Max Leuthäuser, Uwe Aßmann

**Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for establishing view-based programming**

Erstveröffentlichung in / First published in:


DOI: [https://doi.org/10.1145/2802059.2802062](https://doi.org/10.1145/2802059.2802062)

Diese Version ist verfügbar / This version is available on:

[https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-707720](https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-707720)
Enabling View-based Programming with SCROLL

Max Leuthäuser
Software Technology Group
TU Dresden, Germany
max.leuthaeuser@tu-dresden.de

Uwe Aßmann
Software Technology Group
TU Dresden, Germany
uwe.assmann@tu-dresden.de

ABSTRACT

Present-day software systems have to fulfill an increasing number of requirements rendering them more and more complex. Many systems need to anticipate changing contexts (self-adaptive systems) or need to adapt to changing business rules or requirements (self-optimizing systems). The challenge of 21st century software development will be to cope with these aspects. We believe that the role concept offers a simple way to adapt object-oriented programs to their changing contexts. In a role-based language, an object plays multiple roles during its lifetime. If the contexts change, the dynamic views can be switched easily, and the software system adapts automatically.

However, the concepts of roles and dynamic contexts have been discussed for a long time in many areas of computer science. So far, their implementation in an existing object-oriented language requires a specific runtime environment. Also, classical object-oriented languages and their runtime systems are not able to cope with essential role-specific features, such as true delegation or dynamic binding of roles.

As a solution, this work presents a simple implementation pattern for role-based objects that does not require a specific runtime environment. We also discuss how to apply the implementation pattern of SCROLL for other languages, in particular for behavioral modeling languages in MDSD.

1. INTRODUCTION

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained - on the contrary! - by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by “focusing one's attention upon some aspect”; it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.


Modeling dynamic and complex domains has been investigated for more than 35 years, starting with Charles W. Bachmann in 1977. He proposed role-based modeling [3] to capture both context-dependent and collaborative behavior of objects. Since then, many approaches in different research areas, ranging from data modeling [2] [19] [72] via conceptual modeling [42] [25] through to programming languages [5] [29] [7] [37] emerged. The separateness of these research areas often leads to ignorance of the results of other fields. Consequently, the long period of research on role-based modeling had almost no influence on common software development practice. But as current software systems are characterized by increasing complexity and context-dependence [35], there is a strong demand for new concepts beyond the classical object-oriented design. In detail, while mainstream object-oriented modeling languages, e.g., the Unified Modeling Language (UML) [39], are good at capturing a systems structure, they lack ways to capture the systems behavior, as it dynamically emerges through collaborating objects [3]. Roles are a natural concept capturing the behavior of participants in a collaboration. In addition to that, roles permit...
the specification of interactions independent from the interacting objects. Similarly, roles capture context-dependent behavior of objects, that is only exhibited when the role is played. In turn, the notion of roles can help to tame both increased complexity and context-dependence.

The functionality of complex software systems usually lies beyond the representative capabilities of a single type of software representation. Therefore, an increasingly large variety of heterogeneous representations (e.g., specifications, models, diagrams, programs etc.) are used in the various phases of software development to represent different aspects of a system’s behavior and properties. These essentially represent different conceptual views of the software system, and usually present overlapping information that needs to be kept consistent. As Dijkstra said, the principle of separation of concerns (SoC) requires that a complex problem or system should be looked at in views to simplify its engineering. Hence, techniques to form views, to manage them, and to re-integrate them with the base system are urgently looked for.

Traditional software engineering environments have always been based on a synthetic approach to views in which the different representations of a software system are treated as separate artifacts. As a consequence, the properties of the system under development have to be inferred by synthesizing the information spread over the different views, and the overall coherence of the information has to be ensured by maintaining a large number of pairwise “correspondences” between the separate artifacts. This principle of synthetic views underpinning traditional software engineering environments is also reflected in most of today’s development methods. Indeed, software development by refinement, accompanied by pairwise traceability links to maintain information consistency, is essentially a synthetic one.

We believe it will be beneficial for software engineering, if projective approaches to the creation and evolution of views of software systems are adopted (“everything is a view”). In a projective software engineering environment, none of the views are stored permanently as separate artifacts. Instead, they are all derived on-the-fly from a single, central information source commonly called a single underlying model (SUM). This includes code-level views of the system: In a fully projective environment, there is no single special artifact called the code, which has a dominant role in the lifecycle of a project. Instead, code-level views of the system can be generated on the fly and can be used to add information, just like any other view. The SUM can then be optimized for executability, expressiveness etc. and is free of the need to be parsable. By definition, projective software engineering environments do not require the notion of correspondence links (e.g., traceability links) between views because all views are mutually consistent by virtue of the SUM. They also do not require a strict notion of linear refinement between views since they allow information to be added or changed at any time using the view type that is most appropriate for the stakeholder.

Also at runtime, the projection of systems and their runtime state into views is very important. Several researchers have suggested to establish a field called “models at runtime”, in which the state of a self-adaptive system is maintained as one or several runtime models. In a projective models-at-runtime-system, every time a system enters a new context, a new context-specific view must be derived from the runtime SUM, the single underlying runtime model. During the lifetime of the context, the view must be connected to the system (the runtime SUM) in a causal connection, i.e., changes in the view must be re-integrated into the runtime SUM. When a system leaves the context, the view has to be disabled or abolished. Usually, many contexts, i.e., many views are active at the same time, so that views also influence each other and must be coordinated.

One of the fundamental concepts for view-based separation of concerns in object-oriented systems are roles. Though other forms of views are possible, roles extend the classical object model in a natural way. Roles can be related at runtime to a context as first-class object. (For instance, the role-based language ObjectTeams [23] introduces Teams, simple runtime contexts for roles, as first-class language construct.) Then, the context forms a viewpoint, and its related set of roles a view of the software system. However, role-based separation of concerns at runtime level will fail at the moment. Many of the suggested role-based programming languages have been abandoned by their developers and do not provide a running compiler; others do not provide a runtime system compatible to one of the major platforms. Therefore, we argue that it is necessary to establish the basic concepts of roles (views) and contexts (viewpoints) with an appropriate light-weight tooling, available in a major programming platform so that view-based programming becomes available for the masses. Such a light-weight approach is also beneficial to support different shades of the meaning of roles and contexts [31], because it can be adapted easily by an expert programmer or language engineer.

In this paper, we suggest an implementation pattern for roles and contexts based on dynamic mixins and compiler-translated function calls. The pattern can be hidden in a library, and we present the Scala library SCROLL as an example case study. The pattern can be realized in any language with these prerequisites, in particular in modeling languages with model-based code generation. The pattern uses the standard Java platform and runtime system, and offers dynamic views at runtime, to realize runtime SUM in standard Scala and Java application.

The remainder of this paper is structured as follows. First we summarize the properties of roles (Sec. 2) and views (Sec. 3) as additional introduction. We continue with a section presenting SCROLL (Sec. 4). We show how to use dynamic marker traits, compiler-translated function calls and implicit conversions to realize roles and hierarchic views in Scala. Finally, the evaluation (Sec. 5) tries to classify our work in the context of other contemporary approaches and discusses how to transfer it to modeling languages.

2. PROPERTIES OF ROLES

This section gives an introduction to the properties of roles by analyzing their features. Even if roles have been studied for a long time, the first thorough analysis of them was not published until the year 2000 by Friedrich Steinmann. He identified 15 features of roles that are useful to classify and compare all subsequent approaches. Since then, many languages utilizing roles have been published. However, only two applied Steinmann’s classification scheme, namely [11] and [24]. In this work we will also use the listed features as introduction to the properties of roles, as well as evaluation

1ObjectTeams forms a notable exception in both points.
1. Roles have properties and behaviors.
2. Roles depend on relationships.
3. Objects may play different roles simultaneously.
4. Objects may play the same role (type) several times.
5. Objects may acquire and abandon roles dynamically.
6. The sequence of role acquisition/removal may be restricted.
7. Unrelated objects can play the same role.
8. Roles can play roles.
9. Roles can be transferred between objects.
10. The state of an object can be role-specific.
11. Features of an object can be role-specific.
12. Roles restrict access.
13. Different roles may share structure and behavior.
15. An object and its roles have different identities.
16. Relationships between roles can be constrained.
17. There may be constraints between relationships.
18. Roles can be grouped and constrained together.
19. Roles depend on compartments.
20. Compartments have properties and behaviors.
21. A role can be part of several compartments.
22. Compartments may play roles like objects.
23. Compartments may play roles which are part of themselves.
24. Compartments can contain other compartments.
25. Different compartments may share structure and behavior.
26. Compartments have their own identity.

Figure 1: Friedrich Steimann’s 15 classifying features (1-15), extracted from [42] and the additional ones (16-26) w.r.t. to the context-dependent nature of roles [31].

Criteria. On top of that, the following additional features of roles w.r.t. their context-dependent nature are extracted from [31]. All features are enumerated more compactly in Fig. 1.

16. Relationships between roles can be constrained. If roles depend on relationships, then it might be possible to further constrain them by intra-relationship constraints [19, 7, 34], i.e. irreflectivity, total order or exclusive parthood.
17. There may be constraints between relationships. In contrast to feature 16, this property suggests the existence of inter-relationship constraints, like the subset constraint [20, 19, 10, 34].
18. Roles can be grouped and constrained together. Most approaches suggesting to constrain roles [12, 11, 8] do not permit to group them and apply constraints to a whole group of related roles as suggested in [44, 24].

These three properties specify ways to constrain roles, but do not account for their context-dependence. The use of the term context leads to a dichotomy of its meaning. According to Dey [33], “context [represents] any information that can be used to characterize the situation of an entity”. Thus, everything that can be attributed to an object in a situation contributes to its context. But within modeling languages, context represents a collaboration or container of a fixed, limited scope [15, 27, 38, 50]. To overcome this dichotomy, researchers avoided the term context by using other terms, i.e. Environments [44], Institutions [4], Teams [23] and Ensembles [22]. In turn, we use the term Compartment as a generalization of these terms to denote an objectified collaboration with a limited number of participating roles and a fixed scope.
19. Roles depend on compartments. Roles are dependent on some sort of context. We call them compartments [23, 44, 4, 15, 27, 38, 50, 22]. A typical example of a compartment is a university, which contains the roles Student and Teacher collaborating in Courses [21, 6, 53].
20. Compartments have properties and behaviors like objects [15, 27, 58, 50].
21. A role can be part of several compartments [1, 14, 15, 22]. This property suggests that a role can be part of more than one compartment. Consider again the role type Teacher. It can be used in different compartments, i.e., School or University, where it might be implemented and constrained differently [4].
22. Compartments may play roles like objects. While most approaches use compartments as a grouping mechanism, compartments can be seen as entities similar to naturals being able to play roles, as well [15, 24].
23. Compartments may play roles which are part of themselves. Continuing the argument of feature 22, compartments might be allowed to play roles belonging to the same compartments, as possible in [15, 24].
24. Compartments can contain other compartments. [24, 27, 29]. This nesting is proposed to further structure compartments into smaller sub-compartments [24, 29] and, e.g. enables the representation of a university containing academic departments which in turn contain faculties.
25. Different compartments may share structure and behavior [26, 27]. Compartments may inherit properties, features, roles, and constraints from each other. However, to fully support inheritance and polymorphism of compartments, the rules of family polymorphism have to be applied [26].
26. Compartments have their own identity. This feature is acknowledged by all approaches who treat compartments as first-class entities of the instance level [41, 24, 32, 35, 29, 22]. This feature is a prerequisite for the existence of compartments at runtime.

Researchers have successfully applied the concept of roles to the domain of context-aware systems. This has led to a number of new features attributed to roles affecting both model and instance level. Surprisingly, the definitional dependence of roles [17] is still applicable to compartments representing the definitional boundary and execution scope for their enclosing roles. Hence, the first 18 features highlight the relational nature of roles whereas the last eight emphasize the context-dependent nature of roles.

3. PROPERTIES OF VIEWS

A view is a representation of a whole system from the perspective of a related set of concerns. This meets Dijkstra’s vision not to tackle all aspects of a program at once which would be highly ineffective. In this sense, Dijkstra’s ideas also inspired the IEEE proposal for a standard for software architecture [25] where a set of related concerns is called viewpoints. They govern views (in the sense that each view conforms to exactly one viewpoint) and generate them by projecting parts of a system. For instance, the architectural viewpoint comprises several architectural concerns, such as coarse-grain structure, run-time process structure and run-time connections. This is distinguished from the application-specific viewpoint with the application-specific details of the implementation of components.

Therefore we assume in the following that views are partial and constructive representations of a system if they can be composed to the full representation of it. Several details are important when analyzing the features of views in the sense of modeling, programming and implementing the
4. INTRODUCING SCROLL

This section presents a light-weight implementation pattern for roles and context for dynamic view-based programming. The pattern is demonstrated with the SCROLL (SCala ROles Language) library, a small Scala package that allows for augmenting an object’s type at runtime with dynamic role types embedded in reified contexts, so-called compartments. They are related to a set of roles. Entering such a context within a running system will activate its compartment and all its related roles (if they are bound to player objects). Hence, a compartment is related to a subset of a system (a view), i.e., all its related roles, which can be switched on and off by its activation or deactivation. When activating a compartment relating to a context, the roles of the compartment are merged into their players. Therefore, compartment-based composition switches on and off context-specific views, and compartments in SCROLL are viewpoints switching on and off dynamic views of the system. Additionally, SCROLL compartments may be nested. Therefore, compartments form hierarchical views, and switching on a compartment means to activate all views of all enclosing ones. To demonstrate that the concept of compartments can mimic the one of views the features of compartments encapsulated in roles of different kinds of behavior can be considered. Accessing role-related features, like attributes or functions that are not natively available on the current instance of a player or role object (e.g., the attribute name in Line 6 of Fig. 3) are accessible through the +-Operator, which implicitly converts the current instance to a compound dynamic type enabling the application of the compiler rewrite rules w.r.t. the Dynamic Marker Trait as explained in Sec. 3. After specifying the plays-relationship between all roles and the player instance (Fig. 4 on line 30) the views can be merged. This provides symmetric view-based composition, whereas asymmetric composition (i.e., extension) is available through the standard inheritance mechanism from Scala itself.

Using implicit conversions, the compiler-converted function calls, the representation of roles and compartments can be completely hidden. Therefore, view decomposition of a runtime SUM as well as view composition, the merging of different views into the runtime SUM, is very simple using the SCROLL library.

5. EVALUATION

It is necessary to investigate how well the implementation with SCROLL using and merging compartments as views as well as binding roles dynamically blends into contemporary approaches. Thus, we use the previously defined classifica-

```
1. implicit val dd = From(_.isInstanceOf[RoleA]).
2. To(_.isInstanceOf[RoleB]).
3. Through(ctorView).
4. Bypassing(_.isInstanceOf[RoleA] || _.isInstanceOf[RoleB]).
```

The correct dispatch behavior for the call is stored in variable dd. Due to the implicit keyword, it can be passed implicitly to any function call on a role-playing object, and this leads to the selection of the appropriate context-specific variant of the function function. When the function is called, along any path in the object-role-playing graph from RoleA to RoleC, (calls to From, Through, and To) the roles RoleA and RoleB will be skipped (call to Bypassing), before a role RoleC is found that is invoked. On the downside compiler-translated function calls cannot be debugged easily. Applying compiler rewrite rules hides important typing information to the tooling typically used by most developers, i.e. IDEs with debugger and link tracers. Writing plugins for those IDEs (e.g. Eclipse, IntelliJ) would overcome this issue and is currently under development.

Second, the implementation pattern in the SCROLL library relies on implicit conversions. Scala’s implicit classes allow for packing player and role types into compound dynamic types. All important role features are exposed this way, e.g. adding, removing and transferring roles or accessing role functions and attributes. We illustrate this with a larger example (Fig. 2 and Fig. 3).

The implementation of a robot is separated into four views: one regarding its navigation (NavigationView), one for querying its sensors (SensorView), one for utilizing its actors (ActorView) and finally one for specifying certain behavior (BehaviorView). For the sake of simplicity, the views here only contains one role each, like the ServiceRole, which offers simple movement. In a real system, multiple implementations encapsulated in roles of different kinds of behavior can be considered. Accessing role-related features, like attributes or functions that are not natively available on the current instance of a player or role object (e.g., the attribute name in Line 6 of Fig. 3) are accessible through the +-Operator, which implicitly converts the current instance to a compound dynamic type enabling the application of the compiler rewrite rules w.r.t. the Dynamic Marker Trait as explained in Sec. 3. After specifying the plays-relationship between all roles and the player instance (Fig. 4 on line 30) the views can be merged. This provides symmetric view-based composition, whereas asymmetric composition (i.e., extension) is available through the standard inheritance mechanism from Scala itself.

Using implicit conversions, the compiler-converted function calls, the representation of roles and compartments can be completely hidden. Therefore, view decomposition of a runtime SUM as well as view composition, the merging of different views into the runtime SUM, is very simple using the SCROLL library.

```
Figure 2: An example for the need of customizable role dispatch. It is ambiguous which role is responsible for answering a call to \texttt{function()}. Flat-roles (roles can not play roles themselves, left side) or deep-roles (right side) are semantically the same and are introducing the same ambiguity here.

Figure 3: Class \texttt{Robot} is constructed (dotted arrows) from different views and plays (solid arrows) the contained roles.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>16. Relationships between roles can be constrained.</td>
<td>No. Since there are no first class relationships yet no constraints can be applied.</td>
</tr>
<tr>
<td>17. There may be constraints between relationships.</td>
<td>No. Since there are no first class relationships yet no constraints can be applied.</td>
</tr>
<tr>
<td>18. Roles can be grouped and constrained together.</td>
<td>Partly. Technically one can import roles from anywhere.</td>
</tr>
</tbody>
</table>
pattern in the library, such as the Role-Object Pattern \[9\], to provide dynamic mixins simulated by decorator chains. Such a pattern can always then be added as a workaround, if the language meets the requirements. On the downside, the Role-Object Pattern does not directly support hierarchies of views and needs to be implemented carefully to avoid object schizophrenia \[2\].

If the language offers dynamic mixins, the concept of compartments, i.e., hierarchical views, can be implemented in a similar way as in this paper. In particular, dynamic mixins may also be available in modeling languages, at least with the help of code generation. In the final step of model-driven software development, when code is generated from the models, patterns such as Role-Object Pattern can be employed for simulated dynamic mixins of objects. This scheme is used in the SMAGS system for a flexible implementation of a role-based architectural language \[10\]. On top of such a code generation scheme, a SCROLL-like library can easily be implemented. This indicates that the SCROLL approach is also very useful for modeling languages, because it requires a minimal set of features from the language, and nevertheless, provides hierarchical views. And finally, some modeling environments, such as the Eclipse Modeling Framework (EMF), already provide dynamic proxies, on which dynamic mixins can be built.

### 6.2 Multiple Inheritance / Traits

Although the concepts of multiple inheritance and traits are semantically perfectly fine to implement roles at runtime, they will lead to a very static system with an exponential blowup in the number of required classes for every new view or compartment one needs to add. Additionally, parallel object hierarchies may occur where cross-tree constraints are very hard to maintain.

![Figure 4: The robot is constructed from views.](image)

```
1  case class Robot(name: String)
2  
3  object BehaviorView extends Compartment {
4    case class ServiceRole() {
5      def move() {
6        val name: String = +this.name()
7        val target: String = +this.getTarget()
8        val sensorValue: Int = +this.readSensor()
9        val actor: String = +this.getActor()
10       info(s"My name is $name and moving to the $target with my sensor w.r.t. sensor value of $sensorValue.")
11      }
12    }
13  }
14  
15  object NavigationView extends Compartment {
16    case class NavigationRole() {
17      def getTarget = "kitchen"
18    }
19  }
20  
21  object ObserverView extends Compartment {
22    case class ObservingRole() {
23      def readSensor = 100
24    }
25  }
26  
27  object DriveableRole() {
28    def getActor = "wheels"
29  }
30  
31  val myRobot = Robot("Pete") play ServiceRole() play NavigationRole() play ObservingRole() play DriveableRole()
32  BehaviorView merge NavigationView merge SensorView merge ActorView
33  myRobot.move()
```

Table 1: Comparison of coeval roles at runtime based on 26 classifying features extracted from the literature \[31, 32\]. It differentiates between fully (■), partly (□) and not supported (□ □) features.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Chameleon</th>
<th>OT/J</th>
<th>Java</th>
<th>powerJava</th>
<th>JavaRat</th>
<th>Runem</th>
<th>NextEJ</th>
<th>JavaVM</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

6.3 Aspect / Subject-oriented programming

Aspect-oriented programming allows to implement cross-cutting concerns via joint-points and pointcuts. Often the composition is done statically although there exist a few dynamic approaches. E.g. ObjectTeams/Java (OT/J) \[23\] uses dynamic aspect weaving at bytecode-level for adding role-specific behavior to its players. Subject-oriented programming utilizes different class hierarchies from different perspectives which are comparable to view-based programming. On the downside there is no real composition language and the set of composition operators is fixed. Furthermore, no real control flow on the composition itself exists.

6.4 Delegation and Delegation-Layers

Basically delegation mimics the inheritance mechanism on object level. This requires (the generation of) a lot of management code and leads to object schizophrenia \[2\]. Delegation-Layers on the other hand define layers that group behavior for sets of objects and for sets of classes. Sadly, it implies fixed hierarchies and thus a system design that is too static.

6.5 Other role-based programming languages

Interestingly, most of the existing role-based programming languages are extensions to Java. They are either compiled to Java source code \[10, 21\] or to bytecode \[23\] directly. The first class of these languages focuses mainly on implementing objects playing roles. **Chameleon** \[10\] features roles with so called constituent methods allowing to overwrite methods of their players, which work like advices in aspect-oriented programming. However, the major drawback of Chameleon is the fact that roles ex-

---

**Table 1**: Comparison of coeval roles at runtime based on 26 classifying features extracted from the literature \[31, 32\]. It differentiates between fully (■), partly (□) and not supported (□ □) features.
tend their player to gain access to the player, which is both conceptually wrong [12] and limits the flexibility of roles. **Rava** [21] overcomes these issues by employing the **Role-Object-Pattern** [9] extended with the **Mediator-Pattern** [14]. They use special keywords to steer the generation of the necessary management code. Due to the use of the Role-Object Pattern and generation to plain Java, this solution suffers from object schizophrenia [23]. **JavaStage** [8] eludes this problem, by only supporting static roles, i.e., the roles are directly compiled into the possible players as inner classes. To avoid name clashes, it employs a customizable method renaming strategy. Its main advantages are the capability to specify a list of required methods instead of a specific player class. Surprisingly, this approach limits itself to static roles unable to represent their relational and context-dependent nature. We proceed with **Rumer** [7], which contributes relationships as first class citizens and modular verification over shared state. Furthermore, it provides several intra-relationship constraints usable to restrict these relationships. Roles are the named places of a relationship with attributes and methods but without inheritance. Despite that, roles are only accessible within a relationship and not from their player. The most sophisticated approach to context-dependent roles so far is **ObjectTeams/Java** (OT/J) [23]. Similar to Chameleon above, OT/J allows to override methods of their player by aspect weaving. Besides that, it introduces **Teams** to represent compartments whose inner classes automatically become roles. Notably, OT/J supports both the inheritance of roles and teams whereas the latter leads to family polymorphism [20]. On the downside, it does neither support multiple unrelated player types for a role type nor first class relationships and only a limited form of constraints. This is similar to **powerJava** [3], which also introduces compartments, denoted **Institutions**, whose inner classes represent roles. However, powerJava features the distinction between role interface and role implementation where the former is callable from outside a specific institution and the latter is the institution-specific implementation of the same interface. Both Rava and powerJava are the only research prototypes providing a working compiler. Nevertheless, the project has been abandoned [43]. A more recent approach towards context-oriented programming is **NextEJ** [29] as the successor of **EpsilonJ** [40]. It provides **Contexts** as first class citizens which do not only group roles but also represent an activation scope at runtime. These **context activation scopes** can be nested and act as a barrier where all roles are instantiated and bound automatically. So far, they only published their type-system of the core calculus and no compiler for NextEJ. Consequently, all systems containing objectified contexts as first class citizens, e.g., **Environments** [11], **Institutions** [4], **Teams** [23] and **Ensembles** [22] like SCROLL does (i.e. with **Compartment**s) are suitable for adaption w.r.t. establishing views at runtime. This has to be investigated in the future, up to our knowledge there is currently no literature available on this.

7. CONCLUSIONS

This paper presented an implementation pattern for role-based objects and their hierarchical contexts in Scala, based on dynamic marker traits, compiler-translated function calls, and implicit conversions. The SCROLL library provides view-based programming on top of a standard Scala platform. SCROLL roles and compartments can help to handle both increased complexity and context-dependence of software systems, because with its light-weight approach, projective views on a runtime JVM are made available. The SCROLL approach can be transferred to other languages as well, in particular to modeling languages with model-based code generation. We believe that this is a contribution to the ubiquitous adoption of projective view-based approaches to software engineering - supporting the design metaphor that “everything is a view”.

Acknowledgments

This work is funded by the German Research Foundation (DFG) within the Research Training Group “Role-based Software Infrastructures for continuous-context-sensitive Systems” (GRK 1907).

8. REFERENCES

IEEE. Ieee 1471-2000 conceptual framework for

S. Herrmann. Programming with roles in

S. Herrmann, C. Hundt, and K. Mehner. Translation

S. Herrmann. A precise model for contextual roles:

R. Hennicker and A. Klarl. Foundations for ensemble

C. He, Z. Nie, B. Li, L. Cao, and K. He. Rava:

A. K. Dey. Understanding and using context.


