A Dynamic Instance Binding Mechanism Supporting Run-Time Variability of Role-Based Software Systems

Nguonly Taing[1] Thomas Springer[1] Nicolás Cardozo[2] Alexander Schill[1]

[1]Faculty of Computer Science, Technische Universität Dresden, Germany
[2]Future Cities, DSG, Trinity College Dublin, Ireland
{firstname.lastname}@tu-dresden.de, cardozon@scss.tcd.ie

Abstract

Role-based approaches gain more and more interest for modeling and implementing variable software systems. Role models clearly separate static behavior represented by *players* and dynamic behavior modeled as *roles* which can be dynamically bound and unbound to players at run time. To support the execution of role-based systems, a dynamic binding mechanism is required. Especially, since instances of the same player type can play different roles in a single context, the binding mechanism is required to operate at instance level. In this paper, we introduce a mechanism called *dynamic instance binding* for implementing a runtime for role-based systems. It maintains a look-up table that allows the run-time system to determine and invoke the currently active role binding at instance level. We explain dynamic instance binding mechanism in detail and demonstrate that it is flexible enough to support both adaptation and evolution of software systems at run time.

Categories and Subject Descriptors D.2.11 [*Software Architectures*]: Framework

Keywords Dynamic Binding, Roles, Adaptive Systems

1. Introduction

Variability is one of the major requirements for software systems executed in highly heterogeneous and dynamically changing environments, as it is the case for ubiquitous computing or cyber-physical systems, among others.

The degree of variability of a system is determined by the number of contexts it can differentiate, its ability to perform foreseen or unforeseen changes, the granularity of change operations, and the point in time these changes can be performed.

This paper focuses on a dynamic binding mechanism to enable program variations at run time. Dynamic binding supports the selection and invocation of a particular behavior relevant for the current context. Object binding is normally restricted to foreseen changes where the set of variants of the binding mechanism is fixed, and the number of considered contexts is static [9]. However, this is rarely the case in highly dynamic environments. Thus, unforeseen changes can be supported only if it is possible to extend the set of contexts and behavior implementations at run time. Moreover, the granularity of change operations indicates whether the binding mechanism operates at type or instance level. Since two instances of a certain type can behave differently if they execute in two particular executing contexts, we posit that an instance level binding mechanism is suitable to support such cases.

Dynamic binding mechanisms are used to support run-time variability in several domains. Concepts based on Aspect-oriented Programming (AOP) allow to weave crosscutting concerns into object's code, uniformly adapting all instances of a given type, limiting the flexibility for rebinding variations, and not supporting unforeseen changes [11, 17]. Context-oriented Programming (COP) addresses the support of behavioral adaptation based on context-dependent layer activation. While some COP approaches allow to integrate new behavior or operate at instance level, (un)foreseen instance adaptations are not supported in a single language [18].

Role-oriented Programming (ROP) gains increasing interest as a solution to support variability [13]. Roles encapsulate dynamic behavior and can be bound to players containing static behavior dynamically at run time in relation to a particular context. However, existing ROP solutions bind roles at compile time, imposing a challenge to bind other unforeseen roles at run time.

In this paper, we address the problem of run-time variability for role-based software systems at the instance level, allowing both anticipated and unanticipated adaptations coexist in a single solution. We introduce a mechanism called *dynamic instance binding* that maintains a data structure representing the binding information between player and role instances in a look-up table. That table is used to dynamically invoke the behavior of the role to which a player is currently bound. The mechanism is implemented as part of the runtime, called *LyRT*, supporting the execution of role-based software systems. Based on that implementation, we demonstrate that dynamic instance binding can support flexible (re-)binding of roles and object instances as well as the introduction of new role implementations without the need to restart the system. The proposed mechanism is a generic approach for any OOP language that supports reflection. This mechanism can even be applied to statically type languages, enabling the evolution of legacy systems. The LyRT prototype is a Java-based implementation to demonstrate the challenge of live adaption in the statically typed languages.

The paper is organized as follows: Section 2 introduces the concept of roles. Section 3 presents the proposed dynamic instance binding mechanism, and Section 4 describes the implementation details of the mechanism as part of our role-based runtime. In Section 5 we demonstrate that dynamic instance binding supports unanticipated adaptation based on a case study. Section 6 compares our work to related approaches, and Section 7 concludes and presents our future work.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892687

2. Roles

The concept of roles has already been used in several domains like access-control and database systems. The main idea of roles is reflected in the Role Object Pattern [2]. It proposes to model context-specific views of an object as a set of separate *roles* that can be dynamically bound to and unbound from the resulting *core*. While the core contains attributes and behaviors maintained over its complete life-time, roles also contain attributes and behaviors only presented in a particular context.

For example, a core object *person* has a name and social security number that never change throughout the lifetime of a person. In addition, the person can become a developer or an employee. Such transient parts are made available dynamically (i.e., *activated*), whenever they are needed in a concrete execution context, for example in a company and its tax department.

A set of 26 features is introduced that can be used to characterize role models [13]. In the following, we describe the set of features valid for the role model we use in our work. The main elements of our role model are *roles* which are played by a *core object*. Roles are self-contained objects encapsulating their state and behavior. They require to be bound to a player based on the *play* relationship. A player may play several roles simultaneously and may acquire or abandon roles dynamically.

Figure 1 introduces a payroll management system as an example role model. In the example the class `Person` has different instances (e.g., `ely, bob, alice`), each playing different roles. While `ely` and `bob` play the `Developer` role, `alice` plays the `Accountant` role.

The example adds the notion of a *compartment* [13] to the main elements. A compartment defines a particular scope in which a set of roles is valid and can interact with each other. It contains state and behavior and may itself play roles. The figure includes the compartments `TaxDepartment` and `Company`, in which the Company compartment plays the role of a `TaxPlayer`.

The *play* relationship is not restricted to core objects and roles but a role is also allowed to play another role. In our example (Figure 1), a person `ely` plays the role `Freelancer` that plays the role of `TaxPayer` in the context of the `TaxDepartment`.

Roles can interact with each other via method calls, affecting all core players playing the role. For instance, the `Accountant` role calls the `paySalary()` method to pay monthly salary to all objects playing a `Developer` role. 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 `Company` compartment instance, to distribute it to the `Developer` instances. The `Company` compartment may play a role (`TaxPayer`) inside other compartments (`TaxDepartment`). Dynamic adaptation takes place by unbinding roles of core instances, and binding them to some other roles. For example, the `ely` instance can be unbound from the `Developer` role and bound to the `FreeLance` role at run time.

The example in Figure 1 serves as the showcase that we will use in the rest of the paper to illustrate our mechanism for dynamic instance binding and the implementation of the LyRT runtime.

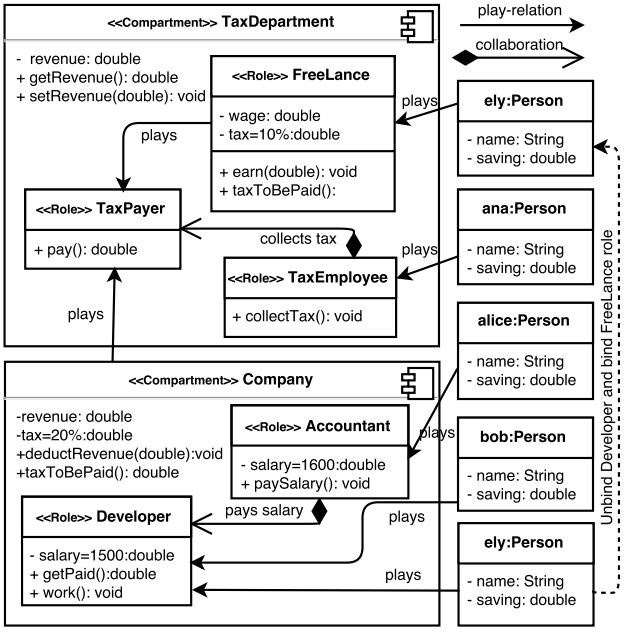


Figure 1: Payroll management system run-time model.

3. LyRT Framework for Run-time Adaptations

This section presents the main mechanisms used in LyRT to enable a flexible run-time environment supporting continuous adaptations even if these have not been foreseen at design time.

3.1 Dynamic Instance Binding

Code weaving is a mechanism to bind the implementation of two objects [11]. This mechanism is used in different programming paradigms geared towards dynamic object (re)composition, such as Aspect-oriented Programming (AOP) [17], Context-oriented Programming (COP) [9], and Role-oriented programming (ROP) [8]. Weaving allows to integrate a behavioral entity (e.g., aspect, layer) implementation with that of a base class. *Object binding* is a higher level abstraction used to express the combination of object models regardless of their implementation (e.g., role binding).

Normally, objects are woven either at 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, however, this may result in a performance overhead. System crashes may occur when reweaving as this mechanism does not assure the validity of 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, copying its state in order to map it back after the newly woven instance has been instantiated. We classify the aforementioned weaving mechanisms as *tight weaving* given that, once woven, the objects become a single program entity.

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 relationship shows the run-time association between binding instances. For example, in our payroll management system, a `Person`

instance binds to a `Developer` instance by saying that the `Person` *plays* a `Developer` role. Note that bound instances remain completely decoupled from each other while appearing as a single object. Programmers interact with a single object instances, however, the two instances are two distinguishable objects throughout their lifespan. A look-up table is used to store the binding relationship at run time. Once bound, two instances are virtually woven. In our example, the `Person` instance has access to all state and behavior of the `Developer`. In order to do this, a proper method dispatching mechanism is needed (c.f. Section 3.3). In our system, unbinding object instances boils down to removing the association from the look-up table and destroying unused instances, for example, removing the *play* association 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.

To deal with unanticipated behavior at run time, the new behavioral classes must be defined and compiled. Whenever the bytecode classes are loaded to the run-time environment, they immediately become available to be bound to other existing instances.

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 adaptation and evolution at instance level.

3.2 Dynamic Role Binding

As mentioned earlier, roles are defined as regular OO classes, easing their binding to object instances that are to play them. Our dynamic role instance binding differentiates the type of instances and their binding relations conforming to the role playing model. Based on the role features described in Section 2, instances are categorized as *players*, *roles*, and *compartments*; these instances are called *actors*. As mentioned before, in order to bind two instances we define the relation between them. The possible binding relations we adopt in LyRT are: *player-plays-role* and *role-plays-role*. *Compartment* may also play role but it is a *player-plays-role* relation. There are more binding relations such as *role-inheritance* and *role-constraints*. However those relations are left out of the LyRT scope at the moment to ease our discussion, but can be easily adopted by our instance binding concept.

Actor types are normal OO classes that do not necessarily relate to each other at the design time. At run time, they can be virtually woven adhering to the role playing model. The relationship look-up table is extended to capture the different types of instance bindings. The structure of the look-up table is expressed in the form of a relational database schema as shown in Figure 2. Each actor instance identity of each type is stored in the respective table of `PlayerInstance`, `RoleInstance`, and `CompartmentInstance` according to their type. The `Relation` table holds the relationship of those binding instances, used later by the method dispatch mechanism in order to use the appropriate instances when invoking a specific behavior.

Table 1 shows the sample data of the run-time model depicted in Figure 1. Instead of using an integer for the instance identity, we use plain text easing the identity tracking. The first row in the table represents the type of a player-plays-role (PPR) relation, between the `bob` and `developer` instances inside a `company` compartment instance. Whereas, `bob` plays a different role, as shown in Row 2. Rows 3 and 4 demonstrate the role-plays-role (RPR) relation that `ely` has the `freelance` role and this role has the `taxpayer` one. These two relations virtually weave the `freelance` and `taxpayer` roles to the `ely` instance. A compartment is just like any player that may play a role. For example, the `company` compartment

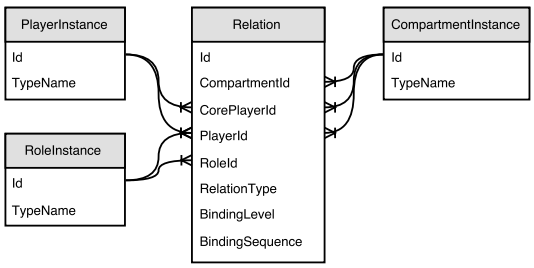


Figure 2: Relationship Look-up Table

plays the `taxpayer` role inside the `tax` compartment under a PPR relation in Row 5. The *binding level* (`Lvl` in Table 1.) represents the depth from a core object instance that occurs in a role-plays-role relation. The *binding sequence* (`Seq` in Table 1.), represents the order of binding at each *binding level*. These two attributes are used by the method dispatch mechanism to find the correct behavior implementation of all bound instances.

Table 1: Sample Data in the Relation table

Id	Com.Id	CoreId	PlayerId	RoleId	Type	Lvl	Seq
1	company	bob	bob	developer	PPR	1	1
2	company	bob	bob	accountant	PPR	1	2
3	company	ely	ely	freelance	PPR	1	1
4	company	ely	freelance	taxpayer	RPR	2	1
5	tax	company	company	taxpayer	PPR	1	1

3.3 Dynamic Method Dispatching

In the role playing relation, a player 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 looks for the method to call among all bound roles in the `Relation` table, and invokes it. If the look-up 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 that have no duplicated method signatures (i.e., no method polymorphism) between player and role instances.

In case of method polymorphism, the method invocation is always resolved firstly for the role implementation. The priority of invocation is given to a role through a traversing process starting from the *latest binding* of the *closest relation* with respect to the player, cascading recursively to the *latest binding* of the *farthest relation*. Finally, this process returns the role instance that has no next binding level. To follow this process we use the `RelationType`, `BindingLevel` and `BindingSequence` attributes of the `Relation` table, finding the proper role instances to the invoked method. The process to find the proper role instance is showcased in Snippet 1.

Snippet 1: Method dispatching for multiple role bindings

```
Object findRole(Object obj){
 if(obj.hasNextBindingLevel()){
  Object[] roles=obj.nextBindingLevelRoles();
  Object role=findMaxBindingSequence(roles);
  return findRole(role); //recursively call
 }
 else { return obj; }
}
```

As an example, Figure 3 presents a core object `O` and the roles it plays in the same compartment, where both roles implement the same `m1()` method. When invoking `m1()`, based on our method dispatching algorithm, the `m1()` implementation in `R2` will be invoked (Figure 3a). This is because in the same binding level (level=1),

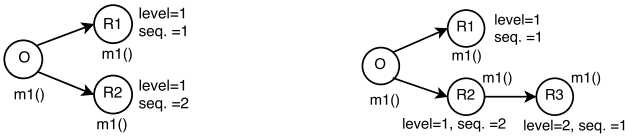


Figure 3: Method Dispatch

R2 has the highest binding sequence number. In case of Figure 3b, the dispatcher initially returns R2 as the responding role, however this still has the next binding relation (level=2). Thus the dispatcher recursively cascades to the next level and returns R3 as the proper role instance for invocation.

4. LyRT Implementation

This section discusses how to implement the dynamic instance binding mechanism for role-based systems. LyRT[1] is a generic framework that can be implemented in any OOP language with support for reflection. Reflection is used for method dispatching. The code snippets shown below are written using the standard Java syntax.

As already explained in the previous section, our dynamic role binding deals with instances. Thus at the type level, players, roles, and compartments are decoupled from each other and are virtually woven together at run time by means of relations formation.

LyRT consists of three main parts: a *registry*, an *instance pool*, and a *look-up table*. The registry works as a mediator to manage all instances and their instance relations. The method dispatching algorithm is also implemented in the registry. All types of instances are stored in the instance pool while relations are captured in the look-up table. The instance pool stores each type of all instances that map to the `PlayerInstance`, `RoleInstance` and `CompartmentInstance` table, respectively. The instance pool is used to get the references of the actual instances for method invocation.

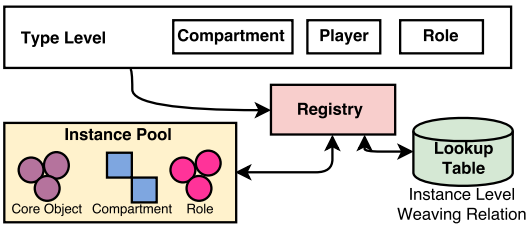


Figure 4: An overview of role run-time framework

4.1 Binding and Unbinding Roles: Flexibility and Adaptability

Section 1 discusses the flexibility and adaptability properties of a dynamic runtime. This section demonstrates how our framework satisfies these properties by means of role binding and unbinding. In the payroll management example, type declarations are regular OOP class definitions as shown in Snippet 2. Snippet 3 shows the main program of the payroll management example, exhibiting flexibility and adaptability. Line 2 gets the reference to the registry, LyRT's core. A `Company` compartment and the `ely` instance of a `Person` type are initialized in Lines 3-4. Line 5 is the binding for the role-playing relation of `ely` as a `Developer` role. In this case, `Developer` is instantiated and loosely woven to `ely` —that is, the objects are not fused together. Once a player and a role are woven, the state and behavior of the role are accessible by the player; only then the player can invoke a role method (Line 6). The *flexibility* of our framework is shown when `alice` instance of type `Person` plays the `Accountant` role (Lines 9-10). The behaviors of these two instances differ according to the roles they play. The framework's *adaptability* is shown when the `ely` instance abandons its current role to bind the `FreeLance` role in the `TaxDepartment` compartment (Lines 13-18).

[1] https://bitbucket.org/RoSIDA/local-runtime

Snippet 2: Type Declaration

```
1 class Person{ //Player
2  private String name; //get/set
3  private String saving; //get/set
4 }
5 class Developer{ //Role
6  private double salary; //get/set
7  public void work(){...}
8 }
9 class Company{ //Compartment
10  private double revenue; //get/set
11  private double tax=0.2; //20%
12  public double taxToBePaid(){...}
13 }
```

Snippet 3: Main Program

```
1 public static void main(String[] args){
2  Registry reg = Registry.getInstance();
3  Object abc = reg.newCompartment(Company.class);
4  Object ely = reg.newPlayer(Person.class);
5  reg.bind(abc, ely, Developer.class);
6  reg.invokeRole(abc, ely, "work"); //ely.work();
7
8  Object alice=reg.newPlayer(Person.class);
9  reg.bind(abc, alice, Accountant.class);
10  reg.invokeRole(abc, alice, "paySalary");
11
12  //ely unbinds developer from abc compartment
13  reg.unbind(abc, ely, Developer.class);
14
15  //ely binds FreeLance in another compartment
16  Object td=reg.newCompartment(TaxDepartment.class);
17  Object fl=reg.bind(td, ely, FreeLance.class);
18  reg.invokeRole(td, ely, "setRevenue", 2500);
19
20  //role plays role. Bind FreeLance to TaxPayer
21  reg.bind(td, fl, TaxPayer.class);
22  //ely now can call pay() method
23  reg.invokeRole(td, ely, "pay"); //ely.pay();
24 }
```

4.2 Roles Play Roles: Object Transformation

In order to further enhance the flexibility and adaptability of the system, roles can play roles. This is done by transforming the core object beyond its own capacities. Figure 1 shows how the `ely` instance becomes a `TaxPayer` after being bound to the `FreeLance` role, because `FreeLance` is bound to the `TaxPayer` role. Note that without the `FreeLance` binding, `ely` would not have the `taxToBePaid()` method, required for a `TaxPayer` (Line 21-23 of Snippet 3). According to our method dispatching, if `ely` is bound to the two aforementioned roles at the same binding level, then `ely` cannot invoke the `pay()` method because the two roles have no relationship to each other. Hence, `ely` does not have the required `taxToBePaid()` for the `TaxPayer` role.

4.3 Role Evolution: Dealing with Unanticipated Behavior

The adaptations discussed so far are foreseen at design time. This section describes how to manage unforeseen adaptations by introducing new roles to the system, without having to restart it. As

seen in Snippet 2, roles are declared as classes. These classes can be loaded at run time using Java's *ClassLoader*. Once roles are loaded, they can be bound to core objects at run time. To achieve this, new behavior must be defined in a role class, later compiled. Then the system evolves as part of an independent daemon thread that loads role instances to the runtime. Loading role instances fires an event that returns the instance's identity from the instance pool (Figure 4). At this point, the role can be bound enabling the new role behavior to the binding instance (Snippet 4). Finally, when the binding operation is completed, method dispatcher delegates to the behavior introduced by the new role.

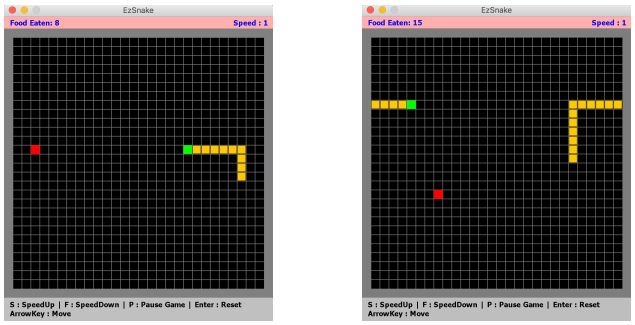
This mechanism is more beneficial than directly modifying object's source code, which is prohibited in Java, as the evolved behavior is modular and reversible. Using this approach new methods or bug fixes are defined in a new role class, taking into account that the signature of the new methods have to match that of the existing ones.

Snippet 4: Evolution for Unexpected Behaviors

```
1 public void evolutionPerformed(Event e){
2   Class roleCls = Class.forName("runtime.NewRole");
3   Registry reg=Registry.getInstance();
4   //Query from instance pool via registry
5   Object[] elys=reg.query(abc, Person.class);
6   //bind instance to role type in abc compartment
7   reg.bind(abc, elys[0], roleCls);
8 }
```

5. Managing Unforeseen Adaptations in LyRT

This section demonstrates how our framework deals with unforeseen adaptations. To do this, we use a real world example of a snake game. Figure 5a shows the default game settings, where the snake crashes whenever it hits the walls. Figure 5b shows the (unforeseen adaptation) behavior of the snake passing through walls. This adaptation takes place by defining, loading, and binding new role during the game play.



(a) Default Settings (b) Passing Wall

Figure 5: Snake Game User Interface

In order to evolve the system and adapt to unforeseen functionality, new behavior is encapsulated in a role, having the same method signature as a core object. Once the core object plays the new role, the invocation priority is given to it. In the snake game the `getNextCell()` method is part of a `Router` object. This method gets the next cell where the snake is going to move, based on its current direction. In the normal game play, if the next moving cell is a wall or the snake body then the snake crashes. To present the variability of the game settings, we want to enable the snake to pass through walls. This modification should happen without restarting the game. Rather than directly modifying the code of the `Router` object, a `PassingWall` role is defined and loaded at runtime (Snippet 5). The `getNextCell()` method in the `PassingWall` role has the same signature in the core object (`Router`). The main difference between the core method and the role method is the passing wall logic, added in Lines 12, 15, 18 and 21. As soon as the binding operation is completed, the call to `getNextCell()` in the `Router` object is resolved by the method defined in the `PassingWall` role. At this moment, the snake can pass through walls lively in the game.

Snippet 5: PassingWall Role

```
1  public class PassingWall {
2   public Cell getNextCell(Cell currentPosition) {
3    Router router = (Router)Registry.getInstance().
         getCorePlayer(this); //get player instance
4    Board board = router.getBoard();
5    int row = currentPosition.getRow();
6    int col = currentPosition.getCol();
7    int dir = router.getDirection();
8
9    if (dir == Router.DIRECTION_RIGHT) {
10     col++;
11     if(col==board.getColCount()-1) col=1;
12   } else if (dir == Router.DIRECTION_LEFT) {
13     col--;
14     if(col == 0) col = board.getColCount() - 2;
15   } else if (dir == Router.DIRECTION_UP) {
16     row--;
17     if(row == 0) row = board.getRowCount() - 2;
18   } else if (dir == Router.DIRECTION_DOWN) {
19     row++;
20     if(row == board.getRowCount()-1) row = 1;
21   }
22   return router.getBoard().getCells()[row][col];
23  }
24 }
```

6. Related Work

LyRT offers dynamic binding of roles and object instances at run time. This section compares our approach with other mechanisms for dynamic object binding, such as AOP, COP, and ROP.

AOP enables run-time adaptation by injecting crosscutting concerns, explicitly into a class by means of join points and pointcuts defined in an aspect module [11, 12]. Adaptations are injected by weaving the aspect code into a class before the program is executed [11]. Dynamic aspect weaving is introduced [3, 4, 16, 19–21] to apply aspects at run time to enable adaptations. In this approach, whenever an aspect is woven with an object, they are bound together at bytecode level —that is, they become one new object. The main advantage of using AOP approaches is the performance of method calls. However, AOP approaches operate at type level adapting all object instances with the same state and behavior and thus, hampering the flexibility of the system. While there is an approach enabling instance level weaving [17], this fuses together object's and aspect's code, making it difficult to retract adaptations.

COP is a programming technique for the introduction of behavioral adaptations [9]. Behavioral adaptations are grouped in layers that can be (de)activated at run time. Whenever a layer is activated, its behavioral adaptations are composed with the base level object, generating a new object. However, behavioral adaptations and objects are not fused together, so it is possible to retract adaptations, by means of layer deactivation. Finally, adaptation of specific object instances is achieved by scoping layer (de)activation locally [10] using dynamic dispatching techniques similar to the ones explained in Section 3.3. Note that most COP languages [18] require the upfront definition of behavioral adaptations (e.g., layers). Nonetheless, in some COP languages implemented in dynamically typed languages, it is possible to include unexpected behavior at run time. This, however, requires the extension of the solution exploiting the underlying language's features, as it is the case

in JavaScript implementations [5]. Therefore, COP languages enable the main features offered by our framework that is dynamic instance binding and unanticipated evolution. However, bringing both adaptations at the instance level while the system is running is not directly supported in current COP implementations. Our framework uses a novel technique to incorporate (un)foreseen adaptations to object instances in statically typed languages that support reflection.

ROP is the programming paradigm arising from the role-object design pattern, such that roles are not tangled with base objects. While current implementations of ROP [13, 14] support dynamic adaptation through role-object bindings, none of them addresses unanticipated adaptation. ObjectTeams/Java [8] uses bytecode libraries (e.g., BCEL[2], ASM[3]) to weave role-specific behavior to an object at either compile or load time and enables adaption at run time by means of teams activation. However, (un)bindings roles at run time are not feasible because bindings are static. Rava [7], EpsilonJ [15], Chameleon [6], and powerJava [1] need a pre-compiler to translate to standard Java, making their bindings static; and therefore (un)binding operations are not supported at run time. SCROLL [14] uses Scala's language features for method rewriting at compile time, and compounds dynamic types for implicit conversion; so it is also considered as a static mechanism. Our framework also uses a ROP approach to achieve adaptivity. In contrast to current ROP implementations, it supports run-time role (un)binding at instance level as it does not fuse together objects and roles.

7. Conclusion and Future Work

Supporting run-time variability is still a challenging task in software engineering. To address this challenge, we proposed the runtime LyRT that supports run-time variability of role-based software systems. We used a role model for system development since it provides a clear separation of static and dynamic system behavior in combination with the ability to bind and unbind dynamic behavior dependent on a particular context. A central component in our runtime is the dynamic instance binding mechanism that supports the binding and unbinding of roles at instance level. As we demonstrated, the runtime is capable to dynamically (re-)bind role and object instances as well as to incorporate new behavior that is dynamically added to the system.

As a next step we will conduct a series of experiments to explore the performance overhead introduced by dynamic instance binding. As a second aspect, we will examine the robustness of LyRT for the incorporation of new behavior by implementing a set of example systems. In the future, we will also work on a language syntax for Role-oriented Programming with the objective of easing the further implementation of systems using ROP, and the ambition to assure safety properties of the run-time adaptation.

Acknowledgments

This work is partially funded by Erasmus Mundus Program, the German Research Foundation (DFG) through the Research Training Group on Role-based Software Infrastructures for Continuous-Context-Sensitive Systems (RoSI) (GRK 1907), and the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project.

References

[1] M. Baldoni, G. Boella, and L. Van Der Torre. Roles as a coordination construct: Introducing powerjava. *Electronic Notes in Theoretical Computer Science*, 150(1):9–29, 2006.

[2] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. Technical report, 1998.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proc. of the Int. Conf. on Aspect-oriented software development*, pages 83–92. ACM, 2004.

[4] J. Bonér. Aspectwerkz-dynamic aop for java. In *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[5] N. Cardozo and S. Clarke. Context slices: Lightweight discovery of behavioral adaptations. In *Proc. of the Context-Oriented Programming Workshop*, COP'15, pages 2:1–2:6. ACM, July 2015.

[6] K. B. Graversen and K. 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.

[7] C. He, Z. Nie, B. Li, L. Cao, and K. He. Rava: Designing a java extension with dynamic object roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th IEEE International Symposium and Workshop on*, pages 7–pp. IEEE, 2006.

[8] S. Herrmann. Programming with roles in objectteams/java. In *proc. AAAI Fall Symposium*, 2005.

[9] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[10] T. Kamina, T. Aotani, and H. Masuhara. Eventcj a context-oriented programming language with declarative event-based context transition. In *Proc. of the 10th Conference on Aspect Oriented Software Development*, AOSD'11, pages 253–264. ACM, March 2011.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP 2001Object-Oriented Programming*, pages 327–354. Springer, 2001.

[13] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. A metamodel family for role-based modeling and programming languages. In *Software Language Engineering*, pages 141–160. Springer, 2014.

[14] M. Leuthäuser and U. Aßmann. Enabling view-based programming with scroll: Using roles and dynamic dispatch for establishing view-based programming. In *Proc. of the Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 25–33. ACM, 2015.

[15] S. Monpratarnchai and T. Tetsuo. The implementation and execution framework of a role model based language, epsilonj. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Int. Conf. on, SNPD'08.*, pages 269–276. IEEE, 2008.

[16] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Proc. of the 3rd ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pages 233–246. ACM, 2008.

[17] H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[18] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Jour. of Systems and Software*, 85(8):1801–1817, August 2012.

[19] Y. Sato, S. Chiba, and M. Tatsubori. A selective, just-in-time aspect weaver. In *Generative Programming and Component Engineering*, pages 189–208. Springer, 2003.

[20] W. Vanderperren, D. Suvée, B. Verheecke, M. A. Cibrán, and V. Jonckers. Adaptive programming in jasco. In *Proc. of the 4th Int. Conf. on Aspect-oriented software development*, pages 75–86. ACM, 2005.

[21] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Hotwave: creating adaptive tools with dynamic aspect-oriented programming in java. In *ACM Sigplan Notices*, volume 45, pages 95–98. ACM, 2009.

[2] https://commons.apache.org/proper/commons-bcel/

[3] http://asm.ow2.org/