three different scenarios. In each scenario, a core object is associated with exactly one role. The compartment has no significant impact on these scenarios. Since the existence of roles requires a compartment, we use only one global active compartment for role binding.

Scenario 1: Passing Through Walls. The aim of this scenario is to show the ordinary dynamic method dispatch which performs the lifting operation, i.e., the method is delegated to the role bound to a core. To allow the snake to pass through a wall, we need to update the logic when the snake reaches the edge of the board. The `Router`'s `getNextCell` method returns the next position based on the current direction. The `getNextCell` method needs to be modified but rather than changing the code directly in the `Router` class, a `PassingWall` role is defined. Snippet 6.11 shows an implementation of the `PassingWall` role which provides a modified logic of the `getNextCell` method. The implementation of this method looks similar to that of the `Router`. The main differences are the passing wall logic added in Lines 12, 15, 18, and 21. Note that Line 3 queries the core object (i.e., `Router` object) which plays the `PassingWall` role. Figure 6.4 elaborates this function as a sequential diagram.

Snippet 6.11: Implementation of `PassingWall` role.

```
1 public class PassingWall implements IRole{
2  public Cell getNextCell(Cell currentPosition) {
3   Router router = (Router)this.getCore();
4   Board board = router.getBoard();
5
6   int row = currentPosition.getRow();
7   int col = currentPosition.getCol();
8   int dir = router.getDirection();
9
10  if (dir == Router.DIRECTION_RIGHT) {
11    col++;
```

```
12      if(col==board.getColCount()-1) col=1;
13    } else if (dir == Router.DIRECTION_LEFT) {
14       col--;
15       if(col == 0) col = board.getColCount() - 2;
16    } else if (dir == Router.DIRECTION_UP) {
17       row--;
18       if(row == 0) row = board.getRowCount() - 2;
19    } else if (dir == Router.DIRECTION_DOWN) {
20       row++;
21       if(row == board.getRowCount()-1) row = 1;
22    }
23
24    return router.getBoard().getCells()[row][col];
25  }
26 }
```

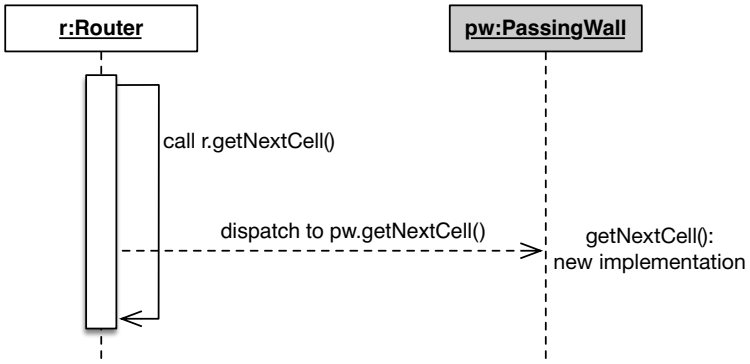


Figure 6.4: Sequential diagram of the passing-wall functionality.

In order to perform unanticipated adaptation for this configuration, an XML configuration file is needed to specify the identity of the current active compartment and the `Router` object. This information can be obtained by the run-time API. Snippet 6.12 shows this configuration file. Whenever the binding operation is completed, the call of the `getNextCell` method in `Router` will dynamically be dispatched to the one defined in the `PassingWall` role. The snake then can pass through the wall without crashing.

Snippet 6.12: An adaptation XML file for passing through walls.

```
1 <?xml version="1.0"?>
2 <adaptation>
3   <compartment type="net.lyrt.Compartment" id="123">
4     <rebind coreId="234" roleType="net.lyrt.demo.snake.PassingWall" />
5   </compartment>
6 </adaptation>
```

There are several ways to trigger the unanticipated adaptation. For example, it can be done either by having a GUI to submit the XML configuration file or through the network. In this implementation, we develop a `FileWatcher` daemon to monitor the change of the XML configuration file. The daemon fires the unanticipated adaptation process when the file change is detected.

Scenario 2: Enabling Moving Food. This scenario shows how it is possible to integrate a new software requirement and to revert back to the original settings by means of adding and removing roles in an unanticipated manner. In the previous scenario, the `PassingWall`

role provided a new method implementation to modify the existing functionality of the core object `Router` by using the dynamic method dispatch to allow the snake to pass through walls. In this scenario, the food is moving randomly within the near-by empty cells when the `MovingFood` role is bound to its core `BoardPanel`. There is no method dispatch presented in this scenario, but the implementation of the new requirement embedded in the role will call and manipulate the available methods in its core and will replace the old behavior.

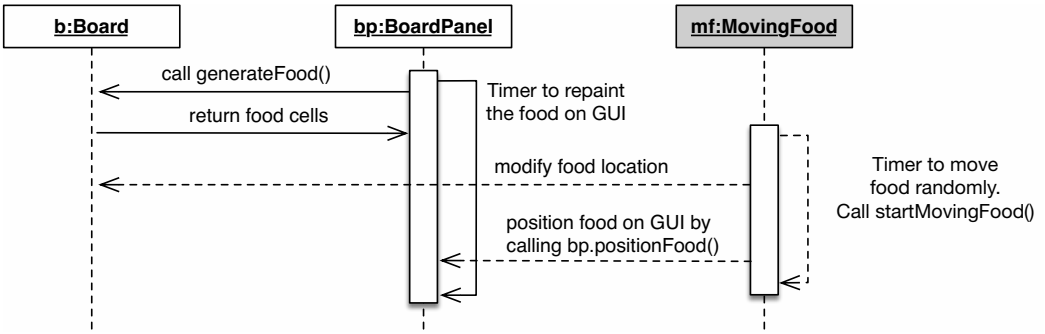


Figure 6.5: Sequential diagram of enabling moving food scenario.

Besides implementing an `IRole` interface, a `MovingFood` role, coded in Snippet 6.13, also implements a `java.awt.event.ActionListener` interface to support the interval execution with the timer `javax.swing.Timer`. When this role is bound, it is initialized and then the timer will be started with a delay of 500 milliseconds, as shown in the constructor. Eventually, the `actionPerformed` method is invoked repeatedly which causes the food to be moved. Line 16 gets a reference to the core `BoardPanel` which is used to manipulate the random cell generation of the food. Lines 19-29 ensure that the food is randomly generated only in empty cells. After that, the `positionFood` method in Line 30 is called to update the food location in the GUI. This process is executed every 500 milliseconds. To ease comprehension, the process is visualized in a sequential diagram as shown in Figure 6.5.

In order to bind the `MovingFood` role to the already running core `BoardPanel`, an XML configuration file, specifying the binding operation, is required. It is shown in Snippet 6.14. If this `MovingFood` feature is no longer needed, we can perform the unbinding operation via the same XML configuration as presented in Snippet 6.15.

Snippet 6.13: Implementation of `MovingFood` role.

```
1 public class MovingFood implements IRole, ActionListener{
2     final static int DELAY = 500; //food moving speed
3     Timer foodTimer;
4
5     public MovingFood(){
6         foodTimer = new Timer(DELAY, this);
7         foodTimer.start(); //Timer ticks for every 500ms
8     }
9
10    @Override
11    public void actionPerformed(ActionEvent e) {
12        startMovingFood(); //repetitive calls every 500ms
13    }
14
15    public void startMovingFood(){
16        BoardPanel boardPanel = (BoardPanel) getCore();
17        if(boardPanel == null) return; //when unbound occurs
18        Board board = boardPanel.getBoard();
```

```
19          while(true){
20              Cell foodCell = board.getFoodCell(); //get next cell
21              Cell nextCell = getRandomCell(board, foodCell);
22              if(nextCell.getType() == Cell.CELL_TYPE_EMPTY){
23                  board.setCellType(foodCell.getRow(), foodCell.getCol(), Cell.
                        CELL_TYPE_EMPTY); //Reset old food cell
24
25                  //Set new food cell
26                  board.setFoodCell(nextCell.getRow(), nextCell.getCol());
27                  break;
28              }
29          }
30          boardPanel.positionFood(); //Position the food on board panel
31      }
32
33      private Cell getRandomCell(Board board, Cell currentPosition) {
34          int direction = (int)(Math.random()*4); //next direction
35          int row = currentPosition.getRow();
36          int col = currentPosition.getCol();
37
38          if (direction == Router.DIRECTION_RIGHT) col++;
39          else if (direction == Router.DIRECTION_LEFT) col--;
40          else if (direction == Router.DIRECTION_UP) row--;
41          else if (direction == Router.DIRECTION_DOWN) row++;
42          Cell[][] cells = board.getCells();
43          return cells[row][col];
44      }
45  }
```

Snippet 6.14: An adaptation XML file for binding `MovingFood` role.

```
1 <?xml version="1.0"?>
2 <adaptation>
3   <compartment type="net.lyrt.Compartment" id="123">
4     <bind coreId="234" roleType="net.lyrt.demo.snake.MovingFood" />
5   </compartment>
6 </adaptation>
```

Snippet 6.15: An adaptation XML file for unbinding `MovingFood` role.

```
1 <?xml version="1.0"?>
2 <adaptation>
3   <compartment type="net.lyrt.Compartment" id="123">
4     <unbind coreId="234" roleType="net.lyrt.demo.snake.MovingFood" />
5   </compartment>
6 </adaptation>
```

Scenario 3: Adding/Removing Obstacles. The purpose of this scenario is to add a new feature (adding/removing obstacle) in the game by explicit method calls, configured in the XML configuration file. Additionally, we demonstrate the ability of the `ClassReloader` allowing several new implementations of the same role type to be reloaded during run time. An obstacle is just a collection of cells whose type is set to `Cell.CELL_TYPE_OBSTACLE` in the `Board` instance. Thus, an `Obstacle` role, lately introduced to be bound to the `Board` instance, can be used to manipulate cells of the `Board`.

Snippet 6.16 shows the implementation of the `Obstacle` role with a `defineObstacle` method which contains a setting of 10 horizontal blocks of an obstacle (Lines 5-7). A visual GUI is

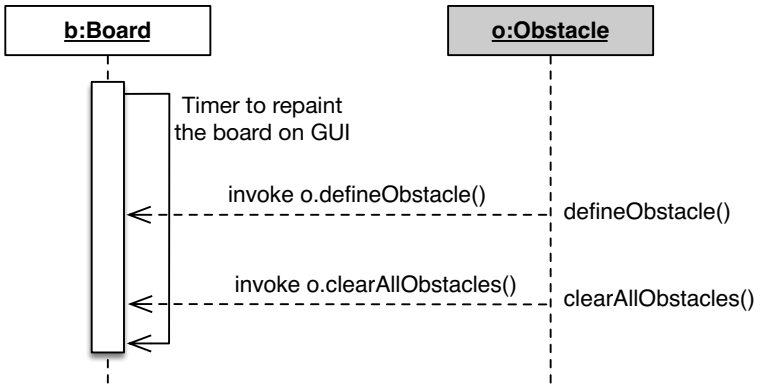


Figure 6.6: Sequential diagram of adding and removing obstacles

depicted in Figure 6.3c. The `defineObstacle` method is explicitly invoked after being bound in which it can be configured in the `invoke` element, specified in the XML configuration file (Line 5 of Snippet 6.17). Figure 6.6 adds an elaboration support in the form of a sequential diagram.

Snippet 6.16: Implementation of adding obstacles in the `Obstacle` role.

```
1 public class Obstacle implements IRole {
2   public void defineObstacle(){
3     Board board = (Board)this.getCore(); //get a refer to a core object
4
5     for(int y=0; y<10; y++){
6       board.setObstacle(13, y+10);
7     }
8   }
9 }
```

Snippet 6.17: An adaptation XML file for adding obstacles in the `Obstacle` role.

```
1 <?xml version="1.0"?>
2 <adaptation>
3   <compartment type="net.lyrt.Compartment" id="123">
4     <bind coreId="420521101" roleType="net.lyrt.demo.snake.Obstacle" >
5       <invoke method="defineObstacle" returnType="void" />
6     </bind>
7   </compartment>
8 </adaptation>
```

Supposedly, during the game play, there is a need to clear all the obstacles in the GUI. To do so, a `clearAllObstacles` method is added to the `Obstacle` role which has already been loaded to the target JVM in the previously unanticipated adaptation. Snippet 6.18 shows a modified implementation of the `Obstacle` role. This role must be recompiled, and the binding of the core object to this role is reconfigured with an explicit `clearAllObstacles` method invocation as illustrated in Snippet 6.19. The `ClassReloader` reads the new implementation of the `Obstacle` role and the role is rebound to the core `Board`. As a result, all the existing obstacles disappear from the GUI.

Snippet 6.18: Implementation of removing obstacles in the `Obstacle` role.

```
1 public class Obstacle implements IRole {
2   public void defineObstacle(){
3     ...
4   }
5
6   public void clearAllObstacles(){
7     Board board = (Board)getCore(); //get a refer to a core object
8     Cell food = board.getFoodCell();
9     for(int row=1;row<board.getRowCount()-1;row++){
10      for(int col=1;col<board.getRowCount()-1;col++){
11        board.setCellType(row, col, Cell.CELL_TYPE_EMPTY);
12      }
13    }
14    board.setFoodCell(food.getRow(), food.getCol()); //set food back
15  }
16 }
```

Snippet 6.19: An adaptation XML file for eliminating obstacles in the `Obstacle` role.

```
1 <?xml version="1.0"?>
2 <adaptation>
3   <compartment type="net.lyrt.Compartment" id="123">
4     <rebind coreId="420521101" roleType="net.lyrt.demo.snake.Obstacle" >
5       <invoke method="clearAllObstacles" returnType="void" />
6     </bind>
7   </compartment>
8 </adaptation>
```

6.1.3.3 Validation

With respect to the requirements, the points to be validated are as follows:

R1: Modularity. The variants, such as `PassingWall`, `MovingFood`, and `Obstacle`, used to adapt the unforeseen functionality, are modeled as roles defined separately from their core objects. Those roles are not given upfront but introduced dynamically at run time as shown in Snippet 6.11, Snippet 6.13, and Snippet 6.16. Therefore, this requirement is satisfied.

R2: Dynamic Activation. Assuming that roles are already loaded into the runtime, on how these roles are dynamically loaded will be discussed in the validation of the next requirement. Since the snake game is running, the role binding operation must be prescribed in an XML configuration file, e.g., Snippet 6.12. The dynamic activation then happens as an event that is fired when there is a change in the XML file. Once activated, the core objects swift to new behaviors embedded in roles.

R3: Late Variants Adoption. The newly introduced roles, `Obstacle`, `MovingFood`, and `PassingWall`, are dynamically introduced and attached to the existing core objects `Board`, `BoardPanel`, and `Router` respectively. The ability of the `ClassReloader` allows these lately introduced roles to be loaded and reloaded to the running JVM, so that they can be bound to their core objects by the parsing process of the XML configuration file. In order to fulfill unanticipated adaptation, a reference to a core object is necessary, and this information can be obtained from the run-time API. The `FileWatcher` daemon monitors the change of the XML file and fires the parsing process. Thus, adaptation

is performed in an unanticipated manner. The three unanticipated adaptations show that LyRT fulfills this requirement.

6.1.4 File Transfer Application

This case study shows that all requirements are needed but emphasizes the consistent behavior, rollback recovery, and continuous deployment features. The case study is motivated by the following scenario:

> A developer is tasked to implement an adaptive file transfer application which supports the concurrent access of multiple clients. The server's function of transmitting data has to be adapted several times. Originally, the server uses a raw format to transmit data while the adaptation requires a compression, or an encryption, or a combination of both to perform a similar task. Assuming that the server uses a shared channel object to handle data encoding/formatting before transmission. Hence, performing adaptation for a new client also affects the existing clients, which is valid according to specification. However, if the existing clients engage in a file transfer, the adaptation should not affect them immediately. The adaptation for these clients is then performed in the next file download.
>
> High availability of the server runtime is another major requirement. The runtime should be able to incorporate better compression or encryption algorithms on-the-fly. Additionally, the runtime should be able to recover from run-time failures which were caused by adaptation in order to improve the application's overall availability.

Aside from the consistent adaptation and the recovery feature, this case study also demonstrates the adaptation that can happen in multiple threads. Objects living in a particular thread may behave differently in other threads, although those objects are derived from the same type. The consistency of the system stays intact as these objects do not share anything among them. However, if a state of an object is modified by multiple threads at the same time, it poses a common racing issue. We do not cover this problem as it opens another interesting research area on the domain of Software Transactional Memory (STM) [MS04] for role-based systems.

6.1.4.1 Expected Functionality

Before explaining the underlying implementation, the expected application functionality is described. First, this case study consists of both the server and the client side. Hence, a simple protocol is needed.

A Simple File Transfer Protocol is based on the Transmission Control Protocol (TCP) established between server and client although we focus mainly on the server side. The protocol is described as follows:

> A server is listening on a specific port accepting two commands, `GET N` and `QUIT`. After a TCP client socket is established, a thread for recent client connection is started allowing the server to accept other clients concurrently. The `GET N` is

sent from a client to request a file. Instead of downloading a real file, `N` denotes a number of chunks for a particular file to be downloaded. After receiving the `GET N` command, the server sends file chunks continuously for `N` times as illustrated in Figure 6.7. For the sake of simplicity, a string of `DATA` is represented as a file chunk. The `QUIT` command is used to disconnect from the server.

In order to integrate a smooth recovery, the server keeps track of the number of requested chunks (`N`) and the chunks which have been sent completely. If there is a failure happening during the sending process, the sending of the remaining chunks can be resumed.

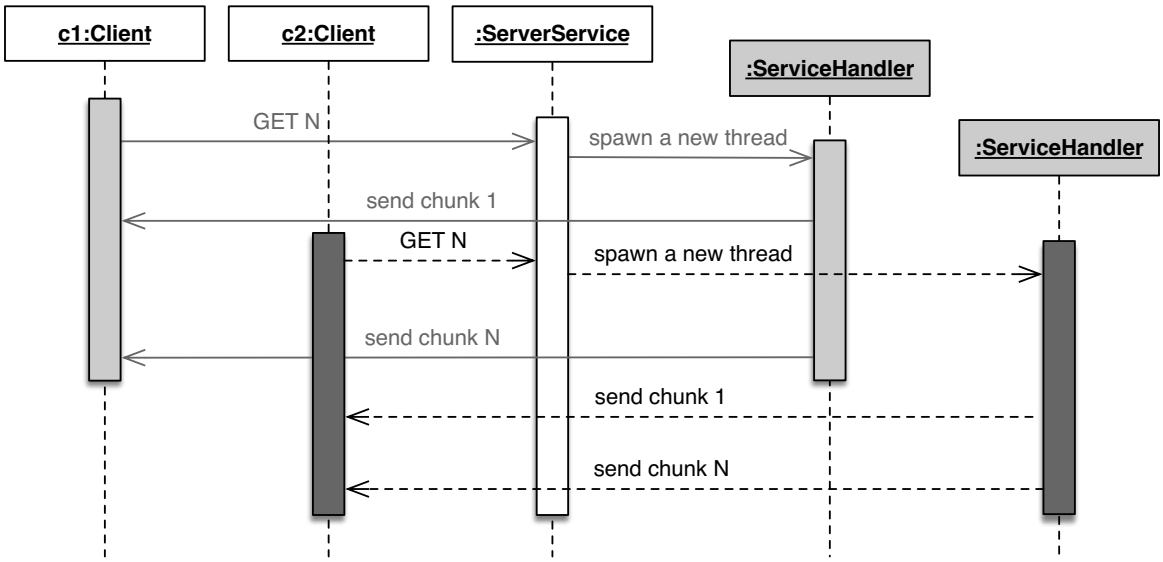


Figure 6.7: A sequential diagram of the file transfer protocol handling two clients.

This case study focuses on the dynamic behavior of the shared `Channel` object which can be used to format data in a raw (original behavior), a compression (`LZ`), an encryption (`AES`) or a combination of the `LZ` and the `AES`. According to the role model, the `Channel` object is a core whereas `LZ` and `AES` are roles. A dynamic activation is needed to be operational.

Dynamic Activation. The server is designed to support multiple clients concurrently by a separate client handling service (`ServiceHandler`). In each client service, there is a single active compartment to handle a scope of dynamic behavioral activation at the instance and the thread level. For clients, they therefore may receive different formats to encode file chunks although the `Channel` object is shared among all clients. This is a purpose of having the shared `Channel` object to demonstrate dynamic activation.

To visually demonstrate the dynamic activation, the server is developed with a GUI that manages all the connected clients. The administrator can select a single or multiple clients to adapt possible combinations as shown in Lines 56-64 of Snippet 6.20. For example, Figure 6.8 shows the server interface connecting to three clients ① but only two of them are selected for adaptation which can be done via a button click ②. A client is identified by its IP address and local port number, i.e., `127.0.0.1:56257`. These clients request a file with 10 chunks by issuing a command `GET 10`. During the transmission of the chunks, the administrator selects two clients, whose port numbers are `56257` and `56259`, to be adapted first by `LZ` and then `AES`. As a result, these two clients receive different formats of chunks while the client with `56258`

port number continues to receive chunks with a raw format as illustrated in Figure 6.9.

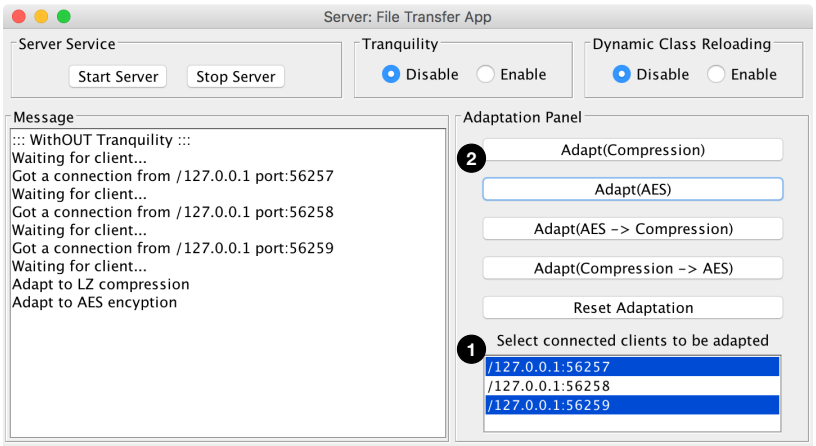


Figure 6.8: A server interface to demonstrate a dynamic activation.

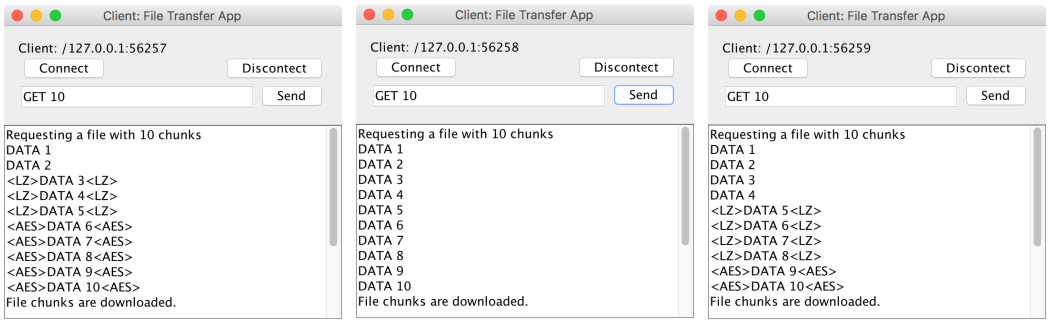


Figure 6.9: An interface of clients to demonstrate a dynamic activation.

Consistent Behavior. Note that the adapted clients face inconsistent behavior of receiving chunks. Evidently, the clients with the connected port number `56257` and `56259` are receiving file chunks with different formats (i.e., partly raw, compressed, or encrypted) as shown in Figure 6.9, so the server should be able to handle this issue although it was forced to adapt.

Recovery. With respect to the *deep-play*-relation, the `Channel` object can be bound to the `AES` role, and the `AES` role then binds the `LZ` role to perform a stacking behavior with the lowering function as explained in Section 4.3.5. This relation can also be that the `LZ` role binds to the `AES` role.

As already explained in Chapter 4 (Section 4.5), the *play*-relation of the `Channel` object works properly with the isolated binding to either `LZ` or `AES` role. However, the interaction of the three entities might cause a run-time crash. This is the case when the `Channel` object binds to the `LZ` role and the `LZ` role binds to the `AES` role in the form of role-playing-role. A `DivideByZero` error comes from the `factor` variable (Line 4 of Snippet 6.24) returning a zero value from the lowering function `invoke("factor")` which refers to the `factor` method of the `LZ` role (Snippet 6.23) due to that fact the `LZ` role is a player of the `AES` role. Not all interactions cause this error at least when `AES` role, a role of the `Channel` object, is a player of the `LZ` role because the `factor` method invocation in the `AES` refers to the one in the `Channel` object, which values 10.

We expect the runtime should be able to recover from this execution failure resulting from adaptation.

Continuous Deployment. It is an additional feature to be expected that allows the defective roles, i.e., `AES`, to be fixed and be integrated once again into the system. We call this process a *continuous deployment* as the runtime continues to operate with a new functionality regardless of failures.

6.1.4.2 Implementation

There are several classes to be implemented including the GUI. Below, only essential classes are presented while the full implementation is available in our GitHub repository.

- `ServerService`: It is a class responsible for creating a server socket and managing the `ServiceHandler` thread class to establish a particular connecting client.
- `ServiceHandler`: It is a thread class handling a client socket and the implementation of the file transfer protocol (Snippet 6.20).
- `Communicator`: It is a class dealing with actual data communication which contains a `send` and a `receive` method (Snippet 6.21).
- `AppState`: It contains a collection of static variables to be shared among other classes.
- `BugSensor`: It is an implementation of a bug sensor managing a rollback recovery process (Snippet 6.25).

Implementation of the File Transfer Protocol. The File Transfer Protocol discussed earlier is fully implemented in a `ServerHandler` class which extends a `Thread` class. Once a client is connected, the server returns a `Socket` and spawns the `ServiceHandler` class parameterizing the returning `Socket`. While the server waits for other clients, the spawned `ServiceHandler` object handles the communication protocol with the connected client. The implementation of `ServiceHandler` is shown in Snippet 6.20.

Apart from the protocol implementation, the code of the `ServiceHandler` class also contains essential code relevant to LyRT. For example, the installation of the bug sensor is in Line 17; the consistency mechanism is in Lines 33-36; and the adaptation process is in Lines 56-64. This code will be discussed later in this section.

Snippet 6.20: Implementation of the `ServiceHandler`.

```
1 public class ServiceHandler extends Thread{
2   Socket socket;
3   Compartment compartment;
4   Communicator communicator;
5   Channel channel;
6   RecoveryProperty recProp = new RecoveryProperty();
7
8   public ServiceHandler(Socket socket, RecoveryProperty pop){
9     this.socket = socket;
10    channel = AppState.channel;
11    communicator = new Communicator(this.socket, channel);
12    this.recProp = pop;
```

```
13      this.compartment = recProp.compartment;
14    }
15
16    public void run(){
17      Thread.currentThread().setUncaughtExceptionHandler(new BugSensor(this));
            //install the bug sensor
18
19      compartment.activate();
20      while(true){
21        if(recProp.isRecovered) { //Resuming sending chunks on error
22          resumeSendingFileChunks();
23          recProp.isRecovered=false;
24        }
25        String msg = communicator.receive();
26        if(msg.equals(Command.QUIT)){ //Disconnect
27          compartment.deactivate(false);
28          AppState.listModel.removeElement(this);
29          break;
30        }else if(msg.contains(Command.GET)){ //Request a file
31          String[] com = msg.split(" ");
32          recProp.numberOfChunks = Integer.parseInt(com[1]);
33          if(AppState.isTranquil) //Tranquility
34            try(ConsistencyBlock cb = new ConsistencyBlock()) {
35              sendFileChunks(recProp.numberOfChunks);
36            }
37          else sendFileChunks(recProp.numberOfChunks); //Without tranquility
38        }
39      }
40    }
41
42    private void sendFileChunks(int startIdx, int n){
43      for(int i=startIdx; i<n; i++){
44        delay(500);
45        communicator.send("DATA " + (i+1)); //simulate a file chunk
46        recProp.offset = i;
47      }
48    }
49
50    private void sendFileChunks(int n){ sendFileChunks(0, n); }
51
52    private void resumeSendingFileChunks(){
53      sendFileChunks(recProp.offset+1, recProp.numberOfChunks);
54    }
55
56    public void processAdaptation(String op){
57      compartment.activate();
58      try(AdaptationBlock ab = new AdaptationBlock()){
59        if(op.equals("AES")) channel.bind(AES.class);
60        if(op.equals("LZ")) channel.bind(LZ.class);
61        if(op.equals("LZ-AES")) channel.bind(LZ.class).bind(AES.class);
62        if(op.equals("AES-LZ")) channel.bind(AES.class).bind(LZ.class);
63      }
64    }
65  }
```

Communicator. The `Communicator` implements real network communication as shown in Snippet 6.21. The parameterized `Socket` holds a reference to a particular connected client, and the `Channel` object is responsible for the encoding and decoding of sent and received

messages respectively (Lines 19 and 26 of Snippet 6.21). That is where the `Channel` object is of importance to demonstrate the behavioral change in the data transmission.

Snippet 6.21: Implementation of the `Communicator`.

```
1 public class Communicator {
2     private Socket socket;
3     private BufferedReader input;
4     private PrintWriter output;
5     private Channel channel;
6
7     public Communicator(Socket socket, Channel channel){
8         this.socket = socket;
9         this.channel = channel;
10        try {
11            input = new BufferedReader(new InputStreamReader(this.socket.
                  getInputStream()));
12            output = new PrintWriter(this.socket.getOutputStream(), true);
13        } catch (IOException ioe) {
14            ioe.printStackTrace();
15        }
16    }
17
18    public void send(String data) {
19        String msg = channel.invoke("format", String.class, data);
20        output.println(msg);
21    }
22
23    public String receive() {
24        try {
25            String msg = input.readLine();
26            return channel.invoke("unformat", String.class, msg);
27        } catch (IOException e) {
28            e.printStackTrace();
29        }
30        return "";
31    }
32 }
```

Compartment. In this case study, compartments are used to activate dynamic behavior, but no other properties or methods are needed. Hence, it is sufficient to use the default `net.lyrt.Compartment` class given by LyRT.

Channel Core Object. The `Channel` object is the core containing three main methods, `format`, `unformat`, and `factor`, to be adapted. By default, the implementation of both `format` and `unformat` methods simply returns the passing `string` message. The additional `factor` method is used to demonstrate the error case which will be discussed later. Snippet 6.22 shows the implementation of the `Channel` object.

Snippet 6.22: Implementation of the Channel core object.

```
1 public class Channel implements IPlayer{
2   public String format(String data) { return data; }
3
4   public String unformat(String data) { return data; }
5
6   public int factor(){ return 10; }
7 }
```

LZ and AES Role. Since the Channel object is required to adapt in order to perform compression, encryption, or a combination of both, these functions are implemented in roles namely LZ and AES for compression and encryption respectively. While the LZ compression role implementation is shown in Snippet 6.23, the AES encryption role is coded in Snippet 6.24. These two roles share the same format method signature with the Channel core object but they are implemented differently to provide compression and encryption functionality. For the sake of simplicity, we discuss only the format method for data preparation to be sent from the server to clients. Furthermore, we return the data with enclosing <LZ> and <AES> to simulate the LZ compression and AES encryption algorithm respectively.

Snippet 6.23: Implementation of the LZ compression role.

```
1 public class LZ implements IRole{
2   public String format(String data){
3     String msg = invokePlayer("format", String.class, data);
4     return "<LZ>" + msg + "<LZ>"; //perform compression algorithm
5   }
6
7   public int factor(){ return 0; }
8 }
```

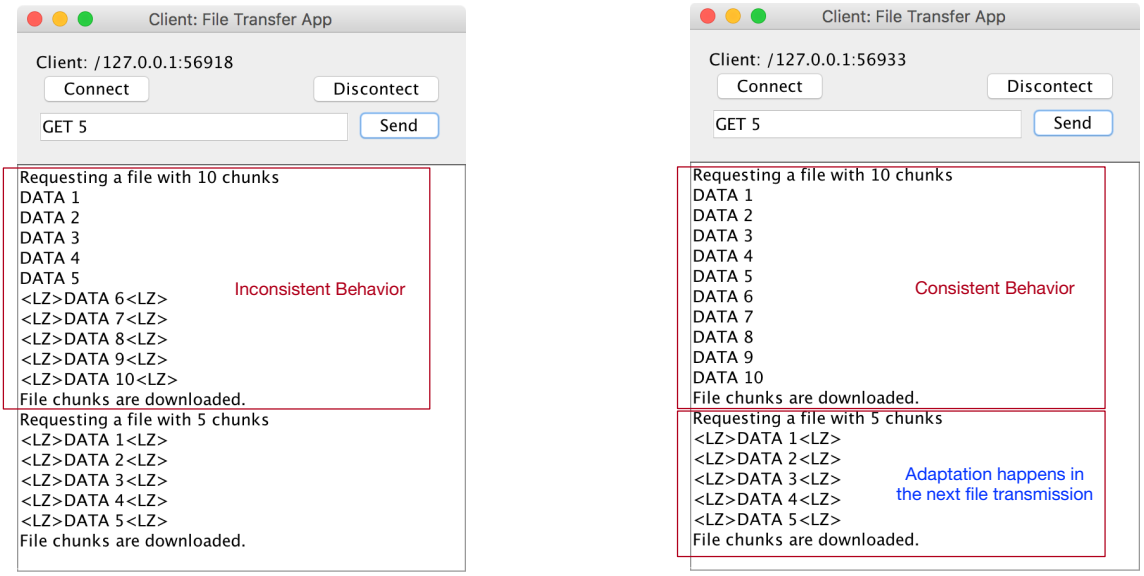
In order to demonstrate the rollback recovery feature, we inject the possible case of the DivideByZero error in the AES encryption role on Line 5 of Snippet 6.24. Unlike the Channel core and the LZ role, the AES role is not equipped with a factor method but will use the factor method from its player by invoking invokePlayer("factor", int.class) on Line 4 of Snippet 6.24.

Snippet 6.24: Implementation of the AES encryption role.

```
1 public class AES implements IRole{
2   public String format(String data){
3     String msg = invokePlayer("format", String.class, data);
4     int factor = invokePlayer("factor", int.class);
5     int errorInjection = 1/factor; //simulating an error in certain
          composition
6     return "<AES>" + msg + "<AES>"; //perform encryption algorithm
7   }
8 }
```

Dynamic Activation. We do not focus on demonstrating the dynamic activation since the two case studies earlier cover that thoroughly. In fact, it is straightforward to use the role binding operation as shown in the processAdaptation method in Lines 56-64 of Snippet 6.20. The main difference is that rather than relying on the InitBindingBlock, the AdaptationBlock is used to hook into the checkpoint process for rollback.

Tranquility. In order to enable consistent behavior, the `ConsistencyBlock` is needed to surround the `sendFileChunks` method in the `ServiceHandler` class (Snippet 6.20), which contains the adaptive behavior of the `Channel` object. The behavior of the `Channel` object will not be changed even though the server runtime is forcefully adapted during the file chunks' transmission. The time the next file is downloaded, the file chunks will be transmitted in the format which was defined in the most recent adaptation.



(a) Without a tranquility setting. **(b)** With a tranquility setting.

Figure 6.10: Impact of tranquility settings on object behavior.

In our example, two clients connect to the server at different times because we want to experiment with and without tranquility settings on the server side. However, both of them download two files with 10 and 5 chunks by sending the commands `GET 10` and `GET 5` one after another. During the first file downloading process, the server adapts to use `LZ` compression. Without tranquility support, there is an inconsistent behavior among file chunks as shown in Figure 6.10a, as opposed to consistent behavior with tranquility support, depicted in Figure 6.10b. In the next file transmission, the `LZ` compression will be used for transmission only for that client which connects with tranquility setting turned on.

Rollback Recovery. In order to recover from run-time execution failures caused by adaptation, a proper checkpoint of the current configuration, a rollback recovery, and a bug sensor are needed (Section 4.5). While the checkpoint process is embedded in the adaptation process, the rollback recovery attaches to the bug sensor whose implementation is shown in Figure 6.25. The bug sensor is installed in each client session in Line 17 of Snippet 6.20, and each session involves an active compartment. Therefore, if an error occurs, it affects only the currently engaging client session.

When the `DivideByZero` error is encountered, the `uncaughtException` method in Snippet 6.25 is invoked. The runtime rolls back from the current defective configuration to the previous one stored in the checkpoint. After that the `ServiceHandler` is restarted while the client socket is still maintained (Line 19). Hence, the client session remains connected. However, the logic of sending file chunks is disrupted. That is why we leave the recovery process for

a particular application partly to the protocol developer. In this case study, our proposed file transfer protocol is integrated with a simple recovery protocol complemented with our rollback recovery mechanism.

Snippet 6.25: Implementation of the `BugSensor`.

```
1 public class BugSensor implements Thread.UncaughtExceptionHandler {
2   private ServiceHandler client;
3
4   public BugSensor(ServiceHandler client){ this.client = client; }
5
6   @Override
7   public void uncaughtException(Thread t, Throwable e) {
8     Registry reg = Registry.getRegistry(); //get registry instance
9     Compartment comp = reg.getActiveCompartments().get(t.getId());
10
11    //Rollback
12    ControlUnit.rollback(comp);
13
14    //Logging to the server message panel
15    AppState.appendMessage (t + " throws " + e);
16    AppState.appendMessage(t.getName() + ">>>>> Rollback >>>> ");
17
18    //Restart server service handler
19    AppState.serverService.restartService(client);
20  }
21 }
```

The recovery protocol keeps track of the number of chunks that have been sent. During the recovery process, the `ServiceHandler` is restarted and the recovery logic resumes sending the left chunks (Lines 22-23 of Snippet 6.20). Again, there is no disruption at the client side due to the execution failure and recovery process happening only at the server side. Figure 6.11 shows a simple interface of the rollback recovery process of both server and client with logging messages. The server adapts several times while serving a client request by binding the `Channel` object to `LZ`, `AES` and the interaction of `AES` and `LZ` role ①. So far, it works fine, as depicted on the client side ②, but in the last attempt (the interaction of `LZ` and `AES`) it causes a `DivideByZero` exception detected by the bug sensor ③. The execution failure happens before the file chunks have completely been sent ④, but there appears no service disruption at the client side ⑤. This proves that our rollback recovery mechanism works as expected.

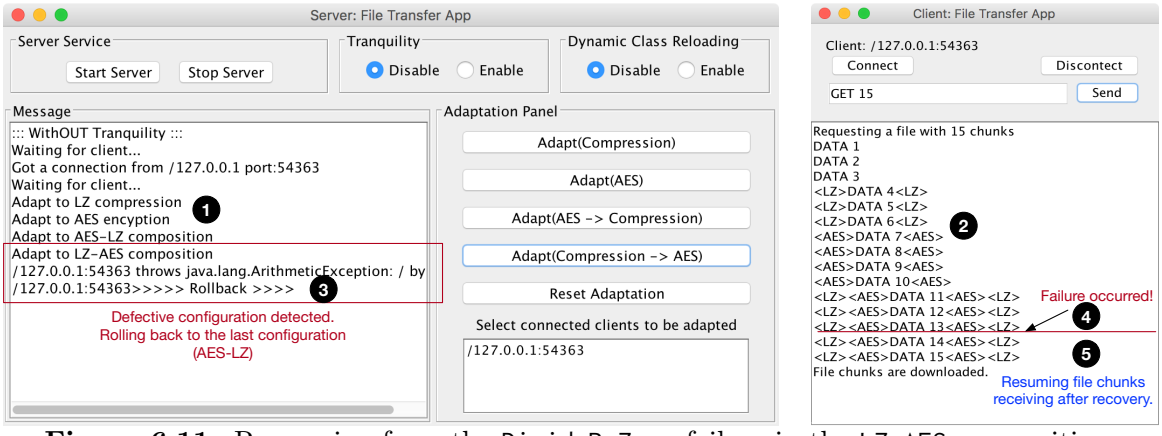


Figure 6.11: Recovering from the `DivideByZero` failure in the `LZ-AES` composition.

Note that in the client message window, file chunks appear in inconsistent state due to the adaptation. In this experiment, the server disables the tranquility support in order to demonstrate the effect of the rollback recovery process. If tranquility was enabled, there would not occur any failure because of the adaptation delay. However, it will cause a failure in the next file request because the failure is detected when a particular composing method is executed.

Continuous Deployment. In addition to exclude of the defective role by rolling back, the defective role is reported allowing the developer to fix the bug hoping that the fixed role can be reapplied once again. With the unanticipated adaptation support, this fixed role can be installed after modification and recompilation.

Figure 6.12 shows the process of the continuous deployment in the File Transfer Application on both server and client. First, the server adapts the connected client twice from `LZ` to `AES` role ① and ②. Apparently, no failure is detected. The server then attempts to adapt the client with `LZ` and `AES` composition which causes an execution failure ③. Therefore, the client still receives the `AES` encoding message ④ due to rollback process at the server side. The defective `AES` role must be altered in the sense that the `format` method and the `factor` variable cannot produce the same `DivideByZero` error. For example, the code in Line 5 of Snippet 6.24 can be modified to be `1*factor` to simulate the bug elimination. Once compiled, the server runtime must enable the dynamic class reloading capability ⑤ before the `LZ` and `AES` composition can be reapplied ⑥. Eventually, the client receives the properly encoded file chunks resulting from the fixed `AES` role ⑦.

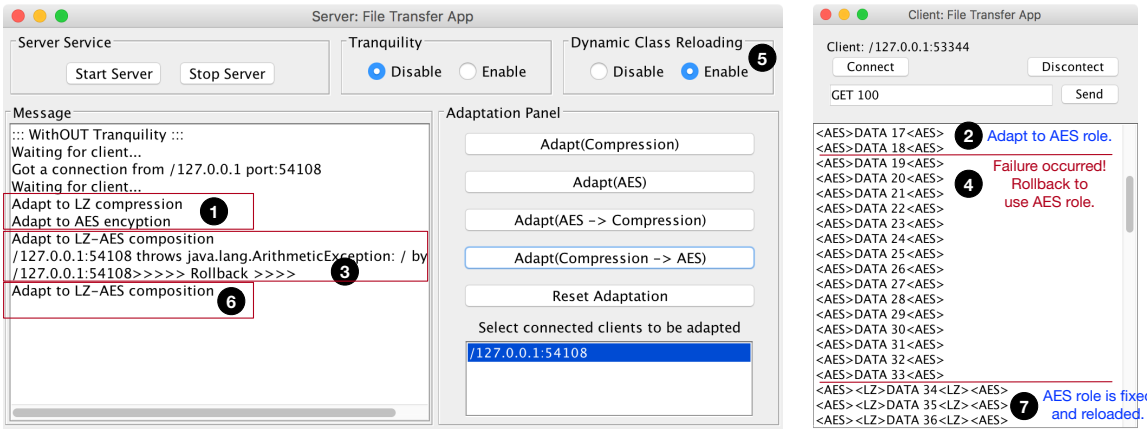


Figure 6.12: Continuous deployment by fixing the `AES` role to be rebound.

6.1.4.3 Validation

With respect to the requirements, the points to be validated are the following:

R1: Modularity. Clearly, there is a separation of the core object and its adapting variants. In this case study, while the `Channel` object is a core, the `LZ` and the `AES` are roles. These entities are defined separately as shown in Snippet 6.22, Snippet 6.23, and Snippet 6.24, but they are bound together at run time. Additionally, the default compartment that is used to scope the dynamic behavior is also defined independently. Therefore, they are modular making this requirement satisfied.

R2: Dynamic Activation. As explained, the `Channel` object is shared among all connected clients whose behavior is adapted based on the bound roles either `LZ` or `AES` or the combination of both. The activation is performed under the binding or unbinding operations which can be seen in Lines 56-74 of Snippet 6.20. Due to the activation support at the instance and the thread level, the `Channel` object has different behaviors executing concurrently for respective clients as shown in Figure 6.9.

R3: Late Variants Adoption. Although we did not present it explicitly, this feature was explained in the continuous deployment scenario in which the reporting bug in the `AES` role is eliminated, and the `AES` role is recompiled to be installed once again. Due to the dynamic class reloading, the new implementation can be dynamically reloaded from the bug-free `AES` role. Hence, this requirement is supported.

R4: Object-Level Tranquility. Having consistent behavior during the transmission of chunks is a goal in this case study. The server's tranquility setting can be enabled and disabled using an interface. Using the tranquility option, the `sendFileChunks` method is guarded by the `ConsistencyBlock` as expressed in Snippet 6.20 in Lines 34-36. As a result, the client receives all the chunks in a uniform encoding in spite of adaptation. The adaptation, occurred in between, will become effective in the next file request (Figure 6.10b).

R5: Failure Handling. While binding the `Channel` object to either the `LZ` or the `AES` role independently is not problematic in terms of execution failures, the interaction of these two roles may cause failures. Snippet 6.25 demonstrates the bug sensor implementation to embrace the `DivideByZero` error raised as an exception and to roll back to the previous working configuration. Complementing with the recovery protocol, the server runtime successfully recovers from failure and resumes sending the remaining file chunks without disrupting the overall file transfer (Figure 6.11).

R6: Continuous Deployment. This is a derived requirement as LyRT supports R3 and R5. We presented a dedicated continuous deployment scenario to deal with a defective role composition (i.e., the `LZ` binding to the `AES` role). The modified or bug-free `AES` role can be cleanly composed to the system once again. All these things are done while the system is running (Figure 6.12). Therefore, it contributes to the high availability of the runtime.

Object-Level Tranquility versus the Original Tranquility

As mentioned in Chapter 2, our behavioral consistency mechanism (Requirement R4) is founded on the concept of *tranquility* [VEBD07]. Still the term *tranquility* is used throughout this dissertation for behavioral consistency. The original tranquility used the term *transaction* to describe the interactions between participating nodes. The nodes are allowed to update when they reach a *tranquil* state (see Section 2.3). To avoid confusion, we coined the term *consistency block* instead of *transaction* (see Section 4.4). The consistency block is used to prevent the engaging objects from changing their behavior in a series of ongoing method executions. The notion of the transaction in the original tranquility concept is applied for the component-based systems where participating nodes are represented as singletons, and their communications are statically defined through ports and connections. Due to these criteria, the original tranquility concept is hardly applied to object level [ESMJ10]. Therefore, a dynamic update of the shared node (i.e., resembling the shared `Channel` object) in an execution environment where the node, participating and executing in multiple threads,

is infeasible.

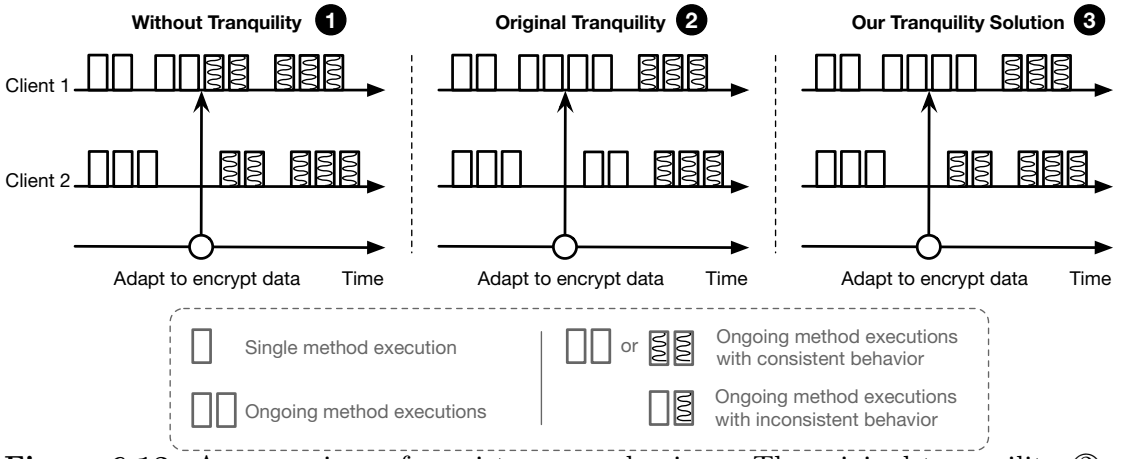


Figure 6.13: A comparison of consistency mechanisms. The original tranquility ② was proposed by Vandewoude et al. [VEBD07].

Figure 6.13 shows a comparison of consistency mechanisms. A system without tranquility mechanism ① is prone to behavioral inconsistencies if updates occur in the middle of ongoing method executions. This case study illustrated the behavior of a server running without the tranquility mechanism enabled (Figure 6.10). The original tranquility concept ② addressed this problem partially using transactions. It causes, however, a long disruption for `Client2`. Although the update of nodes is possible inside a transaction for certain cases, it is likely impossible in the worst case in which the node is shared. Additionally, there is no concurrent support as mentioned earlier. Therefore, a consistent behavior is achieved but with longer disruption for the original tranquility concept [TWS+16] (see Section 2.3 for further discussion on the updatability of nodes in a transaction). Our consistency mechanism ③ was proposed to overcome this issue with the `ConsistencyBlock`. Depending on the connecting time of clients, each client may have different algorithms to format the `Channel` object, but their consistent behavior is ensured in a given `ConsistencyBlock`. The discussion on the problem handling was done in Section 4.4.

6.1.5 LyRT versus the State of the Art

Having demonstrated the three case studies implemented in LyRT, we showed that LyRT matches closely the requirements of run-time variability, set in Chapter 2. Chapter 3 presented a systematic literature analysis, in which parts of the third case study, the File Transfer Application, were implemented with respect to the modularity and dynamic activation as summarized in Table 6.2. Even if we cannot make a direct comparison between those approaches and LyRT, at least it serves as a good reference. Table 6.3 shows the feature comparison of LyRT with existing run-time variability approaches, which have been discussed in Chapter 3.

6.1.6 Other Assessments

Although it works at the level of runtime, LyRT is based on the role concept and inspired by CROM [KLG+14]. CROM itself is also derived from the 26 classifying features of role which

Table 6.2: Reference to various implementations of the File Transfer Application.

Category	Language	Reference
Mixins	Jam [ALZ00]	Snippet 3.1
Traits	ScalaTraits [OAC+04]	Snippet 3.2
Feature-Oriented Programming (FOP)	Jak [BSR04]	Snippet 3.3
Meta-Programming (MP)	Iguana/J [RC02]	Snippet 3.4
Aspect-Oriented Programming (AOP)	AspectJ [KHH+01]	Snippet 3.5
Context-Oriented Programming (COP)	ContextJ [AHHM11]	Snippet 3.6
Role-Oriented Programming (ROP)	OT/J [Her05]	Snippet 3.7

have been presented in Chapter 2. Hence, it is interesting to show how many features LyRT can fulfill as compared to formal CROM[2] [KBGA15] and other ROP languages.

6.1.6.1 Classification According to the Role Features

In this section, we want to show which role features, compiled by Kühn et al. [KLG+14], are mapped to LyRT. The result is summarized in Table 6.4. The feature analysis of ROP languages is discussed by Kühn et al. [KLG+14] and in SCROLL [LA15]. An analysis of formal CROM is also available [KBGA15]. Below, we discuss which of those classifying features that LyRT supports and how LyRT addresses them.

1. **Roles have properties and behaviors.** (■: supported)
   Like a typical object-oriented class, a role can define properties and methods. The `Developer` (Snippet 6.2) and `Accountant` (Snippet 6.3) roles are examples.

2. **Roles depend on relationships.** (□: not supported)
   Although LyRT offers the `invokeRel` method to invoke a method of a collection of core objects (Snippet 6.3), there is no relationship type supported. Hence, roles do not depend on any relationship.

3. **Objects may play different roles simultaneously.** (■: supported)
   In LyRT, objects play different roles simultaneously and also execute the role's methods concurrently due to the support for activation at the instance and the thread level. The behavior of the `Channel` object in the File Transfer Application, described in Section 6.1.4, serves as an example.

4. **Objects may play the same role (type) several times.** (■: supported)
   Role's binding and unbinding operations takes place at run time. There is no restriction to rebind the same role type. For example, the `Channel` object can bind and rebind to the `LZ` compression role several times.

5. **Objects may acquire and abandon roles dynamically.** (■: supported)

[2]CROM has been formalized. The formally validated CROM is also known as formal CROM

Table 6.3: Feature comparison of LyRT with various approaches.

	R1	R2	R3	R4	R5	R6
	Modularity	Dynamic Activation	Late Variants Adoption	Object-Level Tranquility	Failure Handling	Continuous Deployment
Language Facilities						
Inheritance	■	□	□	□	□	□
Mixins [BC90, ALZ00, Moo86]	■	□	□	□	□	□
Traits [SDNB03]	■	□	□	□	□	□
Dynamic Traits [SD05]	■	■	□	□	□	□
Role Object Pattern [BRSW98]	■	■	□	□	□	□
Subject-Oriented Programming (SOP)						
Subject Composition [HO93, HOM95]	■	□	□	□	□	□
Subjective Dispatch [SU96]	■	■	□	□	□	□
Feature-Oriented Programming (FOP)						
Software Product Lines (SPLs)[1]	■	□	□	□	□	□
Dynamic Software Product Lines (DSPLs)[2]	■	■	⊞	□	□	□
Meta-Programming (MP)						
Iguana/J [RC02]	■	■	■	□	□	□
Reflex [TNCC03]	■	■	□	□	□	□
Geppetto [RDT08]	■	■	■	□	□	□
Aspect-Oriented Programming (AOP)						
Static Weaving Mechanisms [KHH+01]	■	□	□	□	□	□
Dynamic Weaving Mechanisms[3]	■	■	⊞	□	□	□
Context-Oriented Programming (COP)						
Local Activation[4]	■	■	□	□	□	□
Global Activation[5]	■	■	⊞	□	□	□
Global Activation with Conflict Resolution[6]	■	■	⊞	⊞	□	□
Role-Oriented Programming (ROP)						
Static Roles [BA12]	■	□	□	□	□	□
Dynamic Roles [GØ03, HNL+06]	■	■	□	□	□	□
Relational Roles [BGE07]	■	⊞	□	□	□	□
Contextual Roles[7]	■	■	⊞	□	□	□
Dynamic Software Updates [SAM13]	□	□	■	■	□	□
LyRT	■	■	■	■	■	■

■: supported, ⊞: partially supported, □: not supported

[1] [Pre97, BSR04, ALRS05, LKKP06, Gri00, LSSP06, MO04, KAB07, AKLS07]

[2] [GS12, RSPA11, TCPB07, LK06, DMFM10, INPJ09]

[3] [BHMO04, NAR08, SCT03, VSV+05, Bon04, VBAM09, PDFS01, SMCS04, AGMO06, JZ10]

[4] [CH05, AHHM11, HCH08, Sch08, SP08]

[5] [AHM+10, SGP12b, LASH11, KAM15]

[6] [KAM11, GCM+10, Car13, GMC08, GMCC13, Bai12]

[7] [Her05, BBVDT06, TUI05, KT09, LA15]

Table 6.4: Comparison of LyRT with role-based approaches based on the role features they support.

Feature	Chameleon [GØ03]	OT/J [Her05]	Rava [HNL+06]	power.Java [BBVDT06]	Rumer [BGE07]	NextEJ [KT09]	JavaStage [BA12]	SCROLL [LA15]	formal CROM [KBGA15]	LyRT
1	■	■	■	■	■	■	■	■	■	■
2	□	⊞	□	⊞	■	⊞	□	□	■	□
3	■	■	■	■	■	■	■	■	■	■
4	■	■	□	■	■	■	□	■	■	■
5	■	■	■	⊞	■	■	■	■	⊘	■
6	□	■	□	■	■	□	■	□	■	□
7	■	□	■	■	⊞	■	■	■	■	■
8	□	■	□	■	□	■	■	■	□	■
9	■	□	□	■	□	■	□	■	⊘	■
10	■	■	■	■	■	■	■	■	■	■
11	■	■	■	■	■	■	■	■	■	■
12	■	■	■	■	■	■	■	■	⊘	□
13	□	■	■	■	□	□	■	■	□	■
14	⊞	⊞	□	□	■	⊞	□	■	■	■
15	■	■	■	■	□	■	■	■	■	■
16	□	□	□	□	■	□	□	□	■	□
17	□	□	□	□	□	□	□	□	□	□
18	□	■	□	□	⊞	⊞	□	■	■	□
19	□	■	□	⊞	⊞	■	□	□	■	■
20	□	■	□	■	■	■	□	■	■	■
21	□	□	□	■	□	■	□	■	□	■
22	□	■	□	□	■	□	□	■	■	■
23	□	■	□	□	□	□	□	■	■	■
24	□	■	□	⊞	■	■	□	■	⊞	□
25	□	■	□	□	□	□	□	■	□	■
26	□	■	□	⊞	■	■	□	■	■	■

■: supported, ⊞: partially supported, □: not supported, ⊘: not applicable
The description of each feature is found in Table 2.1. The classification of existing approaches against the role features is not described in this dissertation but taken from literature [KLG+14, LA15, KBGA15].

In an active compartment, a core object can call `bind` and `unbind` methods dynamically. The core object adapts its behavior accordingly.

6. **The sequence of role acquisition and removal may be restricted.** (□: not supported)
   It is not directly supported although it is not difficult to implement. The reason is that LyRT abstracts the context model from which the information of role acquisition and removal is generated (Section 4.2.2).

7. **Unrelated objects can play the same role.** (■: supported)
   Roles are modeled as unrelated types in which they do not require any association to core objects at design time. For example, the `TaxPayer` role can be played by a `Company` compartment as well as a `Freelancer` role (Figure 6.1). Therefore, this feature is fulfilled.

8. **Roles can play roles.** (■: supported)
   Just like a core object, a role can play roles. An instance of the `Person` class `ely` plays a `Freelancer` role in the `TaxDepartment` compartment. The `Freelancer` role then plays a `TaxPayer` role to form a *deep-play*-relation (Figure 6.1).

9. **Roles can be transferred between objects.** (■: supported)
   Although it was not shown in the case studies, this feature can be easily done by the `transfer` operation (see Section 4.3.10). For example, the person `alice` may transfer the `Accountant` role to `bob` by invoking `alice.transfer(Accountant.class, bob)`.

10. **The state of an object can be role-specific.** (■: supported)
    A role is implicitly instantiated during the binding process via the `bind` method. Although two or more core objects can bind the same role type, each of those core objects is bound to a specific role instance. Therefore, the state, encapsulated in a bound role instance, reflects the state of the core object. In the Tax Management System, `bob` and `ely` play the same `Developer` role, but they have different salary scales attached in each `Developer` role instance (Lines 12-13 of Snippet 6.9).

11. **Features of an object can be role-specific.** (■: supported)
    Clearly, when a core object binds a role, it has additional features encapsulated in the role. For example, a `Person` object can be a `Developer` who works and generates a revenue while another `Person` instance is an `Accountant` who generates a monthly salary.

12. **Roles restrict access.** (□: not supported)
    Any kind of constraints has not yet been incorporated in the current implementation.

13. **Different roles may share structure and behavior.** (■: supported)
    Role types are regular Java classes which can inherit from different role types.

14. **An object and its roles share identity.** (■: supported)
    Although an object and a role are two distinct objects, they become a compound object that shares a common identity when bound.

15. **An object and its roles have different identities.** (■: supported)
    As already mentioned in feature 14, an object and its role have different identities. The runtime manages these identities by using relations that can be manipulated through the standard Java 8 stream API (see Chapter 5).

16. **Relationships between roles can be constrained.** (□: not supported)
    As explained in feature 2, there is no first-class relationship supported. Therefore, constraints between roles based on relationships are not supported.

17. **There may be constraints between relationships.** (□: not supported)
    Similar to feature 16, this feature is not supported.

18. **Roles can be grouped and constrained together.** (□: not supported)
    Roles are grouped in a compartment. Nonetheless, grouping them in a `RoleGroup` as defined in formal CROM [KBGA15] is infeasible. Constraints between roles can be neither applied in a compartment nor in a `RoleGroup`.

19. **Roles depend on compartments.** (■: supported)
    Roles can be bound only in a compartment, and their behaviors become effective when their compartment is active. This is the case when an instance `ely` of the `Person` class is a `Developer` in the `Company` compartment and after that `ely` becomes a `Freelancer` in the `TaxDepartment` compartment. So, roles depend on compartments.

20. **Compartments have properties and behaviors.** (■: supported)
    The compartment definition is just like a general class which includes properties and behaviors. `Company` (Snippet 6.7) and `TaxDepartment` (Snippet 6.8) compartments are examples.

21. **A role can be part of several compartments.** (■: supported)
    Since roles and compartments are separated entities by definition in LyRT, a role may belong to any compartment.

22. **Compartments may play roles like objects.** (■: supported)
    The feature is supported by intention. The `Company` compartment plays a `TaxPayer` role in the `TaxDepartment` compartment as shown in Figure 6.1.

23. **Compartments may play roles which are part of themselves.** (■: supported)
    Compartments and roles are loosely coupled, so there is no restriction on this kind of *play*-relation.

24. **Compartments can contain other compartments.** (□: not supported)
    Since there is only one compartment that can be active at the same time in the same thread, nested compartments are not supported.

25. **Different compartments may share structure and behavior.** (■: supported)
    LyRT fulfills this feature due to the fact that a compartment is a standard class.

26. **Compartments have their own identity.** (■: supported)
    Since a compartment is a regular class, it carries its own identity.

6.1.6.2 Discussion of LyRT in Relation to Formal CROM

Influenced by CROM [KLG+14], LyRT intermingles with a number of role features available in CROM. Due to their differences in terms of usage, we can only informally discuss them based on the feature comparison illustrated in Table 6.4. The discussion below highlights the most important features, which are satisfied by either CROM or LyRT but not by both.

Relationship Type. Formal CROM has a unique relationship type that supports multiplicity and cardinality to express the relationship between the participating roles in a compartment (Feature 2). There is no first-class relationship type supported in LyRT. However, from the runtime perspective LyRT offers the `invokeRel` method to invoke a method of a role participating in a relationship.

Constraints. RoleGroup has been incorporated in formal CROM in order to place certain constraints on roles in a relationship even if the constraint between relationships is not supported (Features 16, 17, 18). LyRT does not support any kind of constraints.

Inheritance. According to Features 13 and 25, roles and compartments in formal CROM do not support inheritance semantically. Since roles and compartments are defined as standard object-oriented classes, LyRT supports these features. However, the *play*-relation is not inherited. For example, a core object `S` plays a role `R`. A subclass `O` that inherits from `S` does not hold the inheritance of the played `R` role. OT/J [Her05] is a role-based language that tackles this problem [HHM04].

Deep roles. Formal CROM supports only flat roles meaning that a core may play roles, but the played role cannot play other roles. Therefore, the *deep-play*-relation or the role-playing role feature is not fulfilled (Feature 8). LyRT supports this feature.

Run-time features. CROM is a model which cannot provide some of the run-time aspects. For example, the ability to dropping and acquiring roles (Feature 5) as well as transferring roles dynamically (Feature 9) are not applicable for formal CROM. These features are fully supported in LyRT.

6.2 Performance Evaluation

Generally, run-time support for adaptation has a negative impact on system performance because of the adaptation management. In case of LyRT, this is in particular due to variants management and late method binding. Performance degradation has been observed in COP [AHH+09, SGP12b] as well as ROP [SC17, Leu17] compared to their respective baselines, and is also observable in LyRT. In this section, various micro-benchmarks are presented to quantify the run-time overhead of LyRT.

6.2.1 Experimental Setup

Benchmarking a Java application is challenging due to various non-deterministic factors [GBE07]. Non-determinism is caused for instance by JIT compilation, optimization in the JVM, and garbage collection. Thus, we write benchmarks in a reliable micro-benchmark framework, called Java Microbenchmarking Harness (JMH)[3], to minimize bias. The results are captured when the benchmark reaches the steady state after a number of warm-up

[3]JMH accessed on August 25, 2017: `http://openjdk.java.net/projects/code-tools/jmh/`

iterations. We chose 20 warm-up iterations and another 20 iterations to observe the actual performance. These 40 iterations in total were forked to run on three different JVM instances in order to improve the uniform distribution of the result. The result of each benchmark is the average of 60 iterations, i.e., 20 iterations in each fork, with a confidence interval of 90% to 95%.

The benchmarks were run on an Apple MacBook Pro with 2.2GHz Intel Core i7 and 16GB of RAM running macOS Sierra version 10.12.6 and Oracle Java SE 8 (build 1.8.0_92-b14). The JMH version is 1.19. In some benchmarks, we measure the performance of Javassist [Chi00], a bytecode rewriting library, to analyze both anticipated and unanticipated adaptation. The Javassist[4] version, used in the benchmarks, is 3.21.0-GA.

Several benchmarks were set up to evaluate the execution time of the following points:

- **Method Dispatch:** shows the overhead of method dispatch compared to the standard invocation.
- **Partial Method Invocation:** demonstrates the performance of stacking behavior of the role-playing-role feature compared with delegation.
- **Adaptation:** exhibits the cost of role binding, unbinding and transferring for both anticipated and unanticipated adaptation.
- **Adaptation versus Method Invocation:** depicts the trade-off of using LyRT by revealing the correlation between adaptation and method invocation.
- **Consistency:** compares the cost of method invocation in a consistency block to the one without.
- **Checkpoint and Rollback:** highlights the time and space overhead to perform the checkpoint and rollback operation.
- **Memory and CPU Consumption:** shows the overhead of memory and CPU consumption.

6.2.2 Method Dispatch

Setup
This benchmark measures the execution time overhead of LyRT's method invocation, which is lifted or dispatched to a role. This benchmark contains a core object with a single method returning a dummy string value. The evaluation is performed on a method call through different techniques such as standard invocation, reflection and the `invokedynamic` instruction [CO14]. For LyRT, we add two additional roles `R1` and `R2`. Each role contains a method defined similarly to the one in the core object in order to show the performance of the method dispatch. In one configuration, the core object binds a role `R1` while in other it binds `R1` and `R1` binds role `R2`. In this benchmark, we solely focus on the lifting aspect or dispatching mechanism in LyRT while the lowering aspect will be discussed in a subsequent benchmark.

[4]Javassist accessed on August 25, 2017: `http://jboss-javassist.github.io/javassist/`

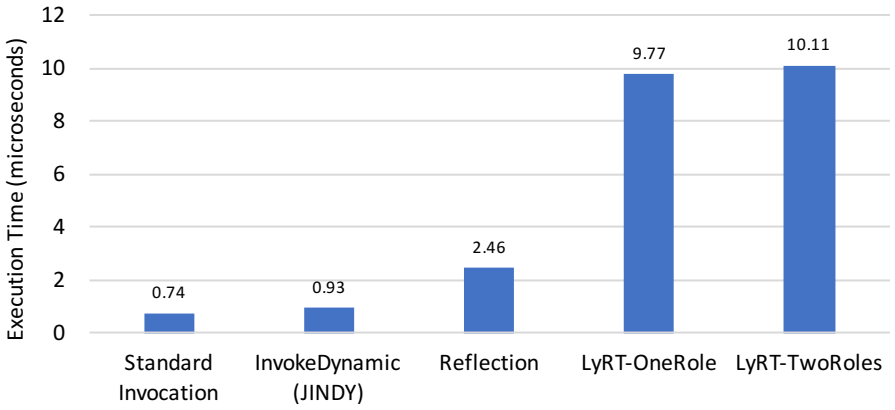


Figure 6.14: Performance of various techniques on method invocation.

Discussion
As depicted in Figure 6.14, the standard invocation is the fastest among all, followed by the `invokedynamic` instruction, which is insignificantly slower. This conforms to the results of Conde et al. [CO14]. Reflection, which was expected to be considerably slower than the aforementioned techniques, performs quite well in Java 8 and is only about 3.3 times slower. The reason is that recent JVM versions inline reflective methods when a certain threshold of repetitive invocations is reached. Although LyRT uses the `invokedynamic` opcode for its method invocation (see Section 5.3.2), its performance is 11 to 13.5 times slower than the pure `invokedynamic` opcode and the standard invocation respectively. This overhead is caused by the fact that, in each invocation, LyRT looks for an active compartment and hashes a method to search through the cached method table in order to find an appropriate role to be invoked. Although this is a very expensive mechanism, it enables LyRT to support contextual method dispatch and therefore dynamic system behavior at run time. The number of roles to be bound has no significant impact on the performance of LyRT since the cached method table stores only the most recent method affecting the dispatching rules. For instance, in the configuration of LyRT-TwoRoles, the method of `R1` was overridden by the method of `R2` due to late binding. Therefore, the process to search through the cached method table remains the same and the performance is eventually comparable.

6.2.3 Partial Method Invocation

Setup
LyRT allows a bound role to play other roles; this allows us to construct a stacking behavior by visiting all roles in the *deep-play*-relation (i.e., partial method in COP domain). The prominent baseline for this benchmark is delegation due to its similarity and instance level operation. A set of classes, whose instances are connected to each other via a method call, is created. Each class contains a method which calls another method of its kind to form a method chain similar to our stacking method. In case of LyRT, a core object is bound to a role, and the role is bound to another role to form a *deep-play*-relation. Each role contains a method which calls its player via `invokePlayer` method. For this setup, roles are stacked together until the fifth level is reached. Although there is no study to show that the fifth level is enough in practical use, this number has been employed in the COP domain to

benchmark the activation of multiple layers.

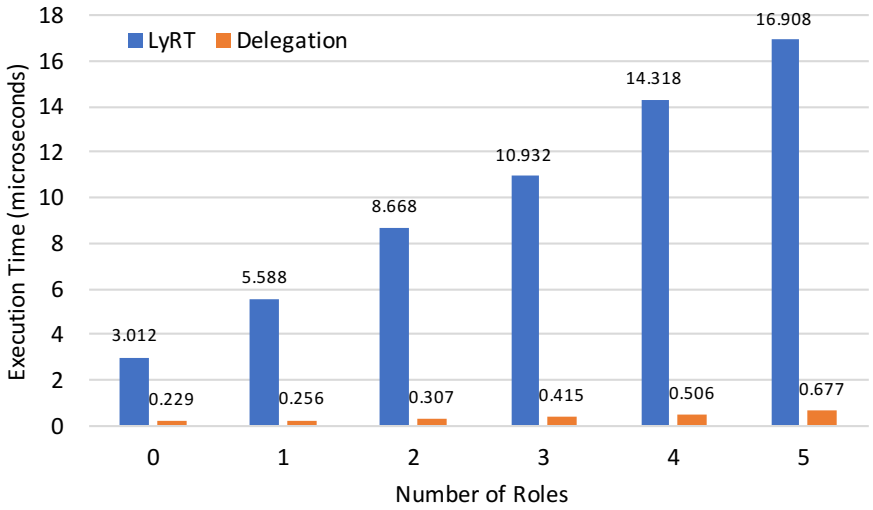


Figure 6.15: Performance of partial method invocation by stacking roles up to fifth level.

Discussion

Figure 6.15 shows the performance comparison between LyRT and its delegation counterpart. The x-axis categorizes a number of partial methods to be invoked while 0 means invoking a method of the core object itself. The value 1 means the method is executed on a role and then lowered to the core which is twice in total. In case of 5 roles, the overall execution is 6. The y-axis shows the time taken in microseconds to perform each category. In both cases, the overhead is linearly increased with respect to the number of partial methods. In average, the slowdown of both approaches is a factor of 1.4 for LyRT and 1.2 for delegation.

However, partial method invocation in LyRT is observed to be around 25 times slower than that of the delegation which is a huge performance impact. The reason is that LyRT always checks whether the compartment is active or not to select the appropriate cached method table for invocation. Additionally, the passing method and its parameters need to be packed and hashed in order to find the appropriate role to be invoked. This process of method selection is rather expensive compared to the actual method invocation itself as shown in the previous benchmark (see Section 6.2.2). The performance of delegation is highly optimized by modern JVM as it has been investigated by Götz and Pukall [GP09]. Moreover, the benchmark on delegation does not consider contextual dispatch and unanticipated adaptation, and methods are statically invoked by an interface.

The result also indicates that the overhead keeps increasing excessively compared to the delegation when the number of roles is on the rise. Practically, we expect that there are only a few roles needed to construct a partial method. For example, only two roles were required in the case study of the File Transfer Application, discussed in Section 6.1.4. In this case, the overhead is reduced.

6.2.4 Adaptation

Setup

Adaptation of an object is an ability to swiftly change its behavior based on context. There are three operations provided by LyRT to support adaptation: binding, unbinding and transferring roles. The rebinding operation is also available but it is a combination of the unbinding and the binding operation. Thus, we exclude it from the benchmark. The remaining operations are measured for anticipated and unanticipated adaptation. In this benchmark, we compare adaptation in LyRT with a bytecode rewriting library, called Javassist [Chi00].

Javassist is powerful as it allows programmers not only to define a new class but also to modify an existing class before it is initially loaded into JVM, resembling unanticipated adaptation. Unlike its competitors, Javassist provides an API for both source-level and bytecode-level translation. Whenever the source-level API is used, programmers can transform a class with source-level vocabularies like `class` and `method` without knowledge of bytecode specification. Therefore, dynamic change of object behavior is possible and easy to implement. Despite transformation capability, Javassist has a low overhead while keeping ease of use [CN03]. According to the state-of-the-art survey illustrated in Table 6.3, only Iguana/J [RC02], Geppetto [RDT08], and DSU [SAM13] should be considered to benchmark for the support of unanticipated adaptation, but we decided not to do so because of the following reasons. DSU is a JVM-specific solution, which is not portable, while LyRT is an architectural approach applicable to many languages. Similar to DSU, Iguana/J is implemented via a dynamic library integrated tightly to the JVM; it is therefore not portable. Geppetto has Smalltalk as its host language making a comparison more challenging.

The setup for this benchmark is to have a number of roles scaling from 10, 100, 1,000, 5,000, and 10,000 to be bound to the same number of core objects. This procedure is also applied to unbinding and transferring operations for anticipated and unanticipated adaptation. For Javassist, a method of the core object is redefined using the source-level API and dynamically reloaded featuring the scaling number. In order to reload a newly redefined class, Javassist by default relies on Java Platform Debugging Architecture (JPDA) forcing the benchmark to run in a debugging mode. Hence, we extended Javassist to reload classes dynamcially by using Java Agent[5]. As an agent, this custom class has an insignificant effect on the benchmark because it runs once per JVM at load time.

Discussion

The result of this benchmark is shown in Figure 6.16. The x-axis shows different categories of benchmarks for 10, 100, 1,000, 5,000, and 10,000 roles. For Javassist, this is a number of times to redefine a class method. The UA prefix in the legend stands for unanticipated adaptation. The y-axis shows an execution time of each category in milliseconds.

Generally, the binding and unbinding operations have a minimal overhead, and their performance is comparable for both anticipated and unanticipated adaptation in the measurements less than 1,000 roles. More than that, the unbinding operation takes a longer time to be completed than the binding one. The reason lies in the unbinding implementation which adds

[5]Java Agent accessed on August 25, 2017: `https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`

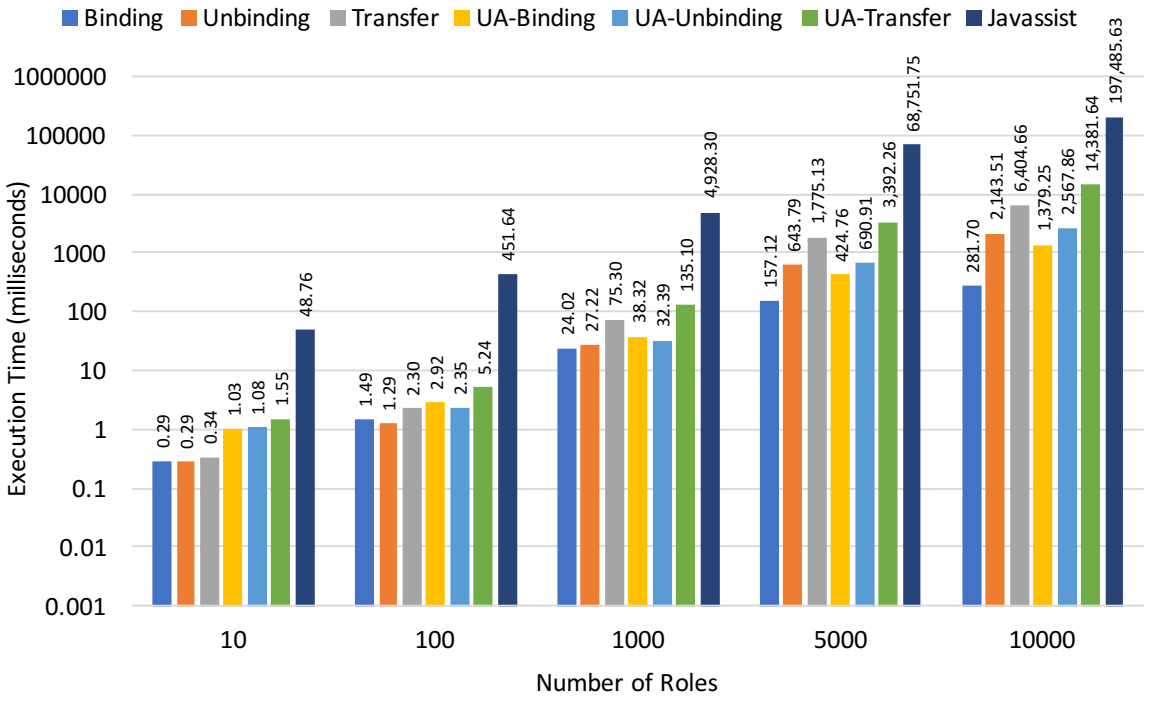


Figure 6.16: Performance of adaptation (UA prefix stands for unanticipated adaptation).

additional logic to check if there is a *deep-play*-relation to be performed, besides adjusting the cached method table.

The transferring operation incurs more overhead as compared to the binding and unbinding operations. The reason is that during a transferring process the binding relation in the lookup table needs to be detached from a source and attached to a destination. Therefore, the cached method table is adjusted on both sides. In contrast to this, binding and unbinding operations update the binding relation and the cached method table once for the engaging core object.

Regarding unanticipated adaptation (with UA prefix), LyRT typically parses an adaptation XML file and loads roles with a `ClassReloader` in addition to the default process performed in the anticipated adaptation. As a result, any adaptation operation, i.e., UA-Binding, UA-Unbinding, and UA-Transfer, for unanticipated adaptation shows an additional overhead compared to its anticipated adaptation counterparts. In average, unanticipated adaptation is about two times slower than anticipated adaptation.

Javassist, in all cases, introduces a significant overhead due to the recompilation of the core object to add a new implementation of a method. In average, Javassist is 12 and 23.5 times slower than the unanticipated and anticipated adaptation respectively. Even if Javassist performs poorly on adaptation, the method invocation of the adapted object runs very fast (see Section 6.2.5). Javassist adapts behavior at the class level for which core objects of the same type are eventually affected by the new definition. LyRT applies adaptation at the instance level, and it is not required to recompile the core object despite the addition of new roles. In fact, newly introduced roles must be compiled before they can be loaded into the

JVM. We consider this compilation to be part of the preparation phase of unanticipated adaptation rather than the unanticipated adaptation process itself. The compilation time for LyRT, therefore, does not count in the benchmark.

6.2.5 Adaptation versus Method Execution

Setup

The overall performance of adaptive systems results from the efficiency of adaptation and the execution of newly adapted entities. To further illustrate the relationship between them, we wrote a benchmark to simulate the adaptation process by scaling from 10 to 1,000 roles which are bound to the same number of core objects. We consider only the binding operation to represent the adaptation in LyRT. After the binding is completed, the respective core objects invoke a certain number of methods ranging from 1,000 to 1,000,000 times. For Javassist [Chi00], we setup the same procedure as described in Section 6.2.4. For 1,000 roles and 1 million method invocations, it is enough to show the correlation of the two approaches.

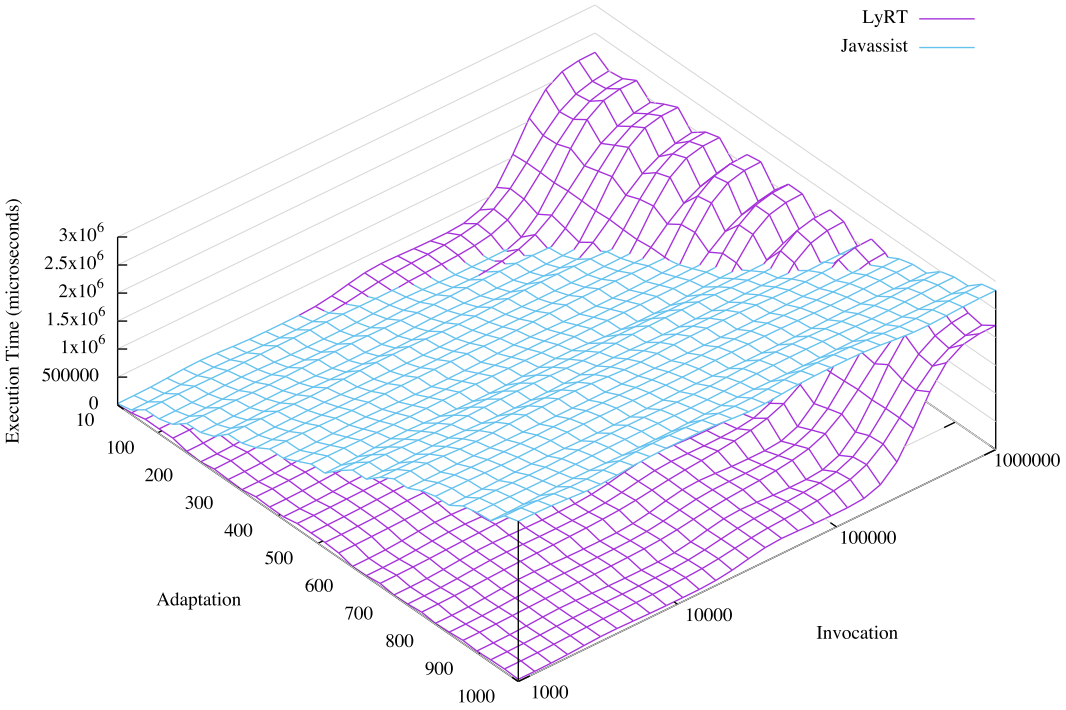


Figure 6.17: Adaptation versus method invocation.

Discussion

Figure 6.17 depicts the result of the relationship between adaptation and method execution of LyRT and Javassist. As expected, LyRT performs well on adaptation although the number of roles keeps increasing (see execution time axis vs. adaptation axis). However, it imposes a huge overhead on method invocation when the number of invocation is more than 100,000 times (see execution time axis vs. invocation axis). This overhead results from the support of contextual dispatch and unanticipated adaptation as presented in Section 6.2.2 and

Section 6.2.3. In contrast, Javassist spends most of the time on adaptation and apparently invokes the method at the speed of standard execution. Chiba and Nishizawa [CN03] claimed that the overhead of Javassist's method execution is about ten nanoseconds which is negligible as the JVM inlines the method. This JVM optimization technique does not hold true for LyRT because roles are still separated entities and JVM has no knowledge about them.

From the result, we can deduce that LyRT is suitable for a dynamic system which is highly adaptable involving a great number of objects to be adapted at once, but it should not be used in a static or rarely dynamic system.

6.2.6 Consistency

Setup
In order to achieve consistent behavior, a consistency block is used. This benchmark analyzes the overhead of entering and leaving the consistency block as well as the method execution within the block compared to the one without using consistency blocks.

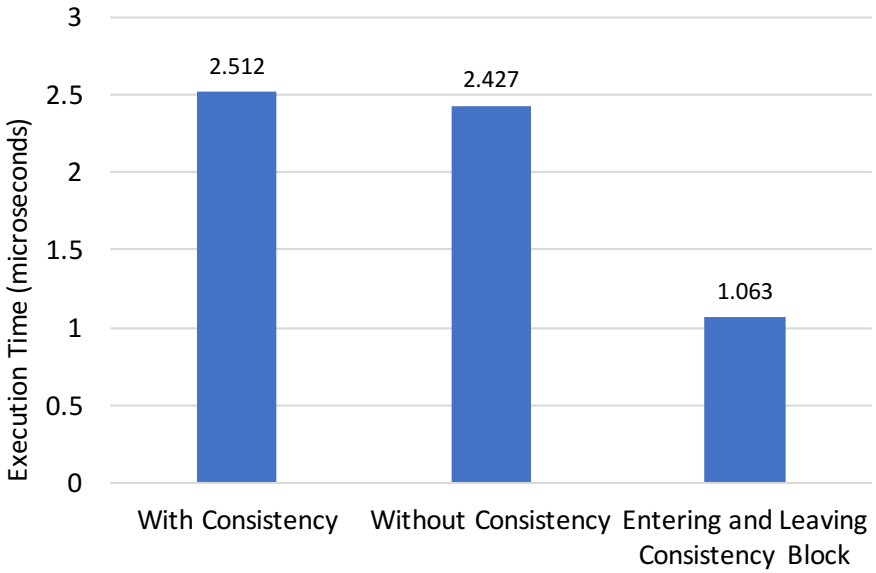


Figure 6.18: Overhead of consistency support.

Discussion
Figure 6.18 shows the overhead of method invocation executing in a consistency block corresponding to one executing without consistency block. The y-axis shows the execution time in microseconds. There is no significant overhead be observed as the cached method table of engaging core objects remains untouched when the runtime detects a running consistency block. The cost of entering and leaving the consistency block is small, and it happens only one time per block regardless of multiple method invocations. A slight overhead occurs when the running consistency block has expired and the cached method table is adjusted according to the prescribed adaptation. So, the cost of achieving consistent behavior is relatively minimal. In average, a method executed with a consistency block is 1.04 times slower than its execution without a consistency block. The whole consistency

including entering and leaving the block is 1.39 times slower than the standard execution in LyRT.

6.2.7 Checkpoint and Rollback

Setup

This benchmark displays the overhead of the recovery support in LyRT. We have separate measurements for checkpoint and rollback within a range from 10 to 10,000 roles. The baseline for this benchmark consists of a class containing a `HashMap<Integer, Object>` data structure to maintain its roles, in which `Integer` represents a hashcode of a role and `Object` references to the role. The role is a typical class defined with an empty method returning a dummy string. To do a checkpoint, we introduce an `ArrayDeque<KeyValue>` to hold a list of a key-value pair of the role and its hashcode collected from all core objects. In case of a benchmark for 10 roles, 10 core objects are created; each has one role resulting in 10 key-value pairs for the checkpoint.

Since we rely on Kryo library to perform a deep copy for checkpoint (see Section 5.6.1), we also measure a pure deep copy for the same amount of roles to back up the discussion. Additionally, the space overhead of the checkpoint will be presented. In general, Java has no operator like `sizeof()` as in C language to determine the actual memory allocation for a particular object. Therefore, we cannot directly measure the space overhead on the checkpoint since the checkpoints are kept in memory. Instead, we serialize the roles with a default format onto a disk and measure the disk usage as a reference for the checkpoint size.

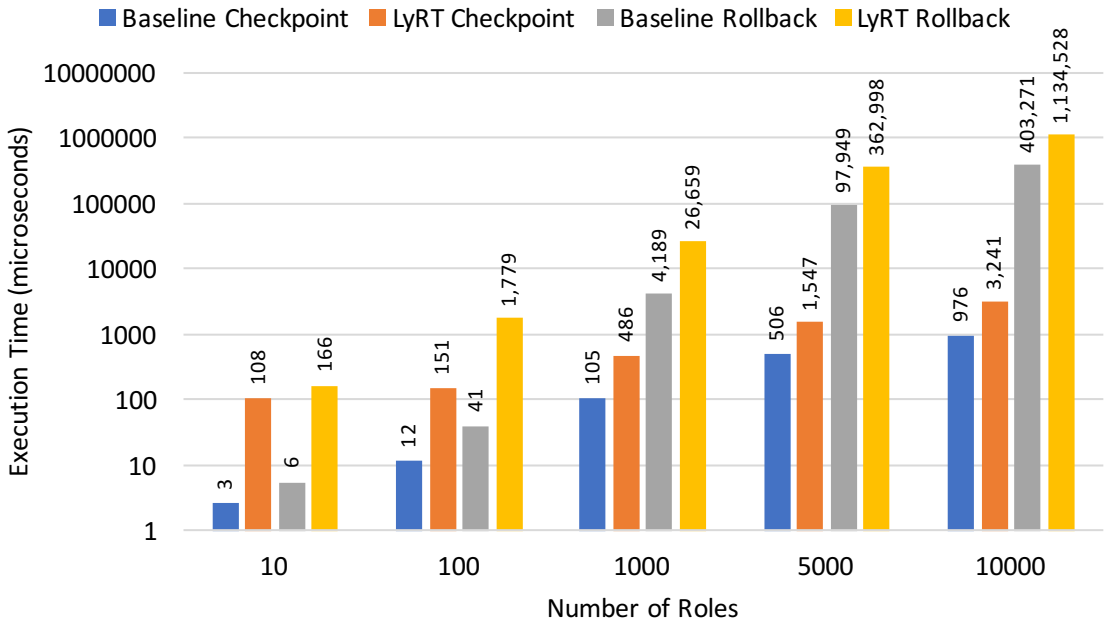


Figure 6.19: Overhead of the checkpoint and the recovery.

Discussion

Figure 6.19 illustrates the time taken for doing a checkpoint and a rollback scaling with the number of active roles ranging from 10 to 10,000. Regarding the checkpoint, LyRT introduces

significant overhead, roughly 13.06 times slower than the baseline. The reason is that LyRT has a larger data structure to be maintained, not only playing relations but also deep-playing relations. The main overhead of the checkpoint is the object reference assignment which changes from the old role to the newly copied role in the lookup table. This process has to be done for each relation in the lookup table before it is pushed onto the stack. In the case of role-playing-role, the player reference in the lookup table needs to be replaced as well. This replacement process is expensive as it runs $O(n^2)$, where n is a relation. In contrast, the baseline runs only $O(n)$, where n is a key-value pair. Despite producing a high overhead, the execution time in LyRT is relatively small, i.e., 3.24 milliseconds for 10,000 roles, which is fast enough with respect to usability.

Both LyRT and the baseline introduce an exceeding overhead on rollback compared to their checkpoint. Although the runtime uses a deep copy to clone roles inside a data structure maintaining roles (a lookup table for LyRT and a `HashMap<Integer, Object>` for the baseline) for checkpoint which seems to be time-consuming, Kryo handles this task efficiently. Times taken for pure copying of 10, 100, 1,000, 5,000, 10,000 roles are 0.23, 0.41, 1.96, 8.31, and 16.12 microseconds respectively. Rollback involves in deleting the current data structure[6] and restoring data in a checkpoint to its respective data structure. This process requires $O(n^2)$ execution, where n is a relation, since we have multiple objects and a set of available roles in the checkpoint to be matched.

However, the rollback in LyRT entails even more overhead than its baseline counterpart as the runtime needs an extra time on adjusting the cached method table for each binding relation after the lookup table is restored. This process is essential in order to have runtime ready for execution. In average, the overhead of rollback in LyRT is 17.23 times higher than that of the baseline.

Unlike checkpoint which is performed in each adaptation, rollback is done only when a failure is detected. Therefore, it should not affect the normal execution of a program apart from the significant overhead. Furthermore, LyRT treats the checkpoint and rollback as a feature which can be turned off by calling `Registry.setRecovery(false)` if it is not needed.

The space overhead of the checkpoint in LyRT depends on the implementation of roles and their amount. As already explained in Section 5.6.1, although we push the whole lookup table (`ArrayDeque<Relation>`) in a compartment to the stack, only roles are serialized (i.e., similar to copy by value). Core objects and their compartment are copied by reference. Hence, the space overhead is relatively minimal. Table 6.5 shows the overhead per checkpoint based on the number of roles. Each role contains only an empty method without specifying any fields. Additionally, we also provide information if the checkpoint is performed for a whole lookup table for comparison.

6.2.8 Memory and CPU Consumption

Setup

Memory and CPU consumption is another interesting aspect to see how much overhead is introduced compared to the baseline. Rather than demonstrating a complete set of memory

[6] LyRT uses a central data structure stored in each compartment whereas the baseline maintains it in each core object.

Table 6.5: The space overhead per checkpoint in Kilobytes.

	Number of Roles				
	10	100	1,000	5,000	10,000
Only Roles	0.05	0.31	2.95	14.67	29.32
Lookup Table	6.43	16.58	118.01	569.88	1,135.30

and CPU usage of a whole application, we highlight only the consumption for the binding operation which could be a basis for estimation.

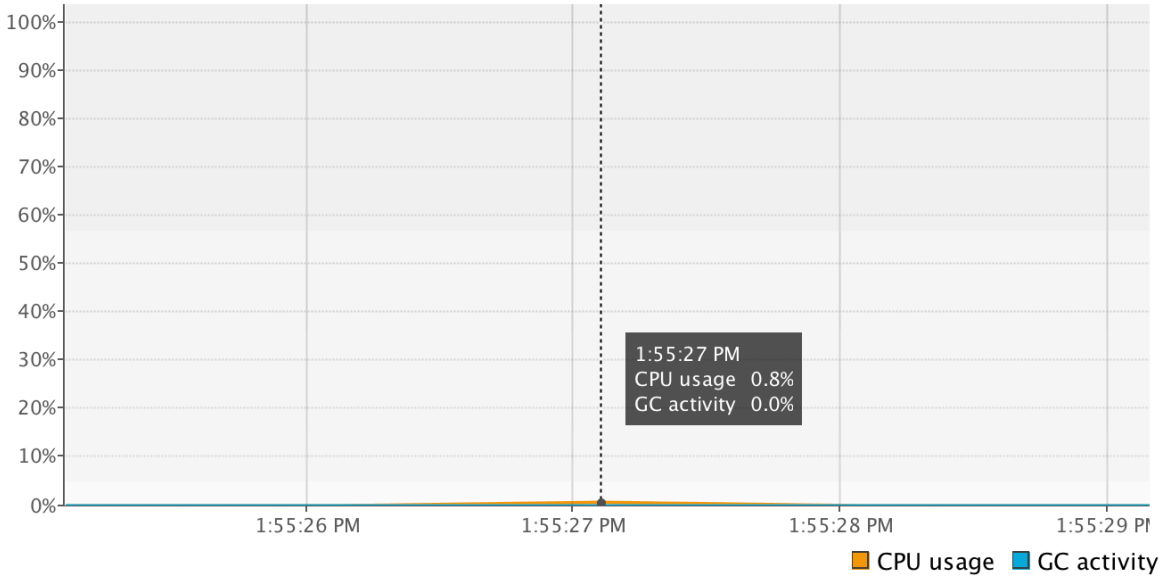
The baseline of this benchmark is a typical object containing a `HashMap<Integer, Object>` data structure to hold references to its roles. While an `Integer` stores a hashcode of a role, `Object` contains a reference to the role itself. Each object has one role. We scale the benchmark from 1,000 to 10,000 roles in which the number of cores equals to the number of roles. Below that value, memory footprint is too small to be projected. For LyRT, we generate core objects ranging from 1,000 to 10,000 while each of them binds to a role.

Benchmarking memory and CPU consumption in Java is prone to bias because of a garbage collector. Although there is no option to suppress the garbage collector, an alternative method is to increase the heap size to be large enough to accommodate the benchmark. After several trials, we found out that within the scale of 10,000 cores and roles, 2GB of heap size is sufficient to prevent the garbage collector from kicking in. We profile the memory and CPU usage with a default tool given by Oracle known as VisualVM[7] (i.e., executed by a `jvisualvm` command line). Figure 6.20 and Figure 6.21 show a snapshot of CPU and memory utilization of the baseline and LyRT produced by the tool. To visualize the utilization, we allow the benchmark to run but being idle for 5 seconds. After the benchmark is completed, it stays inactive for another 2 seconds.

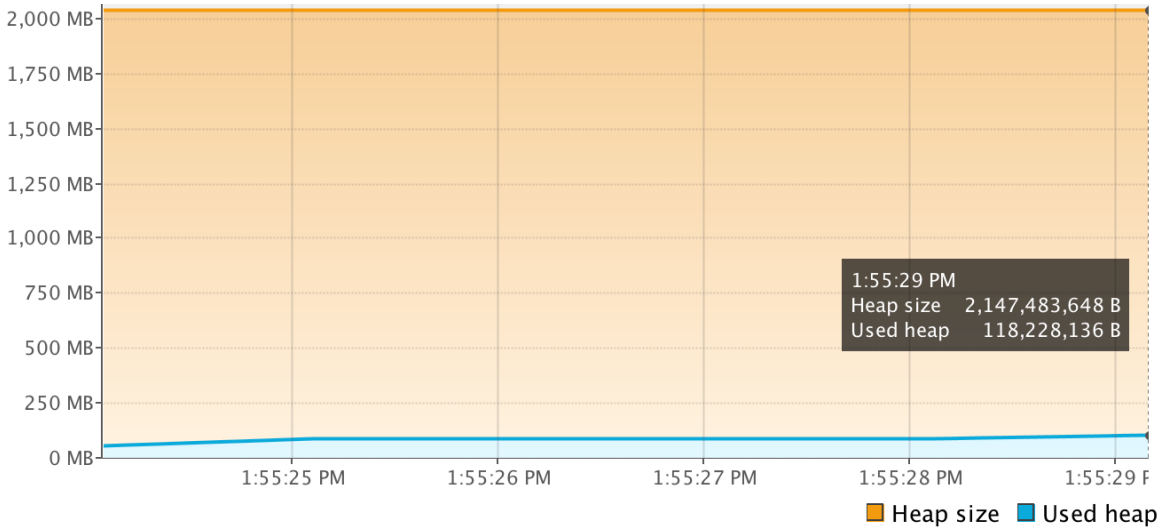
Discussion

Figure 6.22a shows a comparison of memory consumption between baseline and LyRT. Note that both baseline and LyRT have a pre-allocated memory of about 85MB for the 2GB heap size setting (see Figure 6.20b and Figure 6.21b). This number is lower for the smaller heap size. For 1,000 and 5,000 roles, the baseline projects the same amount of memory usage. Although it seems incorrect, the reason would be that it uses the pre-allocated memory. That is why benchmarking below 1,000 roles yields insignificant memory usage for the baseline. LyRT utilizes memory 1.33, 2.78 and 3.27 times more than the baseline for 1,000, 5,000 and 10,000 roles respectively. In average, memory overhead in LyRT is 2.46 times more than that of the baseline. The reason is that LyRT uses JINDY [CO14] to support `invokedynamic` opcode which generates extra `Callable` classes for each core and each role. More elaboration can be found in Chapter 5. In order to speed up the method lookup, LyRT uses these generated classes to build a cached method table with additional meta-data which occupies some spaces. Apart from that, the lookup table and role management code also contribute

[7]VisualVM accessed on August 27, 2017: `https://visualvm.github.io`

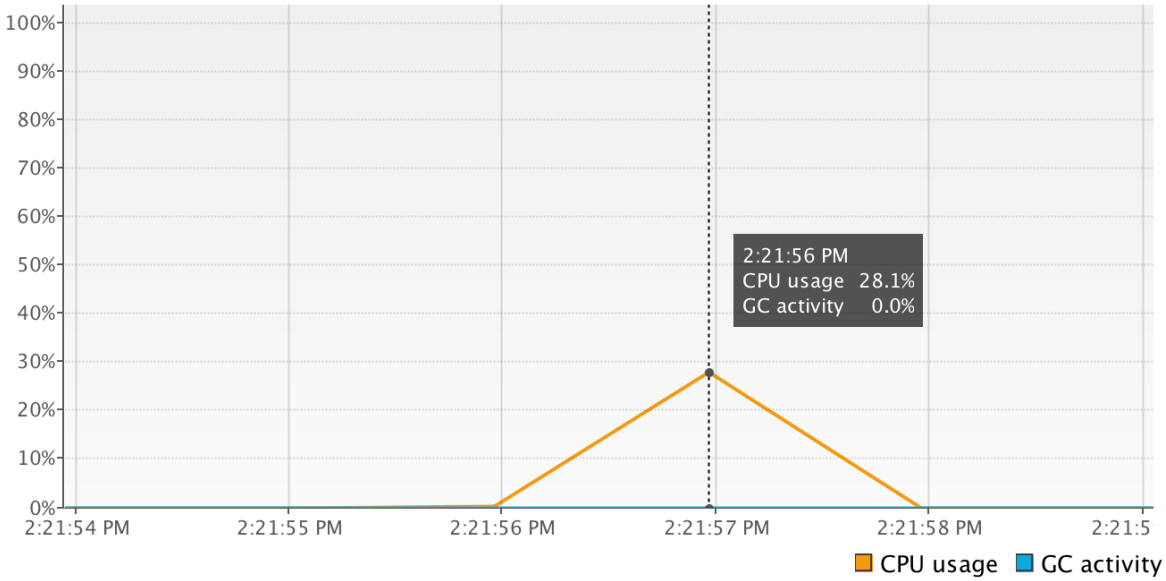


(a) CPU consumption of the baseline.

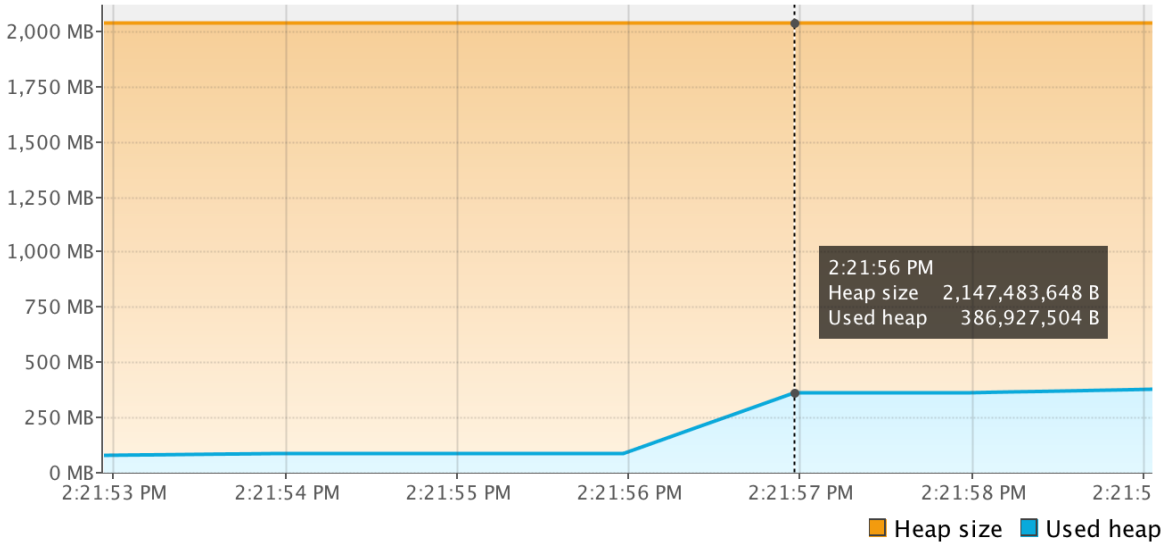


(b) Memory consumption of the baseline.

Figure 6.20: A snapshot of CPU and memory consumption of the baseline for 10,000 roles (GC: Garbage Collector).



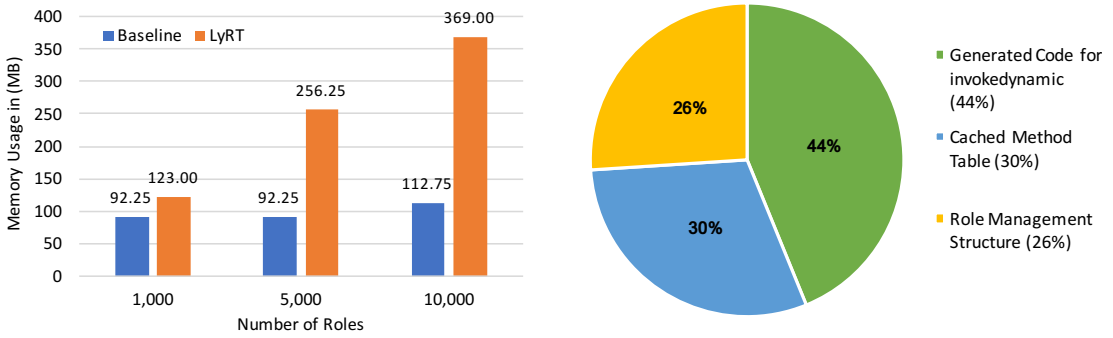
(a) CPU consumption of LyRT.



(b) Memory consumption of LyRT.

Figure 6.21: A snapshot of CPU and memory consumption of LyRT for 10,000 roles (GC: Garbage Collector).

to the memory usage.



(a) Memory usage of the baseline and LyRT. **(b)** Memory dissection in LyRT.

Figure 6.22: Memory consumption.

Figure 6.22b breaks down the memory utilization in LyRT. A large portion of memory usage (44%) is spent on the `invokedynamic` code generation. The cached method tables, which include the cache for core object, lifting, lowering and collaboration functions, account for 30%. The rest is for the role management data structure including the lookup table.

While an average CPU consumption in LyRT is 28.1%, the baseline utilizes only 0.8% for this benchmark. This result is directly perceived from the VisualVM profiling tool. The explanation of the huge overhead is similar to that of memory consumption. Note that the measurement of CPU consumption is solely for the benchmark and it does not count for the clock of the garbage collector because there is no observable garbage-collection activity.

6.3 Discussion

We used the case studies to evaluate the practical feasibility of LyRT in the domain of software systems which requires adaptation at instance level. Crucially, in adaptive software systems, adaptation always involves a performance decline. This also holds for LyRT. Through the benchmarks, we found out that the measured overhead remains mostly acceptable (except for the deep-play relation with more than a few roles) with respect to usability of highly dynamic systems. This section discusses strengths and weaknesses of LyRT and outlines possible future enhancements.

6.3.1 Run-time Versatility

The case studies have shown that LyRT cleanly hosts role-based applications in order to achieve dynamic adaptation with respect to the six requirements derived for run-time variability. We highlight the main features in the following sections.

6.3.1.1 Roles as Variants

Based on run-time variability which separates variants from the base system, utilizing the role concept inherently achieves dynamic adaptation at the instance level due to the decoupling of

role instances from the cores. The notion of compartments further enhances the collaboration between roles and serves as a dynamic scoping of the playing core objects to enable the context-dependent behavior. With our dynamic instance binding mechanism, we realize the role concept to be a computational model for run-time variability.

6.3.1.2 Anticipated Adaptation

The case studies emphasize the need for the proposed dynamic instance binding mechanism to achieve a fine-grained adaptation down to the level of instances. In fact, de facto approaches contributing to run-time variability from different domains like AspectJ [KHH+01] of AOP, ContextJ [AHHM11] of COP and OT/J [Her05] of ROP activate new behavior at the type level adapting all derived instances.

Although there is no consensus showing the pros and cons of type versus instance level adaptation, we believe that adapting objects at the instance level is more flexible from the runtime perspective. Some applications portray perfectly the need for instance-level adaptation, such as the Tax Management System (see Section 6.1.2), which is hardly achieved by type-level adaptation. Additionally, instance-level adaptation, given by the dynamic instance binding mechanism, provides an opportunity for unanticipated adaptation and advanced features such as consistency and rollback recovery.

6.3.1.3 Unanticipated Adaptation

The dynamic instance binding mechanism enables the support of unanticipated adaptation without causing too much overhead in terms of execution time as compared to its anticipated adaptation. This support does not only show the addition of non-obtrusive behavior on-the-fly but also benefits to a system demanding for high availability. The Snake Game is a showcase for demonstrating this feature.

As discussed in Chapter 3, there are not so many systems which enable this capability. First, in order to avoid producing overhead, they opt for compile or load time activation which happens only once during an application life cycle. Therefore, hooking in new behavior at run time is infeasible. Second, in systems which adapt objects at the type level, it is hard to control the precise objects to be adapted without a predicate. Third, unanticipated adaptation faces a major inconsistency issue due to the limited knowledge of all adapted objects. For systems adapting at the type level, it may happen that some instances are idle while others are busy (e.g., with invoking a method). Adapting that given type is undoubtedly problematic for the active instances. In this regard, adapting an object at the instance level has an advantage over that of the type-level adaptation.

Benefitting from instance-level adaptation, LyRT presented a straightforward architecture to gain control of unanticipated adaptation. Only selected instances are adapted. However, unanticipated adaptation is prescribed in an XML configuration file in which object references are needed. Even though we provide an API to query the lookup table, it is still a drawback that developers must be aware of all core objects or roles to be dynamically adapted. Adapting plenty of objects has to be configured carefully; otherwise, it may break the system. Besides, adding new behavior on-the-fly may violate system security which is not in the scope of this dissertation. Moreover, consistency is still an issue, but it is handled by a separate mechanism, discussed in the subsequent section.

6.3.1.4 Consistency

LyRT solves the aforementioned inconsistency problem by providing a consistency block to guard the code block where uniform behavior is required. Executing within this block, core objects are safe to maintain their behavior regardless of adaptation happening asynchronously and randomly. Adaptation happens when the block has expired, and it only affects current thread. Hence, we can minimize the blocking factor of the adaptation in other threads while still maintaining consistency among those threads. However, developers must place this consistency block since they are fully aware of program execution and its expected functionality. That consistency block works on both anticipated and unanticipated adaptation, and produces an insignificant overhead.

6.3.1.5 Rollback Recovery

In order to contribute to high availability and stability of the runtime, LyRT integrates the rollback recovery mechanism to monitor the run-time execution failure arisen from role composition. Although the case study of the File Transfer Application has only two roles, the four possible combinations resulting from compositions of those roles contained at least one composition yielding run-time failure, i.e., `LZ` binds to `AES`. We argue that if a system has a dozen of roles, manual testing for all the possible combinations in order to find bugs is far from trivial. This does not count for roles lately introduced in an unforeseen manner.

The rollback recovery mechanism alleviates this problem by embracing failures at run time through its specialized bug sensor taking advantage of exception handling. Creating checkpoints at each adaptation helps to build composition knowledge to be recovered when a failure is detected. Through the case study and the benchmark, we have demonstrated that the runtime can gracefully recover from the failure without dropping client sessions. However, there are some limitations and open issues, which should be addressed as follows:

How does rollback affect the program execution flow? Once bugs are detected, the runtime rolls back the affected thread to the previous checkpoint and re-spawns the thread. Data might get lost because program flow should resume from where the error is caught after rollback. This is a limitation since Java does not equip with an `on-error-resume` feature. In exchange, we suggest integrating a recovery process in an application protocol as we did in the File Transfer Application.

A failure is detected after several adaptations. Suppose errors are introduced within configuration $AC1$, but the faulting method has never been executed. Then, the runtime adapts to configurations $AC2$ and $AC3$. In $AC3$, the runtime catches the failure of $AC1$. In this case, the system rolls back to $AC2$, which is the wrong choice. We consider it for future improvement.

What is the effectiveness of the bug sensor? The degree of fault tolerance of the proposed mechanism depends on the effectiveness of the bug sensor. Our current implementation relies on the exception handling mechanism to detect the caught and uncaught exceptions at the application level. Nonetheless, JVM-related issues, such as `OutOfMemoryError` that shuts down the JVM, cannot be handled. In this regard, implementing the bug sensor at the JVM level could be more efficient, but it may face compatibility issues between the JVM releases.

6.3.1.6 Continuous Deployment

By taking advantage of the support of unanticipated adaptation and rollback recovery, we can take the continuous deployment feature for granted. This support is essential for adaptive systems demanding for high availability and stability in spite of erroneous situations. A failure, arisen from a particular composition of variants, is reported by the bug sensor to which a fixing version of them can be redeployed allowing the core objects to adapt accordingly. Although trivial, the third case study elegantly demonstrated the need for this feature for highly dynamic systems having hostile compositions.

6.3.2 Overhead

Naturally, enforcing behavioral adaptation at run time entails an overhead in terms of execution time. Our benchmarks confirmed this statement valid. However, the overhead is not a blocking factor for the practicability of LyRT. Despite being small, the case studies show that the overhead is not perceivable by the users. Table 6.6 shows the overhead of each feature in LyRT compared to the baseline counterparts. Although these overheads are considerably high, the actual execution times are relatively small, i.e., a matter of milliseconds for most cases. The chosen baselines are optimal for their respective purpose whereas LyRT is generalized to support all the mentioned features at once. Therefore, the overhead is inevitable. Below, we highlight the causes of the overhead and raise them for future optimization.

Table 6.6: Summary of LyRT overhead regarding its features. Overhead is denoted as a number of times slower (+) or faster (-).

Benchmark	Baseline	Overhead
Method Dispatch	Standard Invocation	+13.50
Partial Method Invocation	Delegation	+25.00
Unanticipated Adaptation	Javassist [Chi00]	-12.00
Unanticipated Adaptation	Anticipated Adaptation	+2.00
Consistent Method Execution[1]	Standard LyRT	+1.04
Consistency[2]	Standard LyRT	+1.39
Checkpoint	Baseline Implementation	+13.06
Rollback	Baseline Implementation	+17.23
Memory Consumption (for binding)	Baseline Implementation	+2.46
CPU Utilization (for binding)	Baseline Implementation	+35.00

[1] A method invocation executed in a consistency block.
[2] A measurement of a method invocation in a consistency block plus the time taken for entering and leaving the block.

Although the implementation was not optimized, we point out sources of penalty for future optimization. The discussion below is based on the current implementation and the chosen data structure for the lookup table as it can be found in Chapter 5.

Two primary sources of performance degradation need to be investigated. On the one hand, the dynamic method dispatch checks an active compartment and associating roles to be dispatched. This checking has to be done for every invocation which contributes to a compelling amount of overhead. This task should be inlined for repetitive method calls. In fact, once a target method is found, its execution speed is comparable to the standard invocation; thanks to the `invokedynamic` opcode. Thus, the point to optimize is to find the optimal process of method selections. On the other hand, since role instances are kept separated from the core object, manipulating these roles is expensive due to the need for listing and comparing processes. Normally, there is an $O(n)$ search over the lookup table (`ArrayDeque<Relation>`), where n is the number of binding relations. Building the cached method table, however, requires an $O(n^2)$ search because of the *deep-play*-relation where it can be reduced to be $O(n\ log(n))$. The chosen data structure to represent the lookup table also influences the overall performance.

Another implementation aspect which should be considered for optimization is not to use Java 8 lambda and stream API very much. We totally rely on these new features for lookup-table manipulation. Instead, a traditional imperative programming style with iterations and `for-each` loops should be employed since it is around five times faster than the functional style lambda and stream. This result is publicly confirmed, and our benchmark also reports similarly. In turn, we lose the sweet spot of readability in functional style if it is omitted.

As a remark, no matter how performance is optimized, we will never reach the speed of the static systems. This is a trade-off between adaptation support and performance of the run-time system.

6.3.3 Practicability

Through the benchmarks, we have scaled the number of roles up to 10,000 to show the performance of adaptation. At this size, the role binding, in general, performs pretty fast; the unbinding takes longer while the transfer takes even longer. In average, it takes 1.5 seconds as we assume the transfer operation will not be used quite often. Whether this number is perceivable by users depends on the program size. Even if there is no quantitative study on how many percentages of roles to be used in a typical adaptive system, we can get a reference from the closest domains. Gregor Kiczales, the inventor of AOP and AspectJ, once mentioned on the Dr.Dobb[8] website that around 15% of program code could be placed in aspects. This number aligns with the finding of Sven Apel who studied on 8 applications written in AspectJ [Ape07, p. 113]. Salvaneschi et al. [SGP11] also believed that this number is reasonable for COP. In this regard, it should rationally hold true for ROP and LyRT. Therefore, a role-based application whose number of roles amounts to 10,000 may contain more than 50,000 classes. At this size, spending 1.5 seconds in average for adaptation of 10,000 roles should be acceptable by the users. Practically, execution times required for anticipated and unanticipated adaptation approximate to 25 and 35 milliseconds respectively for an average-size application whose number of roles amounts to 1,000. Therefore, these adaptation times are relatively small for practical use. Table 6.7 summarizes execution times taken for anticipated and unanticipated adaptation with respect to the number of roles.

[8]The 15% Solution accessed on August 18, 2017: `http://www.drdobbs.com/the-15-solution/184414845`

Table 6.7: An average execution time for adaptation including role binding and unbinding of an application with respect to the 15% solution[1].

	Number of Roles (15%)				
	10	100	1,000	5,000	10,000
Number of Core Objects (85%)	57	567	5,667	28,333	56,667
Anticipated Adaptation[2]	0.29	1.39	25.62	400.45	1,212.60
Unanticipated Adaptation[2]	1.06	2.64	35.35	557.84	1,973.56

[1] The 15% Solution accessed on August 18, 2017: `http://www.drdobbs.com/the-15-solution/184414845`
[2] Execution time in milliseconds.

While scaling on adaptation imposes an insignificant overhead, the method execution of newly adapted core objects inevitably entails a considerable overhead which is subject to optimization. Because of this reason, LyRT is suitable for highly adaptive systems comprising thousands of roles to be adapted at once, but it shall not be used for static or rarely adaptable systems demanding for speed.

At the scale of 10,000 roles, the benchmarks of the checkpoint and the rollback show a significant overhead as opposed to their baselines. However, their execution times are relatively small concerning the practicability of a typical application, i.e., 0.32 seconds and 1.13 seconds for checkpoint and rollback, respectively. Moreover, the rollback overhead affects the overall system performance only when a failure is detected.

Similarly, LyRT uses memory 2.46 times more than the baseline and utilizes CPU even more. However, we still manage the case studies to run on resource-constraint devices such as the first model of Raspberry Pi whose memory is limited to 256MB, and it is powered by a 700MHz single-core ARM processor.

6.3.4 Generality of LyRT

LyRT was proposed to be a general approach as much as possible without relying too much on the host languages. In Section 4.6, we have already described which features are required to implement a prototype for LyRT. Although Java is chosen as the host language, the main purpose is to show the generality of the approach which is not tightly attached to the host language. It is even easier to implement LyRT in dynamically typed languages such as Ruby or Python whose MOP is fully supported for open implementation. Section 5.8 further explains this claim.

6.4 Chapter Summary

In this chapter, we exhibited a systematic approach to evaluate LyRT. First, we demonstrated the prototype of LyRT by using it to implement three case studies in the domain of adaptive software systems. For each case study, we validated LyRT against the run-time variability requirements. These validations show that LyRT is more advanced than the existing variability approaches with respect to adaptation due to the incorporation of unanticipated adaptation and consistency control. Furthermore, LyRT integrates the rollback recovery mechanism in order to improve run-time stability. Subsequently, we compared LyRT with existing role-based approaches based on the classifying features of roles. LyRT satisfies 19 out of 26 features which mainly targets the run-time aspects.

Second, several micro-benchmarks were developed to quantify the overhead of LyRT regarding its supported features. The result shows that LyRT is suitable for highly adaptive systems while falls short on static or rarely adapted systems. That is the intention of developing LyRT in order to trade performance with flexibility, i.e., adaptation, consistency and failure handling in particular. Some points related to implementation have been sketched out for future optimization, i.e., instead of using lambda expressions and Java stream API, an imperative `for-each` loop should be used to manipulate roles. In the next chapter, we will conclude this dissertation by summarizing the contributions and outlining future work.

Chapter 7

Conclusion and Future Work

Adaptive software systems are designed to operate in an ever-changing environment. Traditionally, building such systems leads to convoluted designs in order to achieve the necessary run-time flexibility. Variability is a promising approach to develop adaptive software systems while keeping software components (i.e., variants) modular for reusability. However, variability focuses on variant compositions at build time to create multiple software products. Run-time variability focuses on one product but deals with run-time adaptation by shifting the composition from build to run time. Dynamic composition of variants enables a system to adapt its behavior accordingly.

This dissertation addressed the shortcomings of the existing approaches which are inadequate to provide comprehensive support for run-time variability. They are often limited to anticipated adaptation support in which the system behavior only changes with respect to a set of predefined execution environments. This makes them unsuitable to address practical problems where the execution environment is not fixed and often unknown until run time. Enabling unanticipated adaptation support alleviates this issue but holds several implications yielding undesirable behavior, such as inconsistency and potential run-time failure. Depending on the type of variants and their activation mechanisms, providing support for unanticipated adaptation might be a technological challenge. Throughout this dissertation, we argued that instance-level adaptation is necessary in order to not only support unanticipated adaptation but also handle inconsistency and run-time failure comprehensively.

7.1 Summary of Contributions

The goal of this dissertation was to provide flexible run-time support for adaptation as well as to improve run-time availability and stability. In the context of roles, as discussed in Chapter 2, we presented a run-time architecture, called LyRT, to offer comprehensive support for run-time variability. LyRT is designed to support the coexistence of anticipated and unanticipated adaptation at instance level. Additionally, we solve two fundamental adaptation problems which are inconsistency in ongoing method executions and run-time handling of failures due to role compositions. In order to solve these problems, we presented three mechanisms in Chapter 4, which are seamlessly integrated into LyRT. These three mechanisms are our core contributions which are summarized as follows:

Dynamic Instance Binding Mechanism. The role concept lays a prominent foundation to support instance-level adaptation. Along with CROM [KLG+14], a dynamic instance binding mechanism was proposed to loosely capture the binding relation between core objects and their roles. This binding relation stores enough information not only for

roles to be dispatched but also for their context-dependent behavior to be encapsulated in an activation of a compartment. This relation is stored in a lookup table, a data structure to manipulate roles and their associated compartment. The runtime keeps the three kinds of objects separately allowing to lately adopt new roles at run time. A dynamic method dispatch is determined based on the current information stored in the lookup table. Therefore, this mechanism allows us to achieve both anticipated and unanticipated adaptation at the instance level. The mechanism supports both local and global activation allowing programmers to predefine the adaptation in the currently executing thread or to trigger the adaptation from a global thread. Besides, it allows a core object to bind and execute different roles simultaneously in order to achieve variability in a multi-threaded environment.

Object-Level Tranquility Mechanism. Any system which allows dynamic activation may face inconsistency when objects are not ready to adapt. In other words, those objects promptly change their behavior when they are engaging in a block of code where consistent behavior is required. Based on the concept of tranquility and the dynamic instance binding mechanism, we presented a consistency mechanism working at the object level to refrain the behavior of corresponding objects from being changed regardless of either anticipated or unanticipated adaptation which is triggered asynchronously. In order to enable consistent behavior, programmers have to provide a consistency block, a framework construct, to surround the code demanding consistent behavior. After the block has expired, the runtime reaches a tranquil state in which the adaptation can take place safely. Similar to the dynamic instance binding mechanism, this mechanism operates at the instance and the thread level enabling core objects to live in a consistent state in parallel with the other adapted objects of the same type.

Rollback Recovery Mechanism. Run-time variability requires a dynamic composition of variants. Although practically limited, theoretically, the number of variants is infinite. The dynamic composition of these variants is prone to failure during execution as we cannot assure all the possible compositions to work correctly. We presented a rollback recovery mechanism to remedy this problem. The key to the solution is to install a checkpoint of a system configuration before each adaptation taking place. We proposed a specialized bug sensor, which relies on the application exception handling for the current prototype, to embrace the run-time failure resulting from a defective configuration. The bug sensor signals to the runtime to roll back the system configuration to the most recent checkpoint which worked flawlessly. This mechanism also operates at the thread level. Therefore, the detected failure only affects the encountering thread. This characteristic allows the runtime to perform a lightweight recovery without impacting the other threads.

Each mechanism answers to each of the three research questions formulated in Section 1.2, which were later used to derive six requirements for run-time variability. Rather than answering the questions directly, we validated LyRT against those requirements. The prototype of LyRT was implemented in Chapter 5 and evaluated in Chapter 6. In the evaluation chapter, we implemented three case studies by using the LyRT prototype to validate LyRT against the six requirements. Unlike its rivals discussed in Chapter 3, we argued that LyRT is versatile by satisfying those requirements. This validation proved that the three mechanisms in LyRT meet the objectives, and thus they respond to our research questions. Concerning the run-time performance, we developed several micro-benchmarks to quantify the overhead. The summary of the overhead is shown in Table 6.6. The result

revealed that although there is a considerable overhead compared to the baselines, the execution is still sufficiently small for a practical use of typical adaptive systems. Therefore, LyRT is suitable for a highly adaptive system which requires thousands of objects to be adapted at once, but it should not be used in a static or a rarely adaptive system.

7.2 Future Work

Although LyRT is more versatile in terms of run-time adaptation, consistency and rollback recovery, it inevitably comes with certain drawbacks. In Section 6.3, we discussed the strengths and weaknesses of LyRT. Those weaknesses are the limitations to be addressed in future work.

Optimization. According to the measured overhead, the LyRT prototype needs to be optimized. As mentioned in Section 6.3.2, an immediate action is to eliminate the usage of lambda expressions and to minimize the role-searching process. For a crucial optimization, we need a role-aware JVM in order to get an optimal role tracing when the role becomes active.

Code Readability. The current LyRT prototype is implemented in Java as a library which limits the code readability when a core object needs to execute a role's method. This limitation is because of the strongly and statically typed checking at compile time which does not allow the method to be substituted properly. In turn, we use a string to resemble the method. In order to improve the code readability, an interpreter should be developed to abstract all the function calls in LyRT into pure Java code. We consider to reimplement LyRT in dynamically typed languages as a Domain-Specific Language (DSL) purely embedded in those languages. This technique would not only drastically improve the code readability but would also demonstrate the generality of the approach.

Revisiting the Rollback Recovery Mechanism. In Section 6.3.1.5, we mentioned limitations of the rollback recovery mechanism. As future work, those shortcomings should be addressed, the bug sensor implementation in particular. Furthermore, a more sophisticated case study with a large number of role compositions should be carried out and evaluated along with the feature support for continuous software deployment.

Constraints. As described in Section 2.2, the played roles participating in a compartment may constrain with each other. Enforcing constraints improves run-time consistency. Hence, we plan to integrate these constraints as well as the notion of Role Groups, specified in formal CROM [KBGA15], into LyRT.

Comprehensibility of the Role Model. Inspired by CROM [KBGA15], LyRT introduces a run-time model based on the notion of compartment, role and object model. Therefore, understanding CROM is essential to differentiate between the static and the dynamic parts as well as their associated context so that developers can transform these parts into the respective compartment, role and object model. Kühn et al. [KLG+14] mentioned that the research landscape on roles is fragmented and discontinuous, implying practical limitations. Hence, a user study should be conducted in order to assess the knowledge of applying the role model to a practical application.

Fragility. Enabling unanticipated adaptation may violate system security and integrity. In order to avoid these problems, a contract or a security policy should be specified. Besides, there should be a link between a model and the runtime so that a modification made in the model reflects the runtime change. If the model is checked with a model checker, then the reflective change in runtime is safe to perform with respect to the new specification. ProFeat [CDKB16] is a role-based model checker which can be used to realize this vision.

References

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Lecture Notes in Computer Science*, 3880:135, 2006.

[AHH+09] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, page 6. ACM, 2009.

[AHHM11] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.

[AHM+10] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, pages 50–65. Springer, 2010.

[AK09] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[AKLS07] Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake. Aspect refinement-unifying AOP and stepwise refinement. *Journal of Object Technology*, 6(9):13–33, 2007.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam-A smooth extension of Java with Mixins. In *European Conference on Object-Oriented Programming*, pages 154–178. Springer, 2000.

[Ape07] Sven Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2007.

[BA12] Fernando Sérgio Barbosa and Ademar Aguiar. Modeling and programming with roles: Introducing JavaStage. In *11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT12)*, 2012.

[BA13] Fernando Sérgio Barbosa and Ademar Aguiar. Removing code duplication with roles. In *12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT13)*, pages 37–42. IEEE, 2013.

[Bai12] Engineer Bainomugisha. *Reactive Method Dispatch for Context-Oriented Programs*. PhD thesis, Vrije Universiteit Brussel, 2012.

[BBVDT06] Matteo Baldoni, Guido Boella, and Leendert Van Der Torre. Roles as a coordination construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science*, 150(1):9–29, 2006.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. *ACM Sigplan Notices*, 25(10):303–311, 1990.

[BD77] Charles W Bachman and Manilal Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Databases-Volume 3*, pages 464–476. VLDB Endowment, 1977.

[BD96] Daniel Bardou and Christophe Dony. Split objects: a disciplined use of delegation within objects. In *ACM SIGPLAN Notices*, volume 31, pages 122–137. ACM, 1996.

[BDNG06] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issues and challenges. *Computer*, 39(10):36–43, 2006.

[BGE07] Stephanie Balzer, Thomas R Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *European Conference on Object-Oriented Programming*, pages 323–346. Springer, 2007.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 83–92. ACM, 2004.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30:19, 2002.

[BMN14] Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in Java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 181–186. ACM, 2014.

[Bon04] Jonas Bonér. AspectWerkz-dynamic AOP for Java. In *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[BRSW98] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer, 1998.

[BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[Car13] Nicolás Cardozo. *Identification and management of inconsistencies in dynamically adaptive software systems*. PhD thesis, Université catholique de Louvain and Vrije Universiteit Brussel, 2013.

[CB10] Lianping Chen and Muhammad Ali Babar. Variability management in software product lines: An investigation of contemporary industrial challenges. In *International Conference on Software Product Lines*, pages 166–180. Springer, 2010.

[CC15] Nicolás Cardozo and Siobhán Clarke. Context Slices: Lightweight discovery of behavioral adaptations. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, pages 2:1–2:6. ACM, July 2015.

[CDKB16] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Family-based modeling and analysis for probabilistic systems–featuring ProFeat. In *International Conference on Fundamental Approaches to Software Engineering*, pages 287–304. Springer, 2016.

[CGDM11] Nicolás Cardozo, Sebastian Günther, Theo D'Hondt, and Kim Mens. Feature-oriented programming and context-oriented programming: Comparing paradigm characteristics by example implementations. In *International Conference on Software Engineering Advances. IARIA*, 2011.

[CGMD11] Nicolás Cardozo, Sebastián González, Kim Mens, and Theo D'Hondt. Safer context (de) activation: through the prompt-loyal strategy. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*, page 2. ACM, 2011.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10. ACM, 2005.

[Chi00] Shigeru Chiba. Load-time structural reflection in Java. In *European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[CHOT99] Siobhán Clarke, William Harrison, Harold Ossher, and Peri Tarr. Subject-oriented design: Towards improved alignment of requirements, design, and code. In *ACM SIGPLAN Notices*, volume 34, pages 325–339. ACM, 1999.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Generative Programming And Component Engineering*, pages 364–376. Springer, 2003.

[CO14] Patricia Conde and Francisco Ortin. JINDY: A Java library to support invokedynamic. *Computer Science and Information Systems*, 11(1):47–68, 2014.

[Coo87] Steve Cook. Panel on inheritance. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 35–40, New York, NY, USA, 1987. ACM.

[Dah99] Markus Dahm. Byte code engineering. In *JIT'99: Java-Informations-Tage 1999*, pages 267–277. Springer Berlin Heidelberg, 1999.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Computer Languages, Systems & Structures*, 32(2):125–139, 2006.

[Dey01] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[Dij76] Edsger Wybe Dijkstra. *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs, 1976.

[DMB09] Tom Dinkelaker, Mira Mezini, and Christoph Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, pages 51–62. ACM, 2009.

[DMFM10] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[DMVS89] R Dixon, T McKee, M Vaughan, and Paul Schweizer. A fast method dispatcher for compiled languages with multiple inheritance. In *ACM SIGPLAN Notices*, volume 24, pages 211–214. ACM, 1989.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.

[DSC+99] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *Workshop on Reflection and Software Engineering*, pages 169–188. Springer, 1999.

[DVCH07] Brecht Desmet, Jorge Vallejos, Pascal Costanza, and Robert Hirschfeld. Layered design approach for context-aware systems. In *First International Workshop on Variability Modelling of Software-intensive Systems*, pages 157–165, 2007.

[ESMJ10] Peter Ebraert, Hans Schippers, Tim Molderez, and Dirk Janssens. Safely updating running software. In *ECOOP'10 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2010.

[Fel90] Matthias Felleisen. On the expressive power of programming languages. *ESOP'90*, pages 134–151, 1990.

[FFN00] Robert E Filman, Daniel P Friedman, and Peter Norvig. Aspect-oriented programming is quantification and obliviousness. 2000.

[Fow97] Martin Fowler. Dealing with roles. In *Proceedings of PLoP*, volume 97, 1997.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[GCM+10] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c. In *International Conference on Software Language Engineering*, pages 246–265. Springer, 2010.

[GCX+14] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. Low-disruptive dynamic updating of Java applications. *Information and Software Technology*, 56(9):1086–1098, 2014.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[GJ11] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Run-time phenomena in dynamic software updating: causes and effects. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 6–15. ACM, 2011.

[GMC08] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object system. *J. UCS*, 14(20):3307–3332, 2008.

[GMCC13] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. Context traits: Dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 209–220. ACM, 2013.

[GØ03] Kasper B Graversen and Kasper Østerbye. Implementation of a role language for object-specific dynamic separation of concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[Gon08] Sebastián González. *Programming in Ambience, gearing up for dynamic adaptation to context*. PhD thesis, Université catholique de Louvain, 2008.

[GP09] Sebastian Götz and Mario Pukall. On performance of delegation in Java. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, page 3. ACM, 2009.

[Gri00] Martin L Griss. Implementing product-line features by composing aspects. In *Software Product Line Conference*, pages 271–288. Springer, 2000.

[GS12] Sebastian Günther and Sagar Sunkle. rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming*, 77(3):152–173, 2012.

[GWT+14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. Variability in software systems—a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014.

[HCH08] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[Her05] Stephan Herrmann. Programming with roles in ObjectTeams/Java. In *Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI) Fall Symposium*, 2005.

[Her10] Stephan Herrmann. Demystifying object schizophrenia. In *Proceedings of the 4th Workshop on Mechanisms for Specialization, Generalization and InherItance*, page 2. ACM, 2010.

[HHM04] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation polymorphism in object teams. Technical report, Technical Report 2004/05, Technical University Berlin, 2004.

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[Hil10] Rich Hilliard. On representing variation. In *ECSA Companion Volume*, pages 312–315, 2010.

[HNL+06] Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java extension with dynamic object roles. In *13th IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 7–pp. IEEE, 2006.

[HO93] William Harrison and Harold Ossher. *Subject-oriented programming: A critique of pure objects*, volume 28. ACM, 1993.

[HOM95] William Harrison, Harold Ossher, and Hafedh Mili. Subjectivity in object-oriented systems: workshop summary. In *ACM SIGPLAN OOPS Messenger*, volume 6, pages 117–122. ACM, 1995.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[HPS12] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012.

[HVBL15] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. CoDEL–a relationally complete language for database evolution. In *East European Conference on Advances in Databases and Information Systems*, pages 63–76. Springer, 2015.

[HWS+09] Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Proceedings of the Third International Workshop on Dynamic Software Product Lines*, pages 18–27. Carnegie Mellon University, 2009.

[INPJ09] Paul Istoan, Gregory Nain, Gilles Perrouin, and Jean-Marc Jezequel. Dynamic software product lines for service-based systems. In *Ninth IEEE International Conference on Computer and Information Technology*, volume 2, pages 193–198. IEEE, 2009.

[JKH+15] Tobias Jäkel, Thomas Kühn, Stefan Hinkel, Hannes Voigt, and Wolfgang Lehner. Relationships for dynamic data types in RSQL. In *Proceedings of the 16th Conference on Database Systems for Business, Technology, and Web (BTW)*, pages 157–176, 2015.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, pages 672–681. IEEE, 2013.

[JZ10] Arkadiusz Janik and Krzysztof Zielinski. AAOP-based dynamically reconfigurable monitoring system. *Information and Software Technology*, 52(4):380–396, 2010.

[KAB07] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 223–232. IEEE, 2007.

[KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development*, pages 253–264. ACM, 2011.

[KAM15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Proceedings of the 14th International Conference on Modularity*, pages 14–28. ACM, 2015.

[KBGA15] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. A combined formal model for relational context-dependent roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–124. ACM, 2015.

[KBRA16] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. FRaMED: Full-fledge role modeling editor (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 132–136. ACM, 2016.

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In *International Conference on Software Language Engineering*, pages 141–160. Springer, 2014.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97: Object-oriented programming*, pages 220–242. Springer, 1997.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[KM96] Bent Bruun Kristensen and Daniel CM May. Activities: Abstractions for collective behavior. In *European Conference on Object-Oriented Programming*, pages 472–501. Springer, 1996.

[KØ96] Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[Kri96] Bent Bruun Kristensen. Object-oriented modeling with roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer, 1996.

[Kri98] Bent Bruun Kristensen. Subject composition by roles. In *Proceedings of the International Conference on Object Oriented Information Systems*, pages 181–196. Springer, 1998.

[KS04] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, volume 7, 2004.

[KT09] Tetsuo Kamina and Tetsuo Tamai. Towards safe and flexible object adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM, 2009.

[LA15] Max Leuthäuser and Uwe Aßmann. Enabling view-based programming with SCROLL: Using roles and dynamic dispatch for establishing view-based programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 25–33. ACM, 2015.

[LASH11] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76, 2011.

[Leu17] Max Leuthäuser. *A Pure Embedding of Roles*. PhD thesis, Technische Universität Dresden, 2017.

[LK06] Jaejoon Lee and Kyo Chul Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Software Product Line Conference, 2006 10th International*, pages 10–pp. Ieee, 2006.

[LKKP06] Kwanwoo Lee, Kyo Chul Kang, Minseong Kim, and Sooyong Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *10th International Software Product Line Conference (SPLC'06)*, pages 10–pp. IEEE, 2006.

[LSSP06] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In *Transactions on Aspect-Oriented Software Development II*, pages 227–255. Springer, 2006.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.

[MCD16] Kim Mens, Nicolás Cardozo, and Benoît Duhoux. A context-oriented software architecture. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 7–12. ACM, 2016.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 127–136. ACM, 2004.

[Moo86] David A. Moon. Object-oriented programming with Flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[MS04] Virendra J Marathe and Michael L Scott. A qualitative survey of modern software transactional memory systems. *University of Rochester, Computer Science Department, Technical Report*, 2004.

[NAR08] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running Java programs. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 233–246. ACM, 2008.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[ODPR08] Audrey Occello, A Dery-Pinna, and Michel Riveill. A runtime model for monitoring software adaptation safety and its concretisation as a service. *Models@ runtime*, 8:67–76, 2008.

[OKH+95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. *ACM SIGPLAN Notices*, 30(10):235–250, 1995.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PDFS01] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. JAC: A flexible solution for aspect-oriented programming in Java. In *International Conference on Metalevel Architectures and Reflection*, pages 1–24. Springer, 2001.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 235–248. ACM, 2005.

[RAB+92] Trygve Reenskaug, Egil P Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Nêss E Ulseth, Gro Oftedal, AL Skaae, et al. ORASS: Seamless support for the creation and maintenance of object oriented systems. *Journal of Object Oriented Programming*, 1992.

[RAF04] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. A comparison of bug finding tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *European Conference on Object-Oriented Programming*, pages 205–230. Springer, 2002.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Computer Languages, Systems & Structures*, 34(2):46–65, 2008.

[RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *ACM SIGPLAN Notices*, volume 33, pages 117–133. ACM, 1998.

[RSAS11] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011.

[RSPA11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *ACM SIGPLAN Notices*, volume 47, pages 3–12. ACM, 2011.

[SAM13] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[SC92] Henry Spencer and Geoff Collyer. # ifdef considered harmful, or portability experience with C news. 1992.

[SC17] Lars Schütze and Jeronimo Castrillon. Analyzing state-of-the-art role-based programming languages. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*. ACM, 2017.

[Sch08] Gregor Schmidt. Contextr and contextwiki. Master's thesis, Hasso Plattner Institut, Postdam, Germany, 2008.

[SCT03] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *Generative Programming and Component Engineering*, pages 189–208. Springer, 2003.

[SD05] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In *European Conference on Object-Oriented Programming*, pages 453–478. Springer, 2005.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[SG99] Peter F Sweeney and Joseph Yossi Gil. Space and time-efficient memory layout for multiple inheritance. *ACM SIGPLAN Notices*, 34(10):256–275, 1999.

[SGP11] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. JavaCtx: Seamless toolchain integration for context-oriented programming. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*, page 4. ACM, 2011.

[SGP12a] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[SGP12b] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: Introducing context-oriented programming in the actor model. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 191–202. ACM, 2012.

[SGP13] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An analysis of language-level support for self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(2):7, 2013.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 1–12. ACM, 2009.

[SLU88] Lynn Andrea Stein, Henry Lieberman, and David Ungar. *A shared view of sharing: the treaty of Orlando*. Brown University, Department of Computer Science, 1988.

[SMCS04] S Masoud Sadjadi, Philip K McKinley, Betty HC Cheng, and RE Kurt Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, pages 1243–1261. Springer, 2004.

[SP08] Christian Schubert and Michael Perscheid. Dynamic contract layers for python. Master's thesis, Hasso Plattner Institut, Postdam, Germany, 2008.

[SR02] K Chandra Sekharaiah and D Janaki Ram. Object schizophrenia problem in object role system design. In *International Conference on Object-Oriented Information Systems*, pages 494–506. Springer, 2002.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):14, 2009.

[Ste00] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[Ste07] Friedrich Steimann. The role data model revisited. *Applied Ontology*, 2(2):89–103, 2007.

[Str93] Robert Stroud. Transparency and reflection in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(2):99–103, 1993.

[SU96] Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems (TAPOS)*, 2(3):161–178, 1996.

[SW95] Robert J Stroud and Zhixue Wu. Using metaobject protocols to implement atomic data types. In *European Conference on Object-Oriented Programming*, pages 168–189. Springer, 1995.

[TBSN01] Éric Tanter, Noury MN Bouraqadi-Saâdani, and Jacques Noyé. Reflex—towards an open reflective extension of Java. In *International Conference on Metalevel Architectures and Reflection*, pages 25–43. Springer, 2001.

[TCPB07] Pablo Trinidad, Antonio Ruiz Cortés, Joaquín Peña, and David Benavides. Mapping feature models onto component models to build dynamic software product lines. In *SPLC (2)*, pages 51–56, 2007.

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. *ACM SIGPLAN Notices*, 38(11):27–46, 2003.

[TOHSJ99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[TSCS17] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. A rollback mechanism to recover from software failures in role-based adaptive software systems. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 11. ACM, 2017.

[TUI05] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *Proceedings of the 27th International Conference on Software Engineering*, pages 166–175. ACM, 2005.

[TWS+16] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. Consistent unanticipated adaptation for context-dependent applications. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 33–38. ACM, 2016.

[UOK14] David Ungar, Harold Ossher, and Doug Kimelman. Korz: Simple, symmetric, subjective, context-oriented programming. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 113–131. ACM, 2014.

[VBAM09] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. HotWave: Creating adaptive tools with dynamic aspect-oriented programming in Java. In *ACM Sigplan Notices*, volume 45, pages 95–98. ACM, 2009.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[VN96] Michael VanHilst and David Notkin. Using role components in implement collaboration-based designs. *ACM SIGPLAN Notices*, 31(10):359–369, 1996.

[VSV+05] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in JAsCo. In *Proceedings of the*

4th International Conference on Aspect-oriented Software Development, pages 75–86. ACM, 2005.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 10–19. ACM, 2010.