

Subjectopia

Unifying Subjectivity

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Daniel Langone
28. February 2011

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Jorge Ressa
Institut für Informatik und angewandte Mathematik

Acknowledgments

I would like to thank my supervisor Jorge Ressia for all his invested hours and all the input – Prof. Oscar Nierstrasz for all the support for the master thesis – Tudor Gîrba and Lukas Rengli for the review – my friends Urs Ochsner and David Bärtschi, my family and Irene Mühlemann for their patience and getting me always back on earth.

Abstract

Subjective behavior is essential for applications that must adapt their behavior to changing circumstances. Many different solutions have been proposed in the past, based, for example, on perspectives, roles, contextual layers, and “force trees”. Although these approaches are equally expressive, each imposes a particular world view which may not be appropriate for all applications. We propose a unification of these approaches, called SUBJECTOPIA, which makes explicit the underlying abstractions needed to support subjective behavior, namely *subjects*, *contextual elements* and *decision strategies*. We demonstrate how SUBJECTOPIA subsumes existing approaches, provides a more general foundation for modeling subjective behavior, and offers a means to alter subjective behavior in a running system.

Contents

1	Introduction	1
2	State of the Art	3
2.1	Self Delegation as an Exemplary Mechanism for Subjective Behavior	4
2.2	Perspectives	6
2.2.1	Perspectives as Views of an Object	6
2.2.2	US – A Perspective-Based Programming Language	7
2.2.3	Symmetry Problem in Perspectives	8
2.2.4	Perspective – Receiver Symmetry	9
2.2.5	Perspective’s Drawbacks	9
2.3	Roles	10
2.3.1	Subjects – Objects with Roles	10
2.3.2	Forces Affecting Subjects	12
2.3.3	Role’s Drawbacks	13
2.4	Context-oriented Programming	13
2.4.1	Context as Layers	13
2.4.2	Context Layer Activation Mechanisms	15
2.4.3	Context Variables	16
2.4.4	COP’s Drawbacks	16
2.5	Subjective Message Behavior	16
2.5.1	Forces Influencing Behavior of Objects	16
2.5.2	Force Tree - A Decision Dispatching Mechanism	17
2.5.3	SMB’s Drawbacks	18
2.6	Problems in Current Subjectivity Models	19
3	SUBJECTOPIA	21
3.1	Subjects	22
3.2	Decision Strategies	22
3.3	Contextual Elements	25
3.4	Modeling Previous Approaches in SUBJECTOPIA	27
3.4.1	Perspectives as Decision Strategy	27
3.4.2	Roles as Decision Strategy	28
3.4.3	COP as Decision Strategy	29
3.4.4	SMB as Decision Strategy	30
3.5	Implementation	30

4	Validation	35
4.1	Case Study – Mobile Mail Application	35
4.2	Case Study – Group Programming Application	38
4.3	Case Study – Bank Account Application	40
4.4	Case Studies in Moose	40
4.4.1	Short Introduction to Moose	40
4.4.2	Case Study – Subjective Behavior of Group of Entities in Moose	42
4.4.3	Case Study – Subjective Behavior Influenced by Third-Party Entities in Moose	44
4.4.4	Case Study – Subjectives menus in Moose	46
5	Conclusion	48
5.1	Future Work	49
	Appendices	53
A.1	Installation Guide	53
A.2	Introductory Example for SUBJECTOPIA	54

1

Introduction

We, as humans, generally strive to be objective, that is we try to behave in a unique and consistent way, independent of personal feelings or external influences. In practice, however, we are often required to behave *subjectively*, that is, we must adapt our behavior depending on circumstances.

In fact, real world entities are subjective. We have learned, for example, in the 20th century that physical measurements are relative to the frame of reference used by the observer. As a consequence, real-world problem domains that we model in software applications are also subjective. The elements that collaborate to achieve a common goal may need to adapt their behavior under given circumstances such as for specific events or conditions. An employee of a bank, for example, does not reveal the current balance of a bank account to all persons asking for it. Only authorized persons may asking for the balance to obtain it. The behavior of the employee is subjective to the authorization circumstance.

Object oriented software systems consist of objects which interact by sending messages. Objects follow the *objective* approach, that is they behave always the same way when receiving the same message. We can reach subjective behavior in object oriented by using static idioms or patterns such as self-delegation and double or multiple dispatch patterns [1]. Moreover, objects model their behaviors as if they were intrinsic to them, which makes adapting the object for new requirements difficult. Therefore, we force the developer of the object to forego the advantages of the object-oriented style or anticipate all future adaptations. To faithfully model real-world domains we need other mechanisms to model subjectivity.

Subject-oriented programming was first introduced by Harrison and Ossher [2]. They advocated the use of subjective views to model variation to avoid the proliferation of inheritance relations. We analyzed other models for subjectivity and extracted the proposed key approaches as follows:

Perspectives. Smith proposed adding multiple *perspectives* to an object. Each perspective implements different behavior for that object [3]. When an object sends a message through a perspective the receiver behaves differently depending on the perspective chosen by the sender of the message. Perspectives model subjective behavior as a set of different views for an object which influences its behavior.

Roles. Kristensen introduced the concept of *roles* to model subjective behavior [4]. We attach a role to an object to specify additional or modified behavior. Kristensen explicitly models *subjects* – objects with roles – whose behavior depends on the role they are playing for the sender of a message.

COP. Costanza *et al.* introduced *Context-oriented programming* (COP) [5]. COP splits the behavior of an object into layers that define the object’s behavior for a particular context. We can activate and deactivate layers to represent the actual contextual state. When an object sends a message the active context determines the behavior of the object receiving the message.

SMB. Darderes and Prieto proposed *subjective message behavior* (SMB) [6]. SMB splits the different behaviors for a message into a set of independent methods. A tree based decision mechanism, called *force tree*, determines the behavior of the object when object receives the message.

Although formally the approaches are equivalent in expressive power they are not equally suitable in all circumstances. The subjectivity models provide no cohesive sense of what subjectivity exactly is. Each of these approaches imposes a particular modeling paradigm which may be appropriate for certain problem domains, but not for others. Consider the use case where a user sends an email using a mobile device [6]. The application sends the email immediately if the network is available. Otherwise it retains the email and sends it when possible. Roles and perspectives are not suited to model the network object in contrast to COP and SMB. The subjective behavior of the network is not about a sender knowing or seeing an object through a role or perspective. Nevertheless, the network is either available or not in the current context. Therefore, COP, providing explicit modeling of context, is more appropriate to model subjectivity in this domain. Also SMB suits this problem domain as it models the subjective decision at the side of the network object.

Furthermore, the responsibility of determining which subjective behavior to select may lie varyingly with the sender of a message, the receiver, or even the context. For example, in the perspective- and role-based approaches it is the sender of the message which determines the used perspective or role. In SMB it is the receiver of the message that evaluates the force-tree and thus selects the behavior. Consider communicating with a person who might be at work or on holidays. The same message triggers different responses depending on the context the person is. Nevertheless, it makes more sense for the receiver or the context and not the sender to determine the subjective behavior.

Subjectivity models so far force us to look at the problem domain from a certain point of view. Moreover, they fix the decision for the subjective behavior either at the receiver, sender or context.

SUBJECTOPIA To alleviate these problems, we propose a framework, called SUBJECTOPIA, which unifies and generalizes the earlier approaches. SUBJECTOPIA reifies three key abstractions that are only implicit in the other approaches. A *subject* is an object that behaves subjectively. Any object can become a subject. *Decision strategies* explicitly model subjective behavior. A decision strategy determines the appropriate subjective behavior based on the value of a set of *contextual elements*. We can configure decision strategies to model roles, perspectives, force trees or layers, thus subsuming the earlier approaches. Furthermore, they can be dynamically adapted at runtime, which is important for adapting long-lived software systems.

Chapter 2 presents a review of previous approaches to modeling subjective behavior. In Chapter 3 we explain how SUBJECTOPIA models the subjective behavior of objects and discuss our implementation. Chapter 4 validates our approach by showing the drawbacks of previous approaches in solving subjective problems and demonstrates how SUBJECTOPIA circumvents these shortcomings. In Chapter 5 we summarize the paper and discuss future work. In Section A.1 we show how to install and use SUBJECTOPIA in Pharo [7].

2

State of the Art

Developers often use delegation and dispatching patterns [1] to model subjective behavior. Even if these solve the problems they do not provide a model for subjective behavior. This moved Harrison and Ossher to propose the first subjectivity model [2]. Inspired by this idea, other researchers came up with more and different models for subjective behavior. Instead of unifying current subjectivity approaches they imposed new interests and requirements to what subjectivity is and how to model it. As we are aiming to unify the current subjectivity models we analyze the requirements to subjectivity based on the previous approaches. We used the results of the analysis to define the SUBJECTOPIA approach.

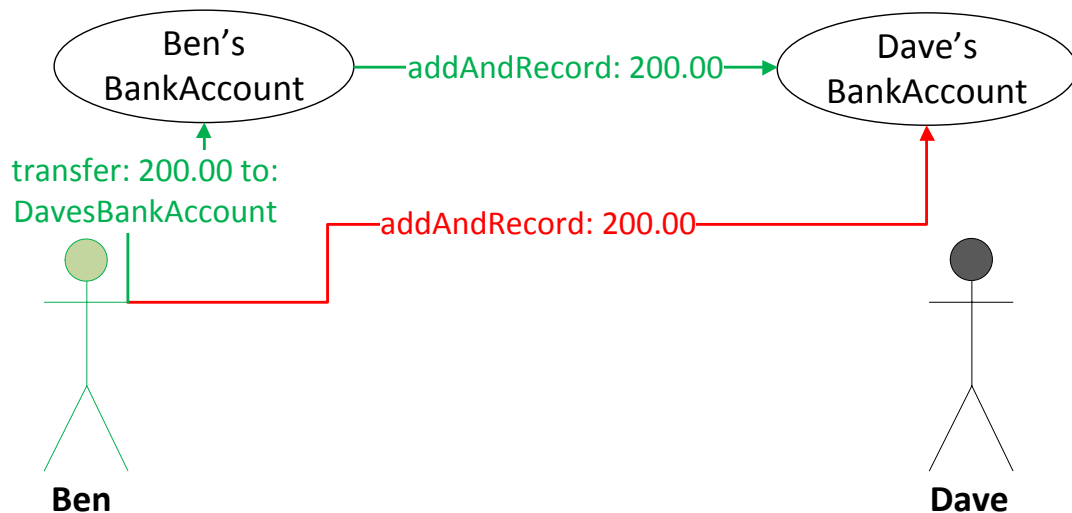


Figure 2.1: The message `addAndRecord:` sent from Ben's Bank Account – green path – should have an impact on the balance, whereas the same message sent from Ben – red path – should not.

Our analysis focuses on the most relevant models of subjective behavior. In this section we explain

the subjective models and discuss their drawbacks. To this end we use an explanatory and canonical bank account example described in Figure 2.1 [3]. The use case¹ consists of users transferring money through bank accounts. The user object sends the message `transfer:to:` to its bank account indicating as arguments the amount of money and to which bank account to transfer the money. Next, the bank account object decreases its balance by the amount of money to be transferred and sends the message `addAndRecord:` to the bank account receiving the money. A user should not be able to directly send the message `addAndRecord:` to a bank account to guarantee that only bank accounts trigger transfers of money to maintain the balance invariant in the whole banking application. As a consequence the bank account behaves subjectively for the message `addAndRecord:`, depending in whether the sender of the message is a user or a bank account object.

Before we start introducing the four most relevant approaches to model subjective behavior we propose a possible object oriented solution for the bank account use case. We show how object oriented programming, even with patterns, does not model subjective behavior in all problem domains.

2.1 Self Delegation as an Exemplary Mechanism for Subjective Behavior

We use the self delegation pattern when the receiver of the message needs a reference to the sender of the message [1]. Since the subjective behavior of the bank account objects for the message `addAndRecord:` depends on the sender of the message, we use self delegation.

Figure 2.2 shows the class hierarchy and the sequence diagram of our object oriented solution for the bank application. We assume that most of the bank entities are not allowed to change the balance of a bank account. Hence, we define the default behavior for the message `allowedToAdd` in the `BankEntity` as follows:

```
BankEntity>>allowedToAdd
^False
```

All bank entities that are not allowed to change the balance of a bank account by sending the message `addAndRecord:`, reuse the default behavior for the message. Bank entities that can change the balance of a bank account override the behavior for the message `allowedToAdd` as follows:

```
BankAccount>>allowedToAdd
^True
```

The class `BankAccount`, for example, overrides the message, whereas the class `BankUser` does not. The method `addAndRecord:` message in the class `BankAccount` looks as follows:

```
BankAccount>>addAndRecord: aNumber from: anObject
(anObject allowedToAdd)
ifTrue:
    [balance := balance + aNumber.]
ifFalse:
    [self error: 'Rejected']
```

The use case demands that the behavior of the bank account for the message `addAndRecord:` depends on the class of the object sending the message. Instead of determining the class of the sender object we override the message `allowedToAdd`. We directly define in the class of the bank entity whether we allow an object to change the balance of the bank account or not. New bank entities do not require changes the method `addAndRecord:` of the bank account class.

¹Readers unfamiliar with the syntax of Smalltalk might read the code examples aloud and interpret them as normal sentences: An invocation to a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2.`

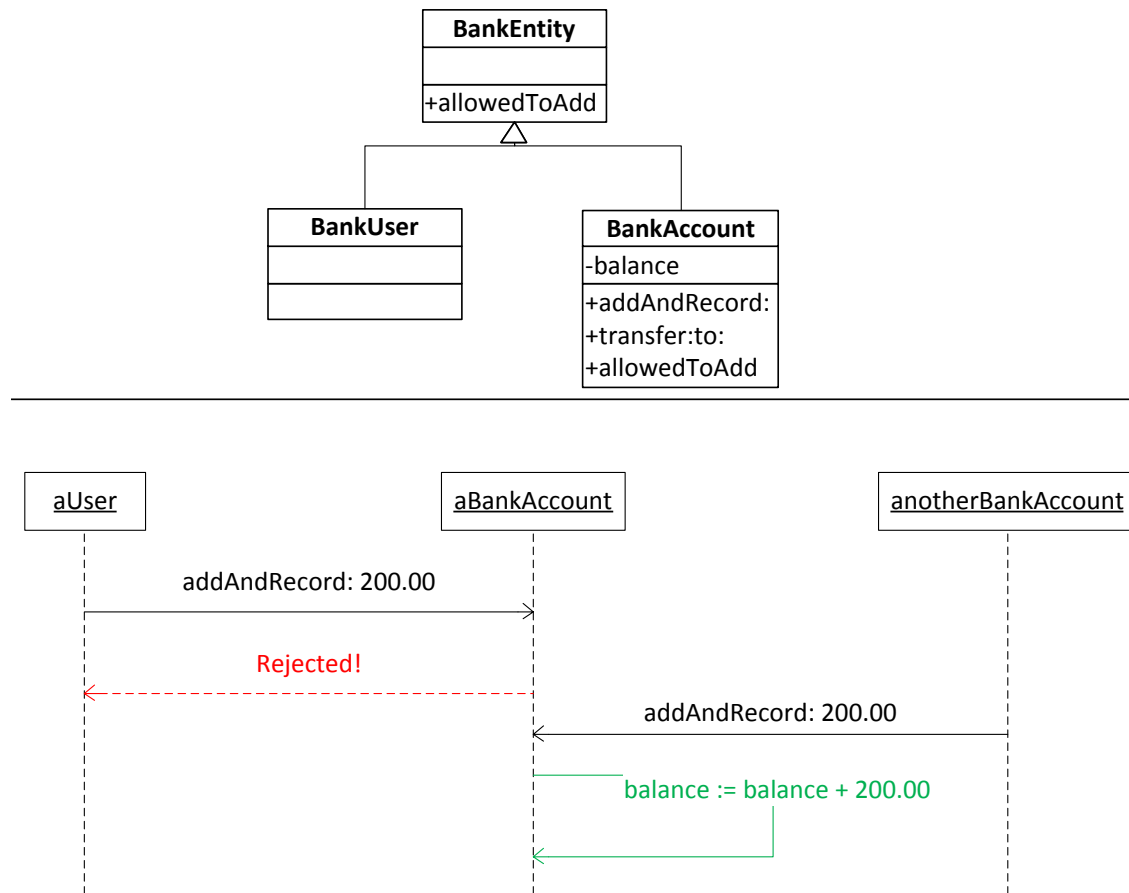


Figure 2.2: Class hierarchy and sequence diagram of object oriented bank account solution using self-delegation.

We cannot use patterns to generally model subjectivity, as they are only suitable for some particular problem domains. The self delegation pattern, for example, only solves use cases where the behavior of an object depends on properties of the message sender. Moreover, the self delegation pattern does not model the subjective behavior of the bank account object realistically. The bank account asks the sender of the message if he is allowed to send `addAndRecord:`. In reality a customer of the bank is not asked if he is allowed or not to change the balance of a bank account. Hence, self delegation does not reflect reality for this particular use case.

Patterns also miss explicit objects that model the subjective behavior. Patterns rely on the object oriented paradigm defining all the behavior of an object to be intrinsic to it. The behavior for the message `addAndRecord:` is not intrinsic to the bank account object, because it also depends on the sender of the message. To overcome the problem, patterns split the behavior of a subjective messages all over the application, which makes adaptation of subjective behavior tedious work. When the requirements for the behavior `addAndRecord:` change, we have to adapt at least the methods `allowedToAdd` and the method `addAndRecord:` in the bank account class.

The mentioned limitations moved researchers to define their own model for subjectivity. We next introduce the four most relevant subjective behavior models.

2.2 Perspectives

Smith and Ungar define Subjectivity as a system having multiple overlapping but not identical *realities*. The perception of a system always depends on the frame of reference of the observer. For example, the perception of a bank account is different for another bank account than for a customer of the bank. To this end, Smith and Ungar propose to model subjective behavior as a set of possible *views* towards an object [3].

2.2.1 Perspectives as Views of an Object

Perspectives model the different *views* that objects have when sending a message. A perspective is a composition of zero or more hierarchically ordered *layers*. Each layer links to its parent layer. The object viewing another object through a perspective sees the composition of the perspective's starting layer and all its layer parents. Each layer is a composition of *pieces* modeling one behavior for one message and one object. Each object has exactly one, possibly empty, piece on each layer. Smith and Ungar also mention how to use pieces to model additional instance variables for an object. In our discussion we focus on the behavior only, but perspectives can be easily extended to support subjective instance variables as well.

The behavior of an object receiving a message depends on which perspective the sender of the message *views* the receiver object through. *Subjective objects* are objects behaving subjectively because of having perspectives.

To model the bank account, for example, we need two perspectives. We call the perspective for user objects `UserPerspective` and the perspective for bank account objects `BankAccountPerspective`. Each perspective consists of one layer and one piece. The pieces model the two possible behavior for the message `addAndRecord:`, described in the sequence diagram in Figure 2.3. We have to locate each piece in one layer. In our use case each layer models exactly one perspective.

The piece in the `UserPerspective` implements the following behavior for the message `addAndRecord:`:

```
UserPerspectivePiece>>addAndRecord: aNumber
  ^self error: 'Rejected'.
```

The above code notifies the rejection of the request to the object sending the message `addAndRecord:` through the perspective `UserPerspective`. The `UserPerspective` models the view user objects have to a bank account object. The `BankAccountPerspective` defines, through its piece, the following behavior for the message `addAndRecord:`:

```
BankAccountPerspectivePiece>>addAndRecord: aNumber
  balance := balance + aNumber.
```

The `BankAccountPerspectivePiece` changes the balance of the subjective object of the class `BankAccount`. The perspective models how the bank account sending the message views the bank account to which he sends the message. A perspective has access to the instance variables of the object it belongs to.

Figure 2.3 describes the use of the perspectives when sending a message. User objects send the message `addAndRecord:` through the `UserPerspective` to the bank account. The first layer in the perspective defines a piece for the message `addAndRecord:`. The piece models the behavior for user objects sending the message to a bank account. When executing the behavior of the piece the user gets notified about the rejection of its request to change the balance. Bank account objects send the same message through the `BankAccountPerspective` to another bank account. The `BankAccountPerspective` has a piece for the message which changes the balance of the bank account that receives the message.

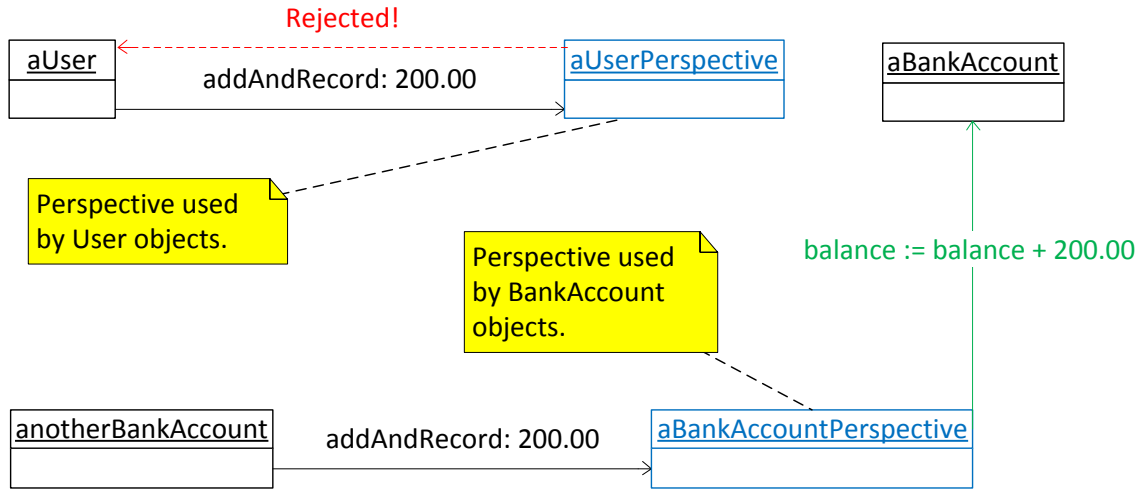


Figure 2.3: Message sending of bank application using perspectives.

Up until now we discussed perspectives consisting of one layer only. Next, we add a new layer to the perspective `UserPerspective`. The new layer has no pieces defined for the message `addAndRecord:` and uses the original layer of the `UserPerspective` as its layer parent. If a user object now sends the message `addAndRecord:` through the `UserPerspective`, the first layer evaluates the message. As the layer does not find a piece corresponding to the message `addAndRecord:`, the evaluation for that message continues at the layer parent. As the parent layer models a piece for the message `addAndRecord:`, the lookup finishes and the object executes the behavior.

Each layer of the `UserPerspective` and the `BankAccountPerspective` contains only one piece for the message `addAndRecord:`. We cannot merge the layers of the perspectives to one layer, because both pieces model the behavior for the message `addAndRecord:` for bank account objects. The perspective based approach is deterministic and consistent if every layer only defines one piece for each message and the class of the object.

2.2.2 US – A Perspective-Based Programming Language

Smith and Ungar introduce the perspective-based programming language *US* [3] based on *Self* [8]. *Self* is a prototype-based object oriented programming language. *Self* uses prototypes to model objects instead of classes. Prototypes do not provide a description but figure as examples of objects. Nevertheless, objects can share behavior by having using *inherits from* relationship. In contrast to classes, prototype-based relationships do not contain any formatting information.

Self uses slots to store behavior and state. In *Self* we activate slots by sending messages. Hence, messages are often sent to `self`, referencing the current object, which gives the programming language its name. Slots either contain normal objects or method objects. The message sending activates the the slot corresponding to the message selector. The activation of a slot either results in returning the object or executing the behavior modeled by the method object.

US extends *Self* with the concept of perspectives by performing the following transformation to an expression in *Self*:

```
expression → expression ⊗ perspective
```

Each expression in *US* requires the use of a perspective to maintain the perspective–receiver symmetry, we discuss in Section 2.2.4. The keyword *here* links to the current perspective just like the keyword *self* points to the current object. Hence, the following expressions are equivalent in *US*:

```
message ↔ message ⊗ perspective  
↔ self message ↔ self message ⊗ perspective
```

The subjective decision is always taken by the sender of the message by selecting the perspective through which it communicates. Consider the implementation describe in Figure 2.3 where a user sends the message `addAndRecord:` to a bank account. The message send in *US* looks as follows:

```
[...]  
aBankAccount addAndRecord: 200.00 ⊗ aUserPerspective  
[...]
```

2.2.3 Symmetry Problem in Perspectives

To discuss a problem arising in changing perspectives during a method invocation we slightly extend our original use case. We additionally model the message `subtractAndRecord:`, subtracting an amount, given as an argument, from the balance of the bank account the message is sent to. Instead of sending `addAndRecord:` we use the message `record:`, implemented as follows:

```
BankAccount>>record: aNumber  
  ^ (aNumber >= 0)  
  ifTrue:  
    [self addAndRecord: aNumber ⊗ aBankAccountPerspective]  
  ifFalse:  
    [self subtractAndRecord: aNumber negated ⊗ aBankAccountPerspective].
```

`aNumber negated` negates the value of the number, thus becomes positive. Hence the decision strategies directly deal with the positive value of the amount to be transferred. Furthermore, consider that the piece for the message `subtractAndRecord:` may only subtract the amount from a bank account if the balance is not negative afterwards. To get the current balance of a bank account the object sends the message `balance` to the bank account. Only bank account objects can view the balance of another bank account, user objects can not. We implement the piece for the message `subtractAndRecord:` in the bank account perspective as follows:

```
BankAccountPerspectivePiece>>subtractAndRecord: aNumber  
  ^ (self balance ⊗ here >= aNumber)  
  ifTrue:  
    [self balance: (balance - aNumber)].
```

Consider the following message send:

```
aBankAccount record: 200.00 ⊗ aUserPerspective
```

Figure 2.4 describes how the message `record:` is sent through the perspective `aUserPerspective`. The `aUserPerspective` uses the `aBankAccountPersepective` in his piece. The keyword `here` used in the `aBankAccountPerspective` for the message `balance` points to the current perspective. We have to answer whether `here` points to the starting perspective `aUserPerspective` or the last active perspective `aBankAccountPerspective`. To answer this question, Smith and Ungar propose two different policies. The *rubber band policy* [3] remembers the perspective used before the message is sent. The message is sent through the new perspective and afterwards switched back to the saved one. Therefore, `here`

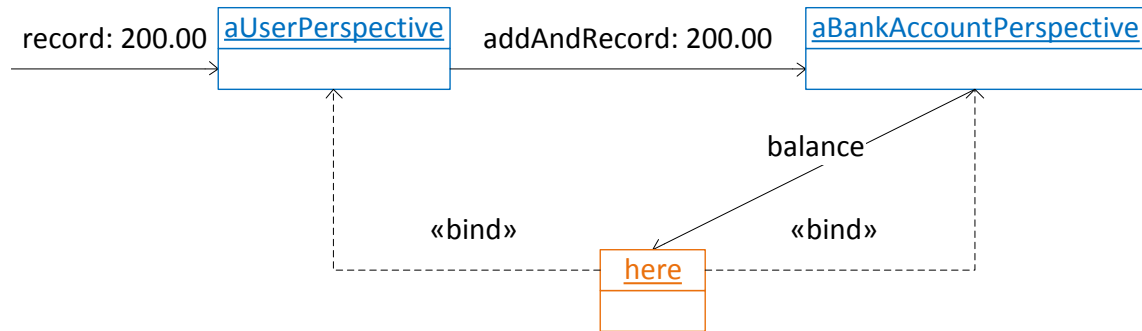


Figure 2.4: Depending on the used policy, *here* points to a different perspective.

corresponds to `aUserPerspective`. The second approach, called *minimal motion policy* [3], states that once a perspective is active it stays active until we explicitly change it. In the minimal motion policy *here* points to the `aBankAccountPerspective`. US implements the second policy to maintain the perspective-receiver symmetry.

2.2.4 Perspective – Receiver Symmetry

We can use perspectives to extend object oriented programming languages with a subjectivity model. There are two kind of perspective based programming languages depending on whether or not they follow the *Perspective – Receiver Symmetry*. A developer has to be aware of this to faithfully model the application based on perspectives.

Consider that a developer models one perspective for the entire system. The system reduces to object oriented style. Every message has to be sent through a perspective. Hence, the behavior of a message is either defined as a piece in the perspective or as a method in the object. Hence, it has the same semantics as object oriented programming languages.

The perspective-receiver symmetry states that a system with only one receiver but multiple perspectives has the same semantics as multiple receivers having only one perspective. Hence, the system reduces to a conventional object oriented system. In this scenario we can translate each perspective to an individual object. The hierarchy of layers corresponds to the hierarchy of the individual objects. To respect the perspective receiver symmetry the system has to use *minimal motion policy*.

Consider the bank account in a perspective based system using the rubber band policy. When we introduced the policies we mentioned that the behavior for the message `record:` depends on the used policy. The bank account still behaves subjectively for the message `record:` under the rubber band policy. Nevertheless, we only have one receiver with multiple perspectives. Hence, the system does not respect the Perspective – Receiver Symmetry.

2.2.5 Perspective’s Drawbacks

The perspective based approach has the drawback that it forces us to translate the problem to suit the model of perspectives. For example, a person behaves subjectively to the same complaint depending if he is at home or at work. Consider modeling the home and work context as perspectives. Still, the sender of the complaint message needs to know the location of the receiver prior to sending the message. The perspective approach has a fixed model implying that it prescribes which perspective to use when sending a message.

The decision of which perspective to use is always taken by the sender of the message. To model the decision at the side of the receiver we require delegation patterns. Considering the bank account example: letting the user choose through which perspective he sees a bank account object does not model reality. We should be able to model the services provided by a bank account to a certain object at the side of the bank account. In this scenario it is the receiver of the message `addAndRecord:`.

Dynamicity is not directly discussed when presenting the perspective based approach. The perspective based approach does not model the evolution of subjective behavior at runtime. Changing perspectives is not supported natively because we hard-code the used perspective when sending a message. Moreover, Smith and Ungar do not talk about how to add new layers to a perspective, other than implicitly in the group programming use case [3].

2.3 Roles

Kristensen [4] stresses the importance of roles in the subjective behavior of entities: “*we think and express ourselves in terms of roles*” when dealing with the real world. Kristensen infers the notion of a role from psychology as a set of connected behaviors, rights and obligations as conceptualized by actors in a social situation.

2.3.1 Subjects – Objects with Roles

The objective behavior of a subject is *intrinsic* to it and called *intrinsic behavior*. Hence, they are directly implemented on the so called *intrinsic objects*. For example, the bank account has an intrinsic behavior for the message `balance`, which always returns the current balance. The subjective behavior is *extrinsic* to the object, because it depends on the context and not only on the object receiving the message. To model extrinsic behavior, Kristensen introduces the notion of *role objects*. For example, the bank account has an extrinsic behavior for the message `addAndRecord:`, whose behavior depends on the sender of the message. We attach role objects to the intrinsic object to add, remove or redefine the latter’s original behavior, making it to behave subjectively. Kristensen calls an intrinsic object together with its role a *subject*.

Kristensen proposes a graphical notation described in Table 2.1 [4] to visualize role based programming.

Figure 2.5 describes the bank account subject consisting of the intrinsic object `aBankAccount` and one intrinsic behavior for the message `balance`. We extend the intrinsic object by the role `BARole` adding a behavior for the message `addAndRecord:` and redefining the behavior for the message `balance:`. The object sending the message to a subject decides which role the subject plays. The sender of the message knows the subject either directly or playing a particular role. For example, a user directly sends the message `addAndRecord:` to the intrinsic object `aBankAccount`.

A subject can have one or more roles. For example, we can extend the bank account subject to model a separate role for the user objects, called `UserRole`. The user role models the behavior for the message `addAndRecord:` for bank account subjects playing the role for user object. User objects send the message `balance` to bank account subjects using the `UserRole`. The role does not model a behavior for that message so it forwards it to the intrinsic bank account object.

Figure 2.6 describes a further extension to our application by defining the `OwnerRole` for bank account subjects. The users, owning the bank account they sent the message to, know the bank account through the owner role. Recall that the original behavior for the message `balance` in the intrinsic bank account object returns an error message. The owner role overrides the `balance` behavior to return the balance of the bank account. The owner role is a specialization of the user role. Hence, the owner inherits all other behaviors from the user role. If an object sends the message `addAndRecord:` using the owner role the bank account

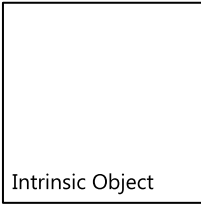
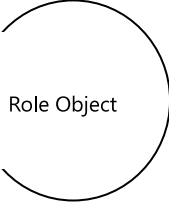

Entity	Graphical notation
Intrinsic Object	
Role Object	
Method	

Table 2.1: Graphical notation for *role-based* systems.

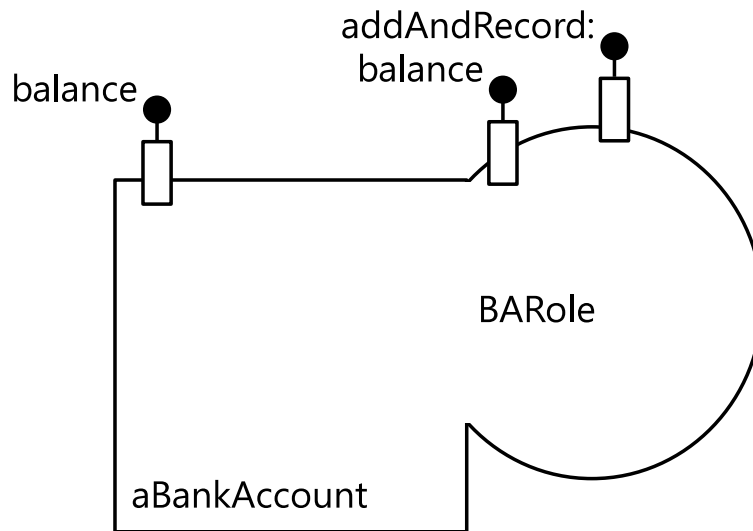


Figure 2.5: The bank account subject with one intrinsic object and one role object.

does not find a method for the message in the owner role. Next, the user role further evaluates the message, which models the behavior for the message `addAndRecord::`.

Dynamicity is an important characteristic of roles, as subjects may change their roles during lifetime. A person, for example, is a student at a given time, becomes an employee and quits being a student. Subjects obey rules when having multiple, overlapping or changing roles. Kristensen refers to it as *Dynamics of Roles* modeling the conditions as regular expressions. Kristensen also introduces a graphical notation to reflect on the dynamicity of roles. [4]. The nature of the dynamicity of roles can be:

Sequential Role instances may only exist in a given sequence. For example, a person getting the professor role needs to have had the student role before.

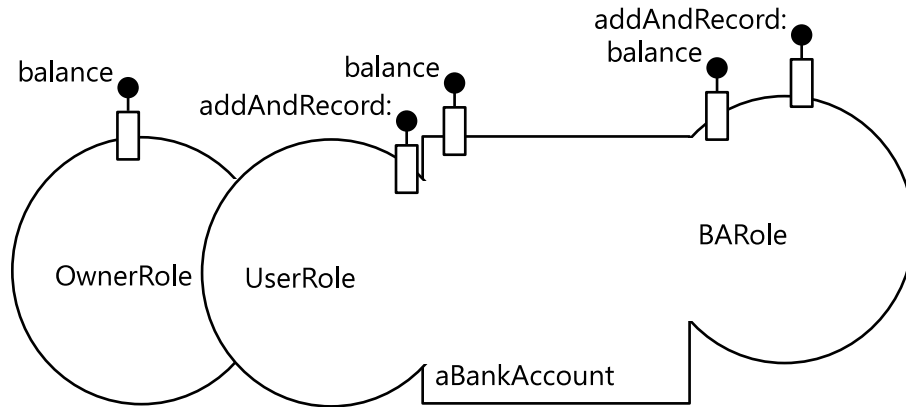


Figure 2.6: The bank account subject with more roles.

Overlapping Role instances may exist in some overlapping form. For example, a person with the role of a student can have the overlapping role of a tutorial assistant.

Composition We can name any mixture of the previous conditions, such as duplication, for example, being a specialization of overlapping roles.

We have seen that a role *is-part-of* the intrinsic object. To model the *is-part-of* relationship Kristensen refers to the location of *part objects* introduced by Madsen and Møller-Pedersen [9]. Location relies on part objects, which, in contrast to a referenced instance variables, belong to the *hosting* object. Roles, for example, belong to a subject and have no responsibility by themselves. We should not reference roles because roles are not normal objects with own responsibilities. Kristensen declares that a role only instantiates when instantiating the subject it belongs to. This condition is also given for part objects. Madsen and Møller-Pedersen model the relation of part objects more concretely. An implementation supporting part objects is BETA [9], for example, using special semantics and syntax to provide location of part objects. Nevertheless, not all programming languages provide a facility to model part objects.

2.3.2 Forces Affecting Subjects

Real world problem domains motivate the notion of subjects. Kristensen defines three forces that affect the behavior of a subject in the real world [10]. We shortly summarize how to use roles to model these influences:

- **Sender Force:** The sender of the message affects the behavior of a subject. For example, the message `addAndRecord:` makes the bank account behave subjective depending on the object that is sending the message. We can model the sender force by using different roles.
- **Context Force:** The context of the sender and receiver object influences the behavior. For example, a user being in the context of the bank building may make a bank account behave differently. Roles do not provide a facility to model context-dependent behavior, as the subjective decision is always taken by the sender.
- **State Force:** The state of the sender and receiver also influences the behavior. Kristensen proposes an extension of roles, called multiple method implementations.

Kristensen is aware of the problem that only the sender takes the subjective decision by determining the role used when sending a message. To overcome this limitation, Kristensen proposes the use of Multiple

Method Implementations [10]. The idea is to have a set of methods modeling the possible behaviors for one message. When the receiver receives a message he decides, depending on the current context, which method to invoke [11]. Chamber already proposes an implementation of multi-methods in the programming language *Cecil* [12].

The bank account, for example, can use multiple method implementations to decide the behavior for the message `addAndRecord:`. In this scenario, the only influence determining the behavior is the sender of the message. Moreover, there can also be time constraints or performance measures influencing the behavior of objects [11]. We refer to the paper for a more detailed discussion.

There are implementations of role-based programming written in BETA [13] and Smalltalk [13].

2.3.3 Role's Drawbacks

Role-based programming forces the developer to model subjective problem domains using subjects playing roles. Let us consider the group programming example [3] of a system for registering changes on source code of an object-oriented application. In the original implementation perspectives modeled the changes to the source code. Hence we can have different views of the same source-code. Using roles for this particular problem domain implies that the source code subject plays different roles for developers. Modeling the source code object using perspective suits this problem domain better than using roles.

Roles model a fixed mechanism to determine the subjective behavior of a subject. Role-based programming asserts that the sender always *knows* which role the subject plays when he sends the message. Consider a person having the role of an employee that complaints to another person having the roles of a chief and an actor. Roles cannot model the influence of the contexts *theater* or *office*, as always the sender of the message determines the role to use. The location of the subjective behavior mechanism is in the internals of the approach and we cannot adapt it to suit the problem domain.

Once the sender decides which role to use the behavior of the receiver is not further adapted. Consider the bank account use case where the users object sends the message to the bank account subject through the owner role. The role model does not provide a mechanism for the bank account to additionally influence the behavior, for example, by changing the role it plays.

2.4 Context-oriented Programming

Context-Oriented Programming (COP) refers to programming language mechanisms and techniques that support dynamic adaptation to context [14]. Costanza and Hirschfeld are the first to introduce COP [5]. The idea is to avoid having to spread context-dependent behavior all over the application. The Model-View-Controller pattern, for example, isolates the domain logic from the user interface by separation of concerns [15]. As a result we spread the behavior all over the application's source code.

2.4.1 Context as Layers

Object oriented programming languages often use if-constructs to allow context-dependent behavior. COP splits objects into a set of layers factoring out context-dependent behavior. We should not confuse layers in COP with the layers introduced in perspectives. Whereas a layer in a perspective is a view towards an object, layers in COP model context.

Each layer models the behavior for multiple objects associated to a particular context. Every definition not explicitly placed in a user-defined layer belongs to the default root layer. When an object receives a message, its behavior depends on the active layer, representing the current context.

Figure 2.7 shows the bank account split into three layers, representing three different contexts. In the sequence diagram `aBankAccount userLayer` means that the `userLayer` is active when the message is sent

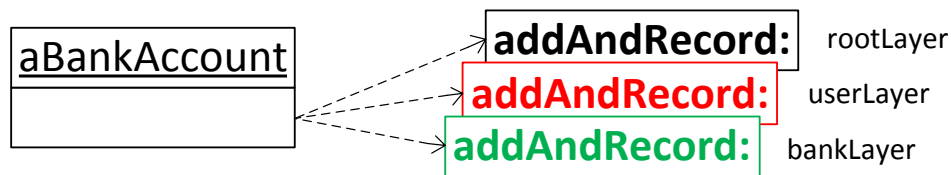


Figure 2.7: Split the behavior of the message `addAndRecord:` of the bank account object into three layers, representing three contexts. Depending in which context the message is sent, the bank account shows a different behavior.

to the `aBankAccount` object. The `userLayer`, for example, represents the context of a user sending the message to an object, in our example to the bank account. The layer implements the behavior of the bank account object for the context of a user sending the message:

```
UserLayer>>addAndRecord: aNumber
^self error: 'Rejected'.
```

The bank account is a *layered class*, because it consists of three layers. The layered class of the bank account has layered methods for the message `addAndRecord:.` The user is not a layered class, as in our use case all its behavior is objective, *i.e.*, classified in the root layer.

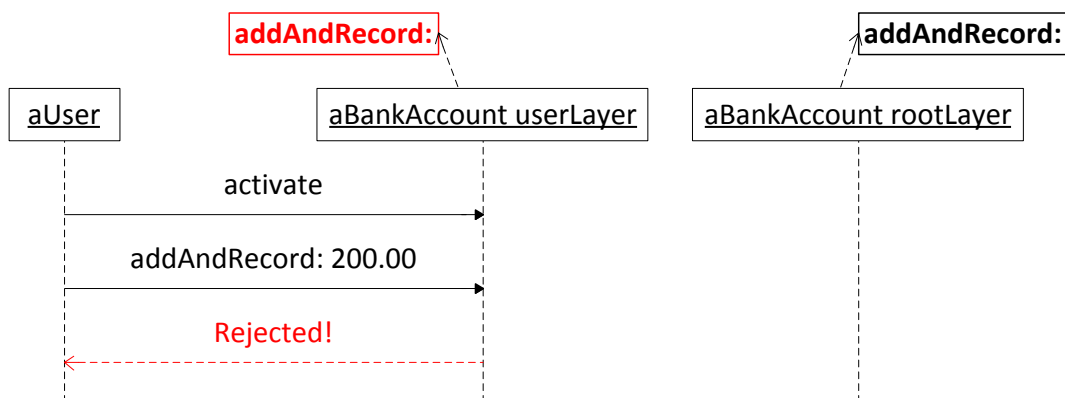


Figure 2.8: User activates context and sends message to bank account..

Figure 2.8 describes the process of the user sending the message `addAndRecord:.` Before sending the message, we choose the context by activating the corresponding layer. Before a user object sends any message he changes the context, *i.e.*, activates the layer `userLayer`. Only then the user sends the message `addAndRecord:` which has a layered method definition in the current context. As a result, the

bank account executes the behavior described by the layered method for the message `addAndRecord:` in the layer `userLayer`.

Consider that the bank account's default behavior for the message `addAndRecord:` is to return an error message. We move the layered method from the `userLayer` to the root layer of the application, because the behavior is the same as in the `userLayer`. If a user now sends the message `addAndRecord:` it first activates the user context. The active context does not define a layered method for that message. Hence, we delegate the evaluation of the message to the next layer, which is the root layer. The bank account object executes the default behavior for the message `addAndRecord:`.

The layered method for the message `addAndRecord:` in the `bankLayer`, corresponding to the context of a bank account sending the message, implemented as follows:

```
BankLayer>>addAndRecord: aNumber
self balance: (self balance + aNumber).
```

The layered method needs to access the instance variable `balance` of the bank account.

Consider a bank account sending the message `addAndRecord:` to another bank account. First, the bank account has to define the bank context as the current context, by activating the layer `bankLayer`. Next, the bank account sends the message `addAndRecord:` to the bank account, which does have a layered method definition in the active context and behaves accordingly.

2.4.2 Context Layer Activation Mechanisms

The sender of the message determines the active context by explicitly activating the layer, modeling the desired context. For example the user object explicitly activates the `userLayer` before sending the message `addAndRecord:` to the bank account object. Any object can activate the context in the COP application, also the bank account. In our particular scenario this implies the use of dispatching mechanisms as the the class of the object sending the message defines the context. To overcome this limitation, researchers defined additional approaches to deal with layer activation in COP:

Explicit layer activation. We can refer to layers at runtime and activate or deactivate them in arbitrary code locations. Furthermore, layer activation is restricted to the current thread of execution only to avoid race conditions and interferences of different contexts. COP languages and environment extensions provide the mechanisms for expressing, activating and composing layers at runtime, but it is the application domain that determines which contextual information is relevant [14].

Implicit layer activation. Layers define conditions for which they are active. The framework determines which layers are active when sending a message. For example, the `userLayer` is automatically activated if user objects send messages. *PyContext* [16] implements this extension, for example.

Reflective layer activation. Layers may have dependencies among them. The activation or deactivation of a certain layer can involve the activation or deactivation of other layers as well. Consider a user object that owns a bank account and sends a message to it. Assume that we implemented a layer representing the context of the owner sending a message to the bank account. Hence, the activation of the owner layer implies that the user layer is also activated. Reflective layer activation is useful if dependencies between the layers exist. For example, we may require the presence or mutual exclusiveness of other layers [17].

ContextL [14] extends the multi-paradigm language CommonLisp [18] with layers and PyContext [16] does the same for Python. Implementations also exist for Java, JavaScript, Smalltalk and Scheme².

²<http://www.swa.hpi.uni-potsdam.de/cop/implementations/index.html>

2.4.3 Context Variables

Implicit layer activation also proposes the use of context variables [16]. Context variables maintain their value while the layer is active. The idea originates from languages implementing dynamic scoping such as the layer activation in ContextL. Dynamically scoped layer activation has the effect that the layer is only active during execution of the contained code. When the control flow returns from the dynamically scoped layer activation it deactivates the layer again [5]. When the application activates a new context it changes value of the context variable. Only when the execution in that particular context finishes it restores the value back to the original value of the context variable.

2.4.4 COP's Drawbacks

COP forces developers to model subjective behavior as a set of contextual layers. Consider the use case where a user sends an email using a mobile device [6]. If the receiver of the email is in the same room as the sender the email becomes of high otherwise of normal priority. The mail deliverer is responsible for delivering the emails selecting the right priority depending on the context. In COP, we split the behavior for sending the message with different priorities into a two layered method, one for high and the other for normal priority. The activation of the layers depends whether or not the users are in the same room. How to activate the right layer is not foreseen in the original COP approach, which only supports explicit layer activation.

In the original understanding of COP each layer modeled a particular context. We cannot model the room as a layer which is active or not depending in which room the user is in. Hence, we redefined the understanding of the room context to suit the COP approach.

Different researchers defined their own layer activations to suit the problem domain. Adapting COP languages to support new layer activations is tedious work, because COP has a fixed view of the activation of layers. The reflective layer activation, for example, checks all layers in the system upon a layer activation, to determine which other layers to activate or deactivate.

2.5 Subjective Message Behavior

Darderes and Prieto [6] define subjective behavior as objects behaving differently for the same message depending on the current invocation context. The bank account, for example, behaves differently for the message `addAndRecord`: depending on the context of the sender of the message. Subjective method Behavior (SMB) suggests to split subjective behavior for a message into a set of independent methods. SMB proposes a generalized method dispatch mechanism to determine the behavior when an object receives the message depending on the context at invocation time.

2.5.1 Forces Influencing Behavior of Objects

The message invocation context influences the behavior of an object making it behave subjectively for the same message. Darderes and Prieto define the message invocation context as the following four *forces*:

Sender force. The sender object of a message.

Self force. The receiver of the message, *i.e.*, the behaving object.

Collaborator force. Any object known by the object receiving the message. We further categorize these into:

Internal collaborators. Objects referenced by instance variables.

External collaborators. Objects referenced by message arguments.

Class collaborators. Objects referenced by class variables.

Global collaborators. Objects referenced by global variables.

Awareness force. Any other object influencing the behavior for a message.

The above mentioned objects are called *forces*, because *force* an object to behave in a particular way. The forces are accessible at invocation time for the object receiving the message. Based on these, the object selects the appropriate method to execute.

The bank account object, for example, behaves subjectively for the message `addAndRecord:` depending on the sender force. The use case only allows bank account objects to change the balance of another bank account by sending the message `addAndRecord:.` Any other object sending the same message to a bank account has no effect on its balance. This leads to two possible behaviors for the message `addAndRecord:.`, being changing or not the balance of the bank account receiving the message. Hence, the set of possible methods of the bank account object for the message `addAndRecord:` consists of two methods.

Based on the sender force we can formulate a condition to determine the behavior for the message `addAndRecord:.` As only bank accounts can change the balance of another bank account we define the condition `recordAuthorized` as follows:

```
recordAuthorized := ((aMessageSender class) = BankAccount)
```

Where `aMessageSender` is precisely the object corresponding to the sender force.

Depending on conditions, like the `recordAuthorized`, the subjective behavior of an object is uniquely determined. Darderes and Prieto propose a response mechanism for subjective messages like `addAndRecord:` based on method dispatching.

2.5.2 Force Tree - A Decision Dispatching Mechanism

The *determinant* is a condition together with a behavior executed if the condition fulfills. We say that a determinant *prevails* if the condition is true at invocation time. The determinant with the `recordAuthorized` condition, for example, prevails if the sender of the message is a bank account.

SMB has two types of determinants depending on the associated behavior. *Method determinants* are determinants modeling behavior. The *force determinant* uses a boolean condition based on one force to decide which determinant to evaluate next. For example the message `addAndRecord:` for the bank account object has two method determinants, for the two behaviors, and one force determinant. The force determinant models the decision of which method determinant to evaluate based on the `recordAuthorized` condition. Figure 2.9 shows the determinants for the message `addAndRecord:` graphically. Looking at it, we detect that the determinants, modeling the subjective behavior, form a tree alike structure based on forces. Hence, we call the resulting tree, *force-tree*. The force determinants are the nodes and the method determinants are the leaves of the force tree. As force trees model subjective behavior, these replace the object oriented methods, modeling objective behavior.

Force trees need to fulfill three properties to always determine the subjective behavior for a message. The force tree has to be complete to determine the behavior based on the context when the message is send. Consider the force tree for the message `addAndRecord:.` There is no behavior for a bank account object sending the message `addAndRecord:` if we eliminate the method determinant `allowRecord`. The force tree also has to be acyclic, as cyclic force trees model unexpected behavior. Consider the force tree for `addAndRecord:` pointing back to the root node instead of pointing to the `denyRecord` method determinant. Hence, the bank account does not have any behavior for a user object sending `addAndRecord:` anymore. This will catch the force tree lookup in an endless loop until the user object eventually converts itself into a bank account object. The last condition for the force tree is that it has to be mutual exclusive. This is normally given as force determinants conditions are binary.

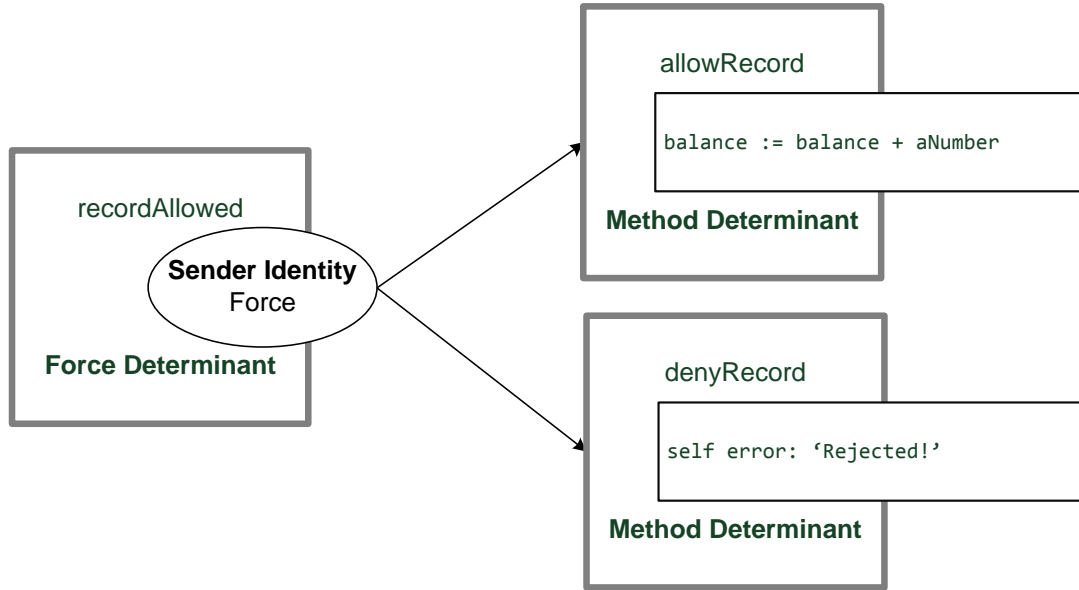


Figure 2.9: The force tree replaces the object oriented method for the message `addAndRecord:` of the bank account.

SMB also allows the modeling of more complex trees for subjective behavior than the proposed in Figure 2.9. Consider that we return different error messages to owners of the bank account than to a normal users sending the same message `addAndRecord:..` The condition for the force determinant is the following:

```
isOwner := ((aMessageSender account) = aReceiverBankAccount)
```

The `aMessageSender` object corresponds to the sender of the message and the `aReceiverBankAccount` is the bank account receiving the message `addAndRecord:..` Figure 2.10 shows the resulting force tree for this use case.

We describe how SMB reacts subjectively to a message using the `addAndRecord:` send from a user to a bank account. We use the simple force tree described in Figure 2.9. As the bank account defines a subjective behavior for the message `addAndRecord:`, the message lookup finds the root node of the force tree instead of the objective method. SMB evaluates the root node, in our example the `recordAllowed`, depending on the current context. The evaluation of a force determinant results in determining the next determinant based on a force condition. SMB evaluates the `recordAuthorized` condition based on the current context. As a user object sent the message the force determinant continues with the evaluation of the `denyRecord` method determinant. The evaluation of a method determinant results in the object behaving. The `denyRecord` method determinant models the throwing of the error message “Rejected!”.

Darderes and Prieto implemented a proof of concept of SMB in the object-oriented Smalltalk environment VisualWorks 3.0nc [6].

2.5.3 SMB’s Drawbacks

SMB forces the developer to translate subjective behavior into a force tree which does not always suit the problem domain. Consider the group programming example [3] modeling a system registering changes on

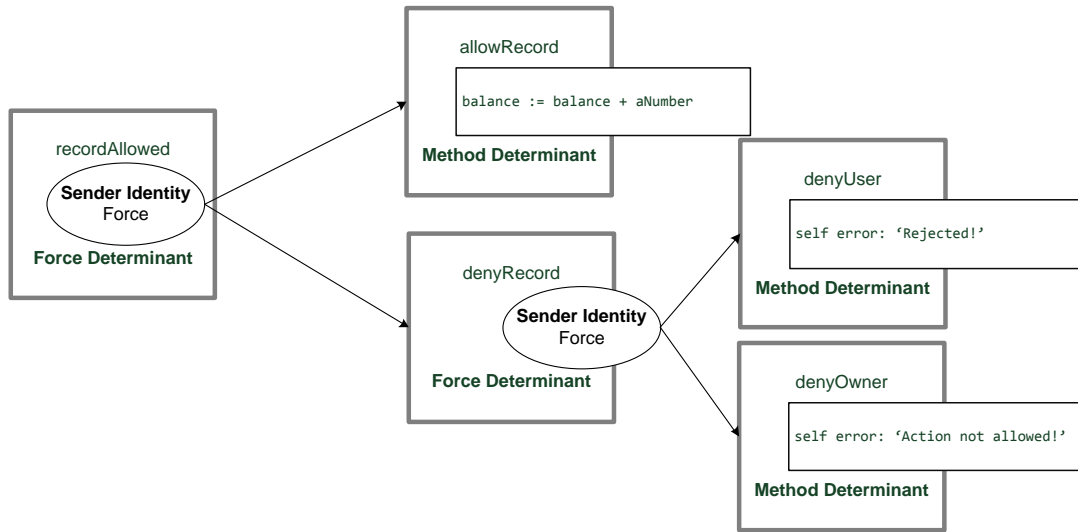


Figure 2.10: A more complex force tree replacing the object oriented method for the message `addAndRecord:` of the bank account.

source code of an object-oriented application. If we intend to view the source as a particular developer we are not evaluating a tree of changes attached to the source code. Hence, SMB does not suit this particular problem domain.

We cannot adapt the point of view of the subjective behavior in SMB. Consider a person behaving subjectively to a complaint depending on whether he is the chief or a friend of the complaining person. We cannot adapt SMB to model this problem domain using roles or perspectives, for example.

The mechanism for determining the subjective behavior is always located at the receiver of the message. The sender of a message can only implicitly, as a sender force, influence the behavior of the object receiving the message. We cannot adapt SMB to let the sender of the message decide for the subjective behavior. For example, it is not possible to implement how an object sending the message *views* the receiver.

2.6 Problems in Current Subjectivity Models

The main approaches that we have analyzed are not suitable to model all the subjective problem domains. The main drawback is that each approach forces the developer to have a fixed point of view on the problem domain. Perspectives, for example, is suitable for the group programming use case [3], whereas SMB is not.

A fixed subjective behavior model implies that the mechanism for selecting the behavior is implicit and that it is unchangeable. Roles, for example, assume that subjective behavior in all problem domains depends on the role a certain subject plays when receiving the message.

The analyzed approaches focus their mechanism for selecting the behavior either on the sender or on the receiver of the message. Because of the mechanisms being implicit we cannot adapt the location of the subjective decision to suit the problem domain. In roles, for example, only the sender of a message decides which roles the receiving subject plays. Context as an influence of the used role in the subject is not foreseen the role-based approach.

In most of the analyzed approaches only the context of the communication, being the sender, receiver and the arguments of the message, influences the behavior of the object. Nevertheless, subjective behavior may also depend on other types of *context*. Consider the email sending using a mobile device [6]. The email is sent with high priority if the sender and receiver of the email are in the same room. The mail deliverer, responsible for sending the messages with the correct priority, cannot retrieve this information from the communication context. Whereas SMB solves the problem using the awareness force, COP, for example, does not suit to model the room context. In this scenario a room is a simple list of users influencing the behavior of the mail deliverer and not a set of layers.

A common drawback of most of the analyzed approaches is that they model subjective behavior directly on objects and not subjects. The responsibilities of a subject differs from those of an object. A subject needs to be able to register those messages for which it behaves subjectively. The SMB approach, for example, needs to register which messages have a force tree.

Not all of the main approaches discuss the importance of adapting subjective behavior at runtime. Whereas objective behavior does not evolve during the lifetime of an object, subjective behavior can. A person, for example, may behave as a student at a given time and later as an employee. SMB, for example, uses a static force tree to determine the behavior of an object for a message depending on the invocation context. The implementation of SMB uses a dynamic language. Nevertheless, the approach itself does not foresee to change the force tree of an object at runtime.

With SUBJECTOPIA we provide a model for subjective behavior that does not force the developer to translate the problem domain to fit the model, since we can translate the model to fit the problem domain.

3

SUBJECTOPIA

The problem of previous approaches is that they fix the subjective point of view. For some subjective problem domains layers are more suitable than roles. In other subjective problem domains roles are more suitable than layers, as discussed in the previous section. The SUBJECTOPIA approach provides a subjective view towards subjectivity. The SUBJECTOPIA approach allows the developer to select the subjectivity model that best fits the current problem domain.

The SUBJECTOPIA approach unifies the previous approaches and provides a model that has no fixed view of what subjectivity is and how to model it. We define the SUBJECTOPIA approach as an *Adaptive Subjectivity Model*, since we can adapt the model for the subjective behavior to the problem domain. We adapt the model to suit the problem domain instead of redefining the problem to suit the subjective behavior model. To this end, SUBJECTOPIA can model perspectives, roles, COP and SMB. We can use perspectives, for example, to model the group programming example [3]. But we can also use SMB to model the mail deliverer in the email through mobile device application [6]. Moreover, in SUBJECTOPIA we can even define our own subjectivity models.

In this section we introduce SUBJECTOPIA¹ and the idea of a unifying subjective behavior model. We implemented both SUBJECTOPIA and the examples presented in this thesis in Pharo Smalltalk².

To this end, SUBJECTOPIA models objects with subjective behavior explicitly as *subjects*. Besides emphasizing the difference to common objects, subjects also have their own responsibilities, for example, registering subjective messages. When a subject receives a message, it needs to determine the behavior from a set of possible behaviors, depending on the current context. We call the process of deciding for the behavior *subjective decision*. *Decision strategies* explicitly model the subjective decision taking mechanism. We can change the subjectivity model to suit the problem domain, because explicit decision strategies allow us to reify the decision. SUBJECTOPIA also introduces *contextual elements* to model context-dependent information such as the invocation context. Contextual element are generic elements to model context that influences the subjective decision and thus the behavior of a subject.

Because of the explicitly modeled decision strategy, subjects and contextual elements, SUBJECTOPIA does not force the developer to use a fixed modeling paradigm. SUBJECTOPIA models roles, context-oriented programming, subjective message behavior and any other subjectivity models. We can experiment

¹<http://scg.unibe.ch/research/subjectopia/>

²<http://www.pharo-project.org/>

with different subjectivity models in the same application.

To explain the SUBJECTOPIA model we use the canonical bank account example from the perspective approach, introduced in Chapter 2.

3.1 Subjects

A *subject* is an object that behaves differently under different contextual circumstances in contrast to objects that always behave objectively. SUBJECTOPIA models subjects explicitly because they have their own responsibilities. A subject may be fully subjective or only present subjective behavior for certain messages. A bank account subject, for example, may behave subjectively for the message `addAndRecord:` and objectively for the message `balance:`. Subjects have the following main responsibilities:

1. Register messages that expect subjective behavior.
2. Evaluate the subjective decision when receiving a messages expecting subjective behavior based on the contextual information.
3. Allow runtime adaptation of the subjective decision for a message.

In SUBJECTOPIA we have two possibilities to define a subject. We can transform a regular object into a subject at runtime by sending the message `becomeSubject` to the object. For example, we can tell the bank account object `aBankAccount` to become a subject by sending `becomeSubject` to it:

```
aBankAccount becomeSubject.
```

The transformation of the `aBankAccount` into a subject adds the necessary behavior to the bank account to behave subjectively for certain messages. The second way is to define a subject by directly subclassing the class `Subject`. The bank account object, for example, can directly inherit from the class `Subject`, having the same effect as sending the message `becomeSubject` to a bank account object.

A subject must be able to register subjective messages by sending the message `register:for:` to a subject. The bank account subject, for example, defines subjective behavior for the message `addAndRecord:`. We call messages that have subjective behavior *subjective messages*. To make the bank account subject behave subjectively for the message `addAndRecord:` we send the following message to it:

```
aBankAccount register: aDecisionStrategy for: #addAndRecord:.
```

The registration process of a subjective message assigns a decision strategy to the message which models the subjective decision process. If the subject receives a subjective message it evaluates the decision strategy. The registration process also replaces any old behavior for the subjective message of the object. Consider the `aBankAccount` having defined an objective method for the message `addAndRecord:`. Hence, the registration process replaces the bank account's original behavior for the message `addAndRecord:` by the decision strategy. The message `register:for:` also replaces subjective behavior allowing a subject to adapt its subjective decision process at runtime.

The class `Subject` models objects that behave subjectively for certain messages. Explicitly modeling subjects also allows us to detect and reflect upon the subjective parts of an application. Other approaches, like COP, encode this information in the internals of the application's source code, requiring the use of *ad hoc* mechanisms to detect subjective behavior.

3.2 Decision Strategies

SUBJECTOPIA uses *decision strategies* to model the process of deciding how a subject behaves when it receives a subjective message. SUBJECTOPIA aims at eliminating fixed points of view towards subjective behavior.

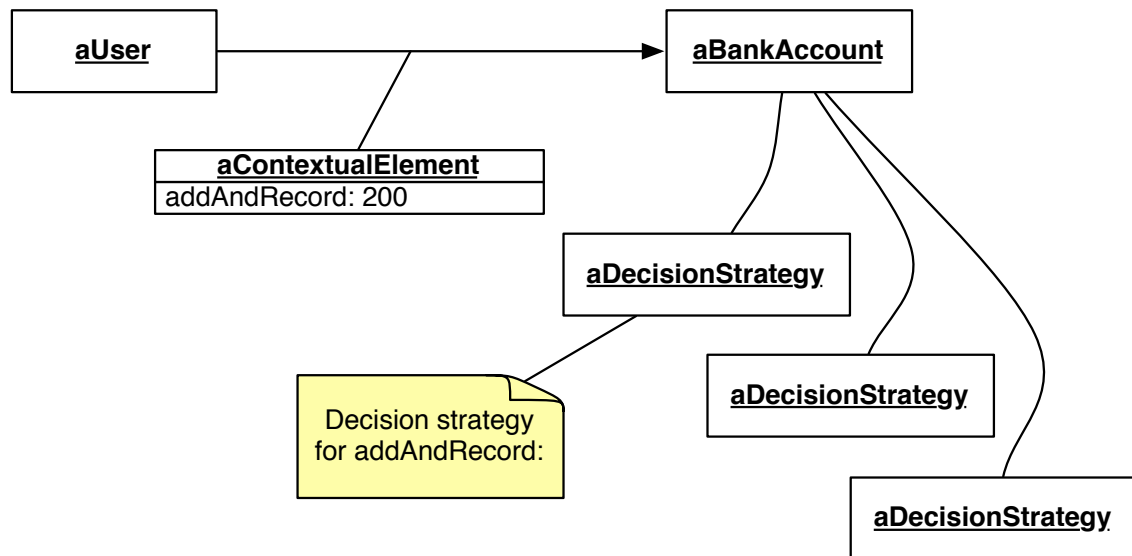


Figure 3.1: The object `aUser` sends message `addAndRecord:` with argument 200.00 to `aBankAccount`. The subject performs a lookup and finds the subjective method. The method evaluates the decision strategy selecting the appropriate behavior for the current context.

Consider that we have already defined the message `addAndRecord:` to be subjective for the bank account subject. The moment we send the message `register:for:` to the bank account subject the method for the message `addAndRecord:` adapts to:

```

aBankAccount>>addAndRecord: aNumber
| message |
message := self currentMessageContext.
^(self findDecisionStrategyFor: #addAndRecord: evaluate: message)

```

The adaptation forces the object to evaluate the decision strategy instead of executing behavior when he receives the message `addAndRecord:`. Since this is still a normal method the subject performs a traditional method-lookup when it receives the subjective message.

Figure 3.1 shows the process of the subject `aBankAccount` receiving the message `addAndRecord:` from `aUser`. Subjective methods use two steps to make subjects behave subjectively. The first step consists in the subject creating a contextual element representing the meta-information of the invocation context. In Figure 3.1 the `aContextualElement` object models the invocation context, containing the following information:

- The message selector `#addAndRecord:`.
- The arguments as a list, *i.e.*, a list with the entry 200.00.
- The sender of the message, *i.e.*, the `aUser` object.
- The receiver of the message, *i.e.*, the `aBankAccount` subject.

The meta-information of the message send represents the context of the communication. Hence, we model the invocation context as a contextual element. We describe the modeling of context in more detail in Section 3.3.

The second step consists in evaluating the decision strategy with the contextual information modeling the context of the communication. We can resolve the evaluation of the decision strategy in three different ways:

- Executing behavior, if the decision strategy directly models behavior.
- Delegating to another decision strategy for further evaluation allowing us to model decision hierarchies.
- Sending a message to the subject if we model all possible behaviors in the class of the subject.

The decision strategy models the subjective decision or behavior by overriding the method `decideOn:`. Subsequently, we model exemplifying decision strategies for all the three ways of evaluating decision strategies. To this end we use the bank account use case.

Consider a simple decision strategy called `SimpleDecisionStrategy` that directly models the behavior for the message `addAndRecord:`. To this end, the decision strategy defines the behavior for the message `decideOn:` as follows:

```
SimpleDecisionStrategy>>decideOn: aMessageContext
((aMessageContext sender class) = BankAccount)
  ifTrue:
    [(aMessageContext receiver) balance: (balance + ((aMessageContext arguments)
at: 1))]
  ifFalse:
    [self error: 'Rejected']
```

The decision strategy checks the class of the sender of the message and reacts accordingly using an if-else construct. The `aMessageContext` object models the contextual element representing the meta-information of the message. The meta-information includes all the relevant communication context and makes it available to the decision strategy. Sending the message `sender` to the `aMessageContext` object returns the sender object of the subjective message. In the scenario described in Figure 3.1, for example, sending `sender` to the `aMessageContext` object returns `aUser`. Sending the messages `receiver` and `arguments` to the meta-information object, return the receiver and the list of arguments of the subjective message, respectively. Decision strategies directly modeling behavior, like the `SimpleDecisionStrategy`, are normally used in hierarchies of decision strategies. The proposed `SimpleDecisionStrategy` has only explanatory value.

Next, we model the decision strategy `TreeDecisionStrategy` composed of multiple decision strategies. The subjective message `addAndRecord:` has two different behaviors, we directly model in two independent decision strategies. The `UserBehavior` decision strategy models the behavior if the sender of the message `addAndRecord:` is a user object. The `BankAccountBehavior` decision strategy does the same for bank account subjects. The `TreeDecisionStrategy` is responsible for determining which decision strategy to evaluate next. For this purpose we require the sender of the message. The method `decideOn:` in the `TreeDecisionStrategy` looks like:

```
TreeDecisionStrategy>>decideOn: aMessageContext
((aMessageContext sender class) = BankAccount)
  ifTrue:
    [BankAccountBehavior new decideOn: aMessageContext]
  ifFalse:
    [UserBehavior new decideOn: aMessageContext]
```

The `TreeDecisionStrategy` models the subjective decision as a tree where the true node is the `BankAccountBehavior` and the false node the `UserBehavior`. The class of the sender of the message determines the condition. We can observe the similarities of this decision strategy to SMB. We treat decision strategies used to model SMB in Section 3.4.4.

SUBJECTOPIA allows us to polymorphically change decision strategies. Consider that we first model the subjective behavior using the `SimpleDecisionStrategy` and then decide to use the `TreeDecisionStrategy`. We can replace the `SimpleDecisionStrategy` by the `TreeDecisionStrategy` by sending the message `register: TreeDecisionStrategyfor: #addAndRecord` to the bank account subject. As a consequence we can experiment with different subjectivity models to find the one that suits the problem domain the best.

Last we model a decision strategy that sends a message back to the subject receiving the message. To this end we change the `TreeDecisionStrategy` and implement the two different behaviors directly in the subject. The `decideOn:` method of the `TreeDecisionStrategy` looks like:

```
TreeDecisionStrategy>>decideOn: aMessageContext
((aMessageContext sender class) = BankAccount)
  ifTrue:
    [(aMessageContext receiver) changeBalanceBy: ((aMessageContext arguments) at:
1)]
  ifFalse:
    [[(aMessageContext receiver) alertIllegalAction]
```

In contrast to the original `TreeDecisionStrategy` we do not model the behavior in decision strategies, but directly on the subject. The subjective decision concludes in the sending of a message to the bank account subject. In this scenario the bank account subject is the receiver of the message `addAndRecord:`. To access the bank account subject we send the message `receiver` to the contextual element representing the context of the communication.

Decision strategies can be replaced. Hence, we can adapt the paradigm for modeling subjective behavior over time. This allows us to experiment with different subjectivity models to find the model that suits the problem domain the best. We can use decision strategies to model perspectives, roles, COP and SMB, discussed in Section 3.4. Because we can register any decision strategy for a subjective message we can use different subjectivity models within the same application. We can use a role-based decision strategy for one message and a perspective-based decision strategy for another, for example. As a consequence the developer using SUBJECTOPIA is not forced to translate the problem domain to suit the model, but he can adapt the subjectivity model to fit the problem domain.

3.3 Contextual Elements

SUBJECTOPIA introduces *contextual elements* to provide a general model for contextual information. A special contextual element is the object reifying the meta-information of the communication discussed in Section 3.2. As well as the communication context contextual elements are available for the decision strategy.

Some applications need to model additional context not retrievable from the context of the communication. We can use contextual elements to model these kind of contextual informations. Consider the use case where users send emails from their mobile devices [6]. If the sender and receiver of the email are in the same room the mail deliverer sends the emails with high priority. We cannot determine the room context from the context of the communication. We can use a contextual element to model the additional information and influence the subjective decision process taken by the decision strategy. We can model the room context, for example, as a contextual element holding a list of all present users in a room.

The decision strategy determines how the contextual element influences the subjective decision. The decision strategy can access, evaluate or ignore the contextual elements. The evaluation of a contextual element by a decision strategy results in one of the following:

- Provide contextual information to the decision strategy.
- Execute behavior.

- Delegate to another contextual element for further evaluation. We call such contextual elements *composed contextual elements*.

Next, we give examples of how decision strategies evaluate each of the types of contextual elements. Then, we discuss how contextual elements influence subjective behavior in more detail.

The meta-information of the communication, for example, is a contextual element that represents the contextual information of the communication. It is directly accessed and influences the subjective decision of the decision strategy. We can model the above presented room context for the sending of emails using mobile devices use case as a contextual element. The contextual element holds contextual information evaluated by the decision strategy. The decision strategy cannot retrieve the room context from the communication context.

Contextual elements are also used to model behavior. We call behavior extrinsic to an object *subjective behavior*. Contextual elements can also model subjective behavior. Some subjectivity models require additional abstractions affecting the subjective decision or behavior of the decision strategy. The role based approach, for example, requires the abstraction of *roles*. Nevertheless, these are not retrievable from the communication context. A contextual element modeling a role, directly implements behavior. We discuss the contextual elements of each of the subjectivity models in Section 3.4. Other approaches ignore contextual elements. Consider a decision strategy modeling SMB. The decision strategy ignores contextual elements modeling roles, for example, as these are not used by the SMB approach. The decision strategy models how a contextual element influences the subjective decision.

Some problem domains or subjectivity models require composed contextual elements. An example for composed contextual elements are the perspectives in the perspective-based subjectivity model. One contextual element models one layer of the perspective. The layers as contextual elements are hierarchically composed to one perspective. The decision strategy then determines the evaluation order of the composed contextual elements.

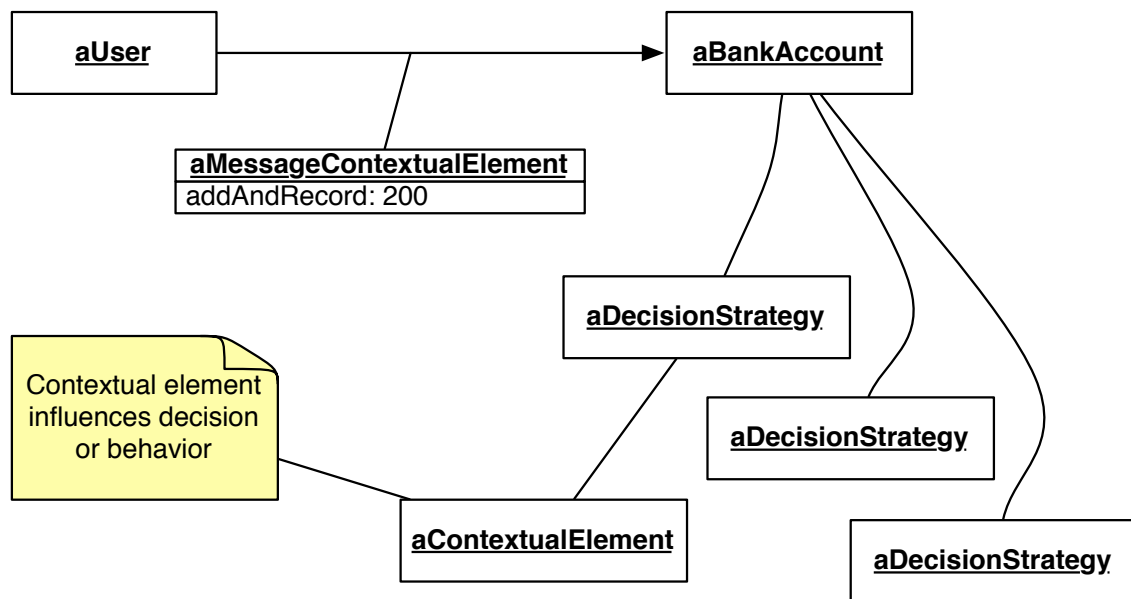


Figure 3.2: Two ways of using contextual elements.

Figure 3.2 describes how a decision strategy uses contextual elements to influence the subjective behavior decision. Decision strategies can access contextual elements in two ways: either the decision

strategy has a reference to it or is sent together with the message.

We use the bank account example based on decision strategies modeling perspectives to describe the use of contextual elements send together with the message. As the sender of the message decides which perspective to use, we say: “The sender sends the message through a perspective”. In Figure 3.2 a bank account subject sends the message `addAndRecord:` through the perspective, modeled as `aMessageContextualElement`. Perspectives modeled as contextual elements contain pieces that model behavior for messages. We describe perspectives as decision strategies in more detail in Section 3.4.1. We registered the message `addAndRecord:` as a subjective message. This adapted the method behavior to allow the use of contextual elements:

```
aBankAccount>>addAndRecord: aNumber through: aContextualElement
| message |
message := self currentMessageContext.
^(self findDecisionStrategyFor: #addAndRecord: evaluate: message with:
aContextualElement)
```

When the bank account receives the message, it evaluates the decision strategy, which delegates the subjective decision to the perspective contextual element. The perspective contains a layer that models a piece for the message `addAndRecord:`. The decision strategy executes the piece and makes the bank account behave subjectively depending on the used perspective.

Consider the use case where users send emails from their mobile devices [6]. We use it to talk about decision strategies directly accessing contextual elements. We model the room context of present users as a contextual element implemented as a list of users. The decision strategy of the mail deliverer has a reference to the contextual element modeling the room it is in. When the mail deliverer receives the email the decision strategy takes the room contextual element to influence the subjective decision. Based on the information provided by the room contextual element the decision strategy sends the email with low or high priority. In Figure 3.2 this corresponds to the contextual element which is directly accessed by the decision strategy.

Contextual elements have no fixed point of view about contextual information. We can use contextual elements to model context without having to translate the problem domain to suit the model. We can model rooms, for example, as a list of present users or, if required, as an object that also includes the temperature of the room. We are not limited to use contextual elements to only provide contextual information, but also to provide contextual behavior. The pieces in the perspective based approach, for example, are contextual elements that model behavior for a message and object. Contextual elements also model additional abstractions such as layers in COP.

3.4 Modeling Previous Approaches in SUBJECTOPIA

In this section we introduce decision strategies modeling perspectives, roles, COP and SMB. This shows that SUBJECTOPIA allows us to easily adapt the subjectivity model to suit the problem domain.

3.4.1 Perspectives as Decision Strategy

In some problem domains, the behavior of an object depends on how the object sending the message *views* the receiving object. The behavior of a bank account subject, for example, depends how the object or subject sending the message *views* the bank account subject. For such problem domains SUBJECTOPIA provides decision strategies modeling perspectives.

Perspectives are external to the context of the communication and thus modeled as contextual elements. We model the layers of the perspective as contextual elements as well. Each perspective points to its root layer. A layer contains pieces that implement behavior for a message and object. We model pieces

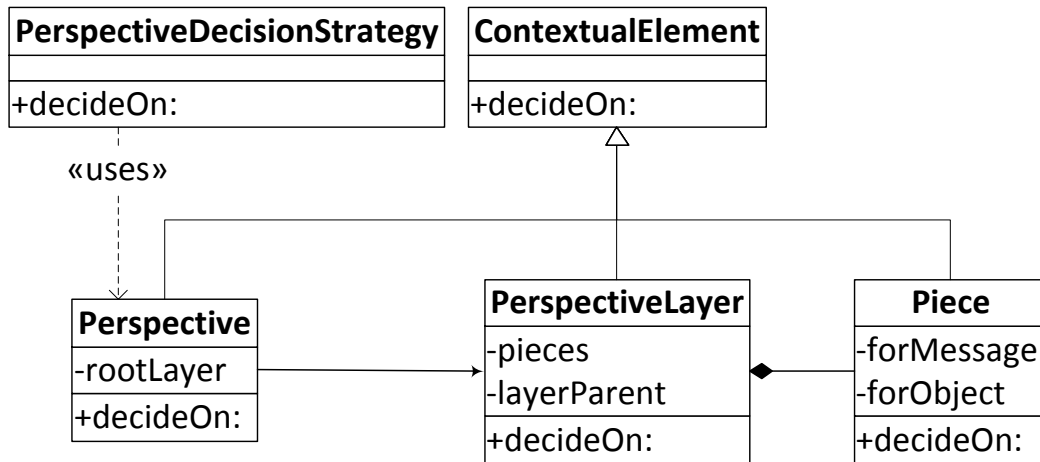


Figure 3.3: Perspectives as decision strategy

as contextual element and override the method for the message `decideOn:`. The piece that models the behavior for the message `addAndRecord:` of bank account subjects for users sending the message, for example, overrides `decideOn:` as follows:

```

UserPiece>>decideOn: aMessageContext
  ^self error: 'Rejected'.
  
```

We sent the contextual element modeling the perspective together with the message. A user, for example, sends `addAndRecord: 200.00` through: `aUserPerspective` to the bank account subject. The decision strategy of the perspective based approach, used to model the subjective decision, does the following:

1. Start at the root layer of the perspective.
2. Find the piece in the layer corresponding to the message selector and the receiver of the message.
3. Send the message `decideOn:` to the corresponding piece.

In the perspective based approach the sender of the message takes the subjective decision by choosing the used perspective. The decision strategy delegates the subjective decision entirely to the perspective.

3.4.2 Roles as Decision Strategy

In some problem domains subjective behavior depends on which role the subject plays when receiving the message. For example, the bank account subject plays a different role for user objects than for bank account subjects. To this end SUBJECTOPIA provides decision strategies modeling roles.

We model roles as contextual elements because these are external to the context of the communication. Roles are composed contextual elements to allow hierarchically ordered roles. We model a slightly different version of roles in SUBJECTOPIA than originally proposed by Kristensen. A role defines methods with the following preambles:

methodAdd. Adds a behavior for an unknown message to the subject.

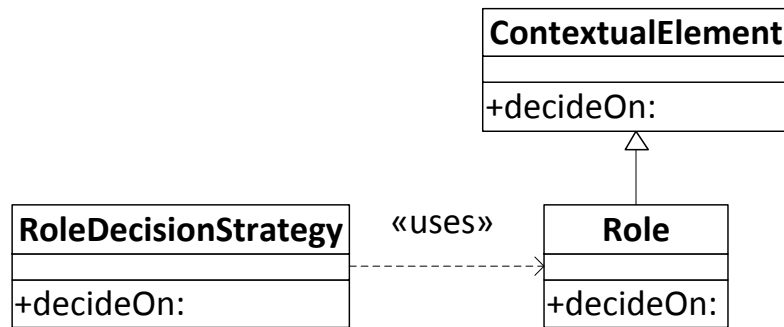


Figure 3.4: Roles as decision strategy

methodChange. Changes the behavior of a message on the subject.

methodHide. Hides the behavior for the message defined on the subject.

For example, modeling the method named `methodChangeAddAndRecord:`, replaces the original behavior of the bank account subject by the one we modeled in the role.

As well as the contextual elements modeling perspectives we send the role contextual element together with the message. The decision strategy searches the corresponding method in the composed contextual elements modeling the role. The subject then behaves accordingly to the preamble the method in the role has.

Roles model the subjective decision at the sender side by choosing which the role played by the subject. The decision strategy modeling roles, entirely delegates the responsibility to the role objects, modeled as a contextual elements.

3.4.3 COP as Decision Strategy

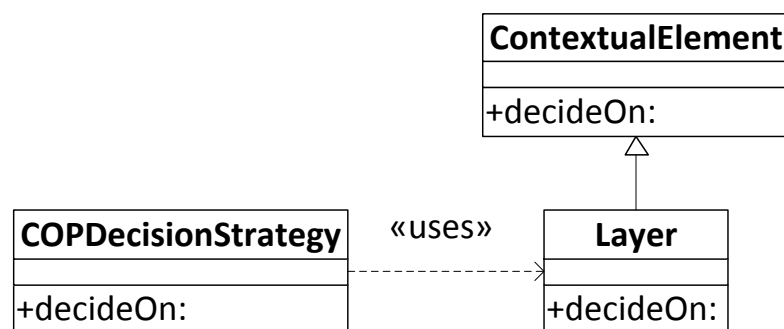


Figure 3.5: COP as decision strategy

In some problem domains subjective behavior depends on different contexts. Depending on which context is active at invocation time, the subject behaves differently. For example, the bank account subject behaves differently in the user context than in the bank account context. To this end SUBJECTOPIA provides decision strategies modeling COP.

In SUBJECTOPIA, COP uses one main system wide decision strategy for all subjects modeled in COP. The decision strategy modeling COP replaces the default decision meta-object of the subjects. We introduce decision meta-objects in more detail in Section 3.5. We model the layers as contextual elements because they are extrinsic to the communication context. The root layer defines the default behavior, *i.e.*, the default context, for all subjects. We can add others representing specific contexts. The active context decides the behavior of the subject for the message.

We define context-dependent behavior directly on the layers. A layer corresponds to a particular subject. We split the behavior of the bank account subject, for example, into two layers: a user and a bank account layer. The layers correspond to the context of a user or a bank account sending the message `addAndRecord:.`

In SUBJECTOPIA the selector of the method in the layer contextual element is a concatenation of the class name of the subject for which we defined the behavior and the message selector. Each layer in the bank account application defines a method for the message `BankAccountAddAndRecord:.` The selector is the concatenation of the class name `BankAccount` and the message selector `addAndRecord:.`

The decision strategy of COP finds, using the contextual information of the communication, the corresponding method in the contextual element modeling the active layer and behaves accordingly.

We activate the layer explicitly in the system wide decision strategy of COP. The decision strategy modeling COP entirely delegates the behavior to the active layer contextual element.

3.4.4 SMB as Decision Strategy

SUBJECTOPIA also models force trees for problem domains where this subjectivity model suits the best. For example, the bank account object decides how to behave for the message `addAndRecord:` depending on the sender force.

We model the force tree determinants as decision strategies. We have two type of determinants, method determinants, modeling behavior, and force determinants, modeling the decision of which determinant to evaluate next. The force determinant depends on the force condition. Forces are retrievable from the context of the communication. Apart from the awareness forces modeled as an additional contextual elements. In SMB the subjective decision consist of the following steps:

1. Evaluate the root node of the force tree.
2. If the node is a force determinant use the communication context to evaluate the force condition. Depending on the result evaluate the `trueDeterminant` or `falseDeterminant`. We repeat this step until the evaluated determinant is a method determinant.
3. If the node is a method determinant, execute the modeled behavior.

The receiver of the message takes the subjective decision in SMB by evaluating the force tree. In SMB the decision strategy directly models the subjective decision and behavior.

3.5 Implementation

We implemented the SUBJECTOPIA approach in Pharo Smalltalk, because of its advanced support for run-time reflection. In SUBJECTOPIA we have two ways to define a subject. First, a subject can directly inherit from the class `Subject` to be able to register subjective behavior. Second, we can send the message

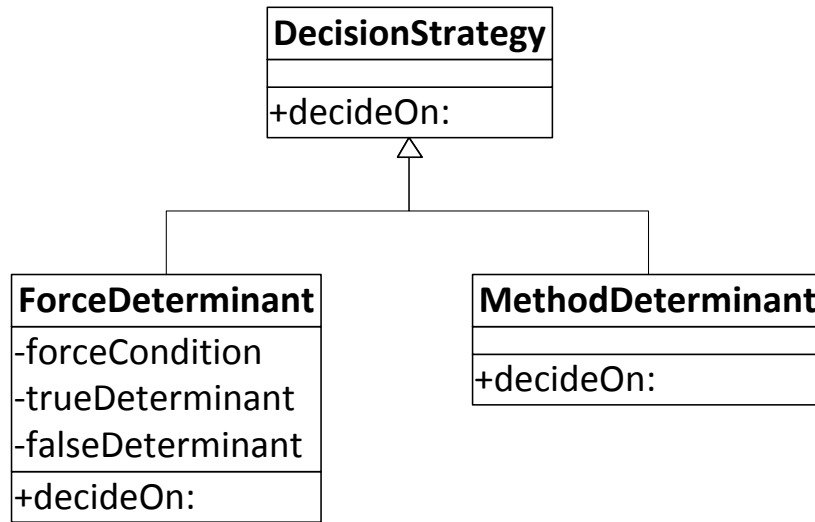


Figure 3.6: Subjective message behavior as decision strategy

`becomeSubject` to any object, to convert it to a subject. Sending the message `becomeSubject` automatically adds the methods to the class of the object for it to behave subjectively. This process converts every instance of that class into a subject as well. In Section 5.1 we propose a more dynamic way of converting objects into subjects. The proposed extension allows us to convert a single object into a subject without affecting its class. Hence, only that particular transformed object defines subjective behavior.

Each subject has a special decision strategy, called *decision meta-object*, which maps subjective message selectors to decision strategies.

Registering a subjective method by sending `register:for:` to the subject consists of two steps. First, it creates an entry in the decision meta-object with the message as key and the decision strategy as value. Second, it adapts the behavior of the registered method. Instead of performing the original behavior the method collaborates with the subject's decision meta-object to evaluate the corresponding decision strategy.

For example, to model subjective behavior on a bank account object we first send the message `becomeSubject` to the object. Second, we define the message `addAndRecord:` to be subjective by sending the message `register: aDecisionStrategy for: #addAndRecord:` to the bank account subject. Consequently, the subject creates an entry with key `addAndRecord:` and value `aDecisionStrategy` in its decision meta-object. Also the following method is automatically generated in the subject:

```

aBankAccount>>addAndRecord: aNumber
  ^self findDecisionStrategyFor: #addAndRecord: evaluate: thisContext.
  
```

The object `thisContext` is a pseudo variable representing the current context of the method execution. Smalltalk automatically generates the `thisContext` variable [7].

Figure 3.8 shows the sequence diagram of a user object sending the message `addAndRecord:` to a bank

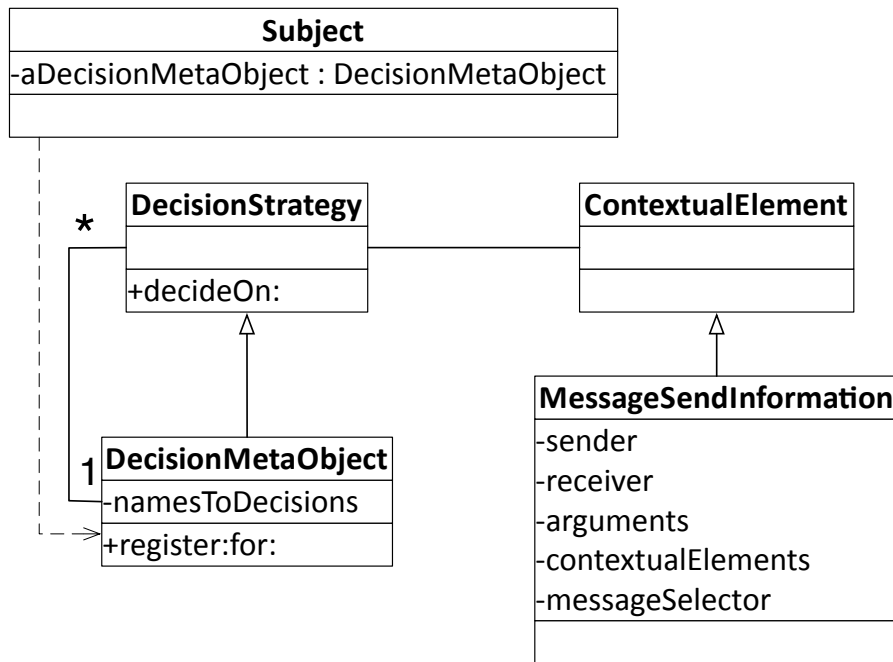


Figure 3.7: Class diagram of SUBJECTOPIA.

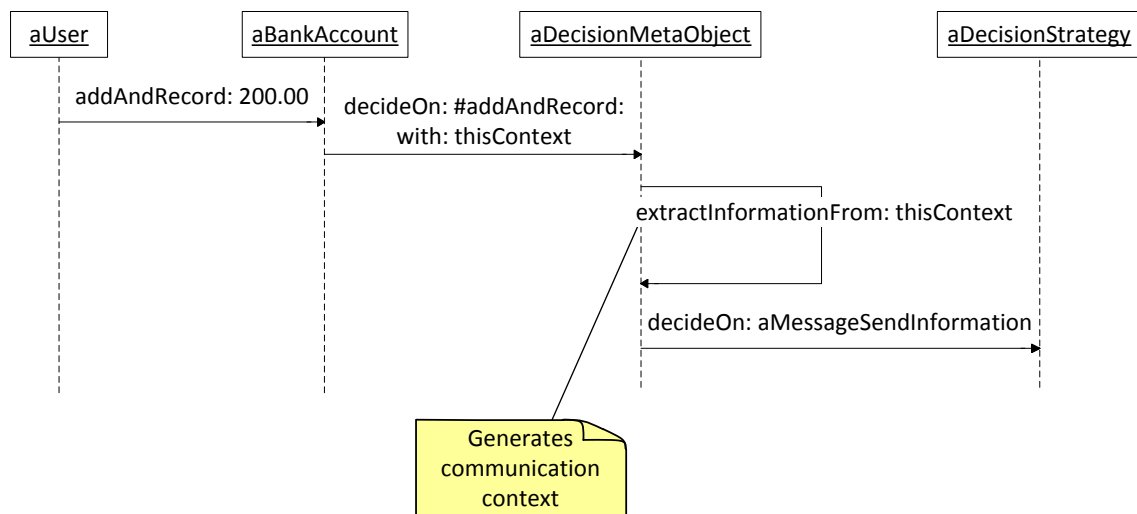


Figure 3.8: Evaluation of a decision meta object in SUBJECTOPIA.

account subject. First, the bank account delegates the search for the decision strategy corresponding to the message selector `addAndRecord:` to the decision meta-object by sending the message `decideOn:with:`.

Next, the decision meta object generates the meta-information of the communication by sending the message `extractInformationFrom:` using the `thisContext` variable of Smalltalk. In SUBJECTOPIA we call the meta-information of the message `MessageSendInformation`. It contains the following information:

- Sender of the message.
- Receiver of the message, thus the subject itself.
- Message selector.
- Arguments of the message.
- Contextual elements that are send together with the message.

Finally, SUBJECTOPIA evaluates the decision strategy by sending the message `decideOn:` and passing along the current communication context `MessageSendInformation` as a contextual element. The evaluation of the decision strategy determines the subjective behavior for the message `addAndRecord:`.

SUBJECTOPIA allows the sender of any subjective message to add `through:` to send a contextual element together with it. Since we are in the context of Smalltalk we solved this by overriding `doesNotUnderstand:` in the `Subject` class. The `doesNotUnderstand:` method looks for the decision strategy corresponding to the message selector without `through:`. Then it evaluates the decision strategy sending the contextual element together with the message send information. It is possible to implement a solution in other languages as well even if it requires modifications to the virtual machine or the compiler. Figure 3.9 describes the `doesNotUnderstand:` process.

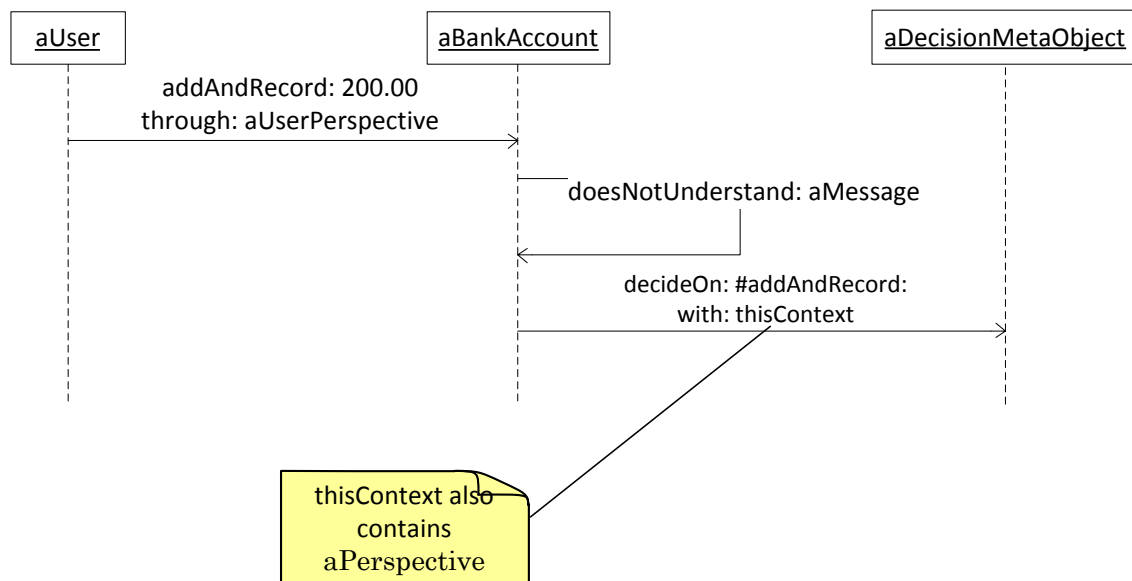


Figure 3.9: Sending contextual elements together with messages in SUBJECTOPIA.

Consider an object that sends the message `addAndRecord: 200.00 through: aBankAccountPerspective` to a bank account subject. In Smalltalk a missing method for a message concludes in the evaluation of the method `doesNotUnderstand:`. The bank account subject does not

define a method for the message `addAndRecord:through:.` Hence, the bank account evaluates the `doesNotUnderstand:` method of the class `Subject`. The subject performs a decision lookup to get the decision strategy for the message `addAndRecord:` from the decision meta-object. The object representing the invocation context includes the contextual element `aBankAccountPerspective`. The decision strategy can take the contextual element `aBankAccountPerspective` into consideration available through the meta-information object.

In SUBJECTOPIA we can polymorphically change the root decision strategy, modeled as decision meta-object, by any other decision meta-object. For this purpose we sent the message `decisionMetaObject:` to the subject. We can replace the decision meta-object to use COP by sending:

```
[...]
^aBankAccountSubject decisionMetaObject: (COP new).
[...]
```

4

Validation

In this section we demonstrate how to use SUBJECTOPIA to solve five particular use cases. In contrast to the previous subjectivity models we can select the best suiting subjectivity model in SUBJECTOPIA to solve the problem domain. We discuss for each use case why current subjectivity models cannot solve the subjective problem domain properly. As we can adapt the subjectivity model in SUBJECTOPIA we can also decide the location of the subjective decision. In SUBJECTOPIA we can use the SMB model for problem domains where the receiver of the message takes the subjective decision. Perspectives in SUBJECTOPIA may model other problem domains where the sender of the message takes the subjective decision.

In this section we extracted only two use cases from the discussion of the previous approaches. Instead of solving general subjectivity problem domains we focus on use cases preferring a particular subjectivity model. We defined three use cases from subjectivity requirements taken from the *Moose* platform for software and data analysis¹. We implemented both SUBJECTOPIA and Moose in Smalltalk. We use the Moose use cases to discuss how SUBJECTOPIA operates in legacy systems.

We briefly explain the use cases focusing on the subjective behavior and point to the source we extracted them from. SUBJECTOPIA models all of the discussed subjectivity models, perspectives, roles, COP and SMB. The use cases demonstrate why the selection of a subjectivity model is subjective to the problem domain.

4.1 Case Study – Mobile Mail Application

Let us consider the mobile mail application introduced by Darderes and Prieto [6]. The use case is about having users sending emails from their mobile devices. A user can only send emails from his own device. The user collaborates with a mail deliverer to send emails to other users. The sending of the email is only further processed by the mail deliverer if the user sending the email is the owner of the device. The mail deliverer has an internal collaborator representing the network to which the user sends the email to. The evaluation order of the emails on the network depends on the priority of the emails. The mail deliverer determines the priority of the email depending on the physical location of the sender and the receiver of

¹<http://www.moosetechnology.org>

the email. If location of the sender and receiver of the emails is the same room the mail deliverer sends the email with high else with normal priority.

In our discussion we focus on two subjective behaviors of the mail deliverer. First, the mail deliverer behaves subjectively when a user sends an email depending on the user-device context. The behavior of delivering the email to the network depends if the user owns the device or not. Second, the mail deliverer behaves subjectively depending on the room context. The mail deliverer sends the emails to the network with different priorities depending on the room context. We locate all the behavior at the mail deliverer subject when sending an email using the message `deliver::`.

The behavior of sending the email over a mobile device depends on the following conditions:

- Only authorized users can send emails *i.e.*, only the owner of the device. Otherwise the user gets an error message.
- The sender and receiver being in the same room makes the email be sent with high priority. Otherwise with normal priority.

In SUBJECTOPIA we followed the original implementation of the use case modeling the subjective behavior using force trees. Figure 4.1 describes the classes and the sequence diagram used in the original implementation of the mobile mail application based on SMB. The class `Device` has an instance variable that points to its owner being an object of class `User`. The user object sends the message `deliver::` to the `aMailDeliverer` object. We model the subjective behavior of the message `deliver::` as a force tree. To model the authorization process we use the `authorizedSender` condition based on the sender force:

```
authorizedSender := (aSender = device owner)
```

The `aSender` object refers to the object that sends the message `deliver::`. The device is an object corresponding to the awareness force. The red path in Figure 4.1 corresponds to the scenario where the user is not logged in.

The mail deliverer object has an internal collaborator representing the network modeled as an object of class `Network`. The `BoardedContext` models the room context with a list of users that are present in the room. We have to be aware that the mobile device may not have enough signal strength to connect to the network. The network object stores the delivery until establishing a connection and then sends the emails according to its priorities. The mail deliverer uses the awareness force to determine the second subjective behavior regarding the priority when sending the email. We define the priority condition based on the awareness force, called `IsImmediateDeliver`, as follows:

```
IsImmediateDeliver := (boardedContext includes: aMail recipient)
```

The boarded context, representing the room context holds a list of users that are present in the room. The second subjective decision correspond to the blue and green path in Figure 4.1. Depending on the `IsImmediateDeliver` the mail is send with high or normal priority.

Figure 4.2 describes the resulting force tree. The evaluation of the force determinant determines, based on the sender identity using the `authorizedSender` condition, the next determinant. If the condition is false, the force tree evaluates the method determinant `notAuthorizedSender`, notifying the user that he is not logged in. Otherwise the force tree evaluates the force determinant `IsImmediateDeliver` based on the `IsImmediateDeliver` condition. Depending on the condition, the mail deliverer sends the email with high or normal priority to the network object.

We discuss the differences of the original SMB implementation and SMB in SUBJECTOPIA. In SUBJECTOPIA we model objects with subjective behavior as `Subjects`. Hence, the mail deliverer is a subject due to its subjective behavior for the message `deliver::`. SUBJECTOPIA uses contextual elements to model contextual information. As the awareness forces model contextual information not retrievable from the communication context we model them as contextual elements. We can model the boarded context, for example, as a contextual element used to influence the subjective decision made by the decision strategy.

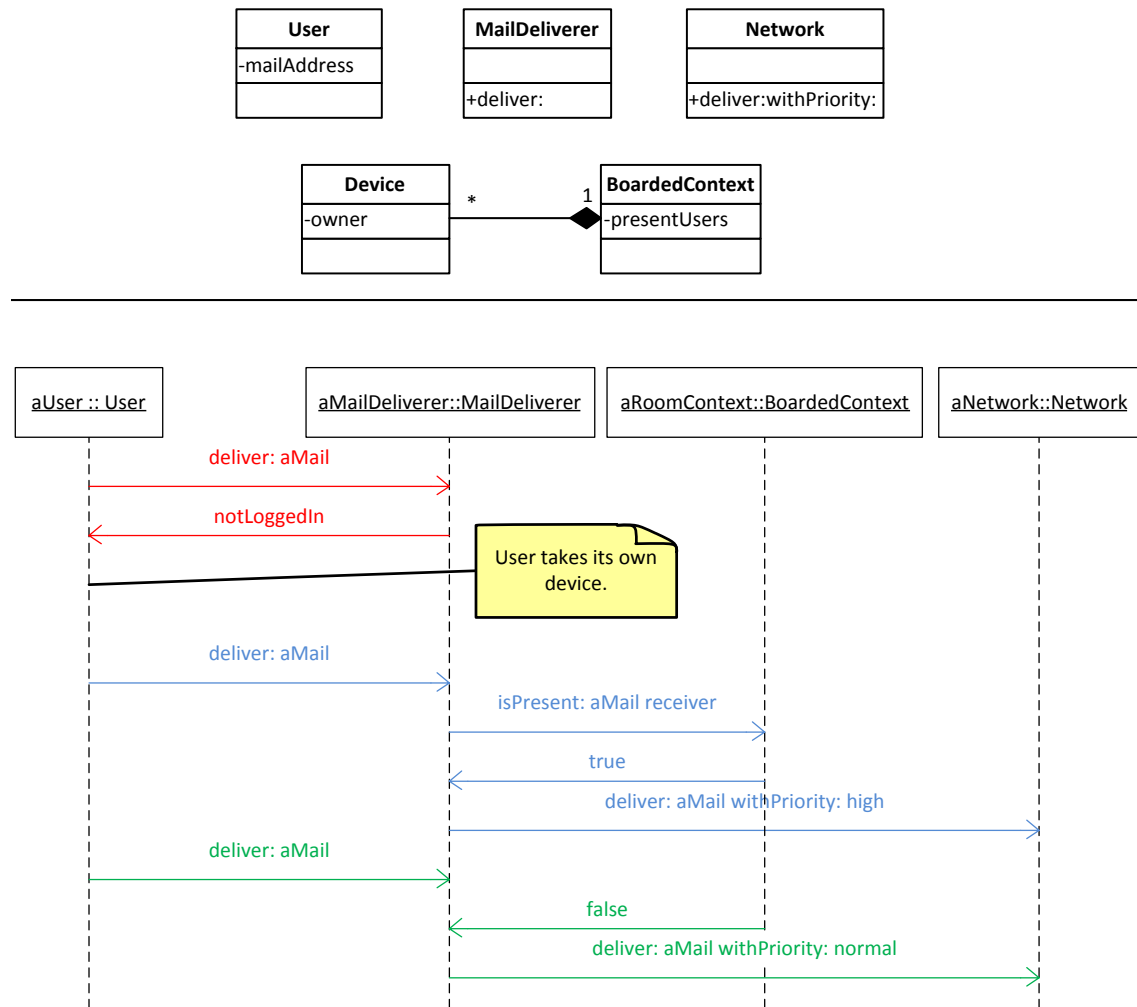


Figure 4.1: Sequence diagram and class hierarchy of the application of users sending emails through mobile devices.

SUBJECTOPIA suits this problem domain, in contrast to perspectives, because SUBJECTOPIA has the capacity to model different subjectivity models. A decision strategy modeling the decision at the side of the receiver of the message suits best this problem domain. Perspectives do not suit this problem domain because this models takes the subjective decision at the sender side of the message.

As SUBJECTOPIA allows us to adapt existing subjectivity models we can adapt the original perspective based approach to suit the problem domain. In SUBJECTOPIA we changed the perspective decision strategy to automatically choose the used perspective depending on the communication context to suit the problem domain Perspectives model the different views users have of the mail deliverer. Because of the changed decision strategy, the mail deliverer, the receiver of the message, and not the user, as in the

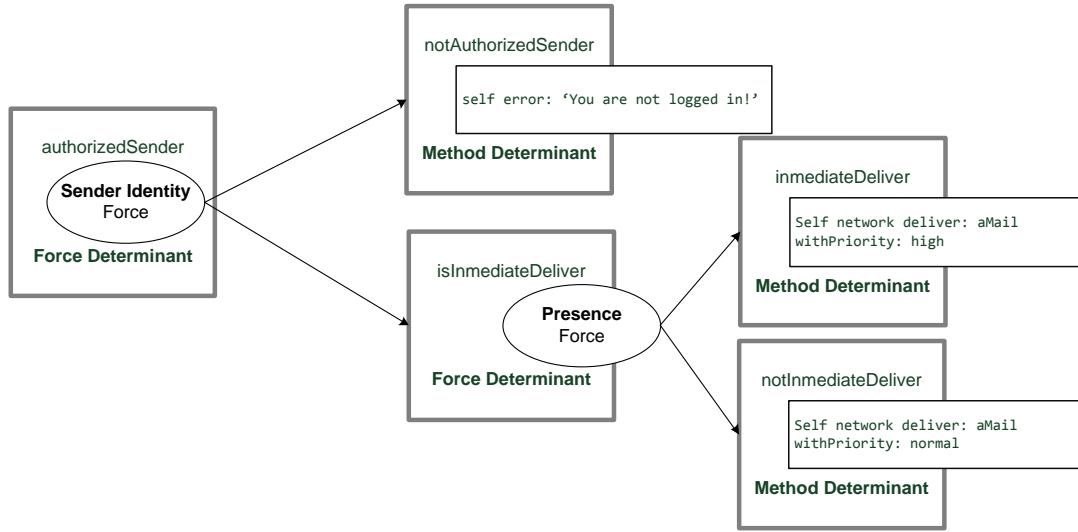


Figure 4.2: Force tree for the message `deliver:` at the mail deliverer.

original implementation, chooses the perspective. Because SUBJECTOPIA models the decision taking process explicitly we can modify it. We can have the mail deliverer responsible for deciding through which perspectives other objects send their messages. The mail deliverer has two perspectives: *delivery* and *deny delivery*, which model the acceptance and denial of the sent emails by users.

Our approach has the advantage of being capable of changing polymorphically the decision strategy. We started with a decision strategy modeling SMB and polymorphically changed it by a decision strategy based on a changed perspective-based decision strategy. This allowed us to choose the model suiting the problem domain the best, without having to change the whole application, but only the decision strategies.

4.2 Case Study – Group Programming Application

Smith and Ungar introduce the group programming application to explain perspectives [3]. In this use case a system keeps track of all the changes to the source code of an object-oriented application. We can either see the changes performed by a single developer or the merged changes of multiple developers.

For this particular example we consider objects to be containers of methods. When a developer needs to see an object's method source code he collaborates with its `MethodContainer`. A `MethodContainer` models a container for the source code of one object. To obtain the textual representation for a particular method the developers send the message `getSourceCodeFor: aMethodName` to the `MethodContainer`. The `MethodContainer` reacts subjectively to the message `getSourceCodeFor:` depending on the contextual view of the developer. To model the different views of the object we use perspectives. Hence, we install a perspective decision strategy for the message `getSourceCodeFor:.` A single perspective defines the changes that one developer performs to the system. We model the changes of the source code for a particular as contextual elements representing layers. The perspectives are composed contextual elements sent by the developer together with the message `getSourceCodeFor:.` The textual representation of the source code is different depending on the chosen perspective.

The perception of the textual representation of the source code depends on the applied changes on

the source code by the developer. Each developer defines its own perspective used to obtain the textual representation of the source code.

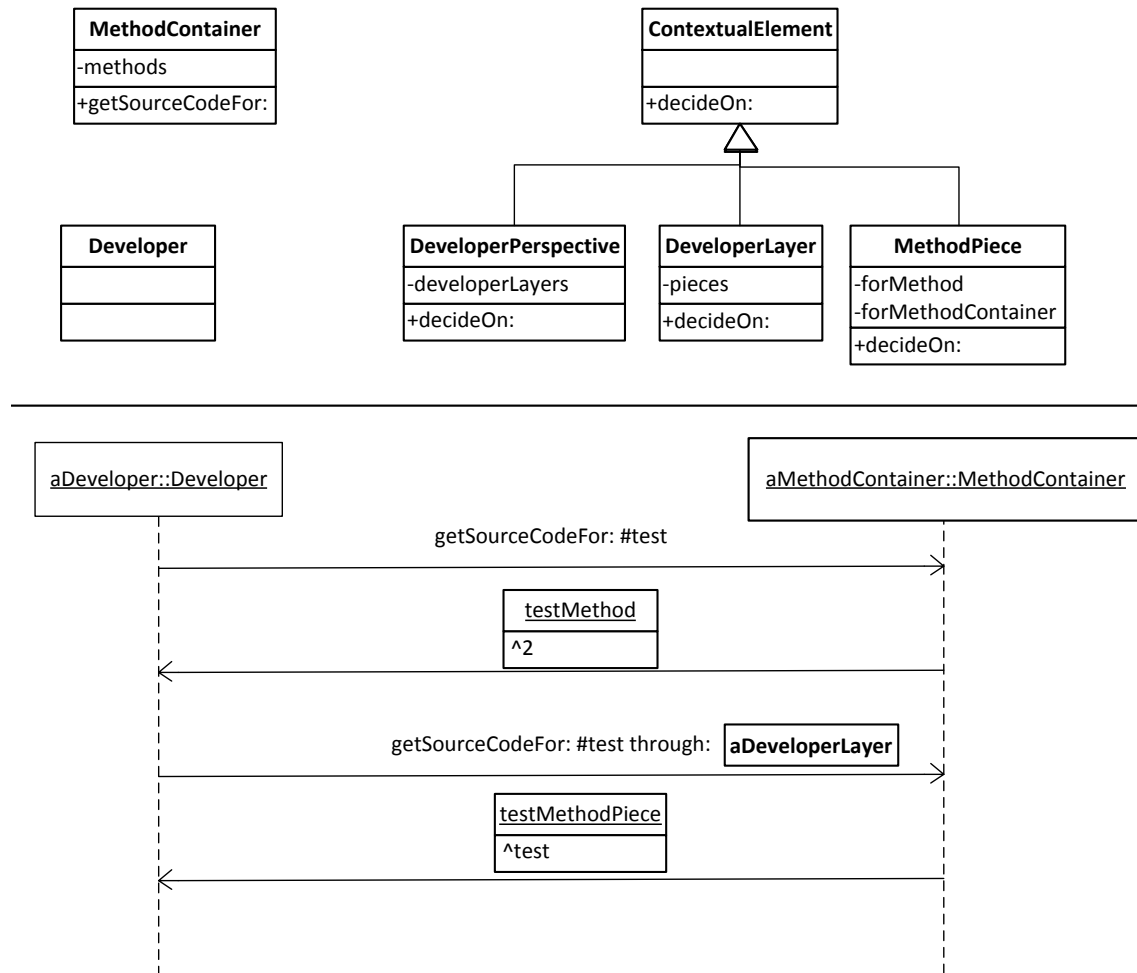


Figure 4.3: Sequence diagram and class hierarchy of group programming use case based on perspectives in SUBJECTOPIA.

In SUBJECTOPIA we use decision strategies modeling perspectives to implement this use case. Figure 4.3 describes the sequence diagram and the class hierarchy of the group programming use case modeled using perspectives. The `MethodContainer` models the original object oriented class as a container of methods. The method container is a subject as it defines a subjective behavior for the message `getSourceCodeFor:`. The message `getSourceCodeFor:` sent to the method container returns the original method of the object. We model the different views of developers of the same method as `DeveloperPerspective` contextual

element. A developer perspective has a set of developer layers. Each time the developer changes a method the application adds a new `DeveloperLayer` to the perspective. The layers have a piece for each changed method. The developer selects its contextual view by sending the message `getSourceCodeFor:through:` to the method container. In Figure 4.3 the sequence diagram shows how a developer sends the message `getSourceCode` to the method container once directly and once using a perspective. The returned source code depends on the contextual view *i.e.*, the chosen perspective by the sender of the message.

We are aware that messages are always sent through a perspective. Hence, we add an empty perspective to directly access the methods of the method container.

Other approaches are not well suited for naturally solving the group programming use case. For example, SMB models changes to the source code as forces. This is not natural because forces influence objects behavior and we need to have multiple views on an object. Additionally, force trees are not supposed to change, *i.e.*, add or remove determinants, at runtime. If we intend to have dynamic force trees we need to check after each change that the force tree is still complete, acyclic, free of simultaneously active determinants and that all leaf nodes are method determinants.

4.3 Case Study – Bank Account Application

The bank account application is introduced by Smith and Ungar [3]. We have implemented the application in SUBJECTOPIA using decision strategies modeling perspectives. The subjective behavior of the message `addAndRecord:` depends on the class of the sender of the message. Only bank account objects sending `addAndRecord:` have an effect on the balance at the receiver of the message.

Figure 4.4 describes the sequence diagram of the bank application. The user sends the message `addAndRecord:` through the perspective `aUserPerspective` modeling the behavior in the piece corresponding to the message. The bank account uses the `aBankAccountPerspective` instead.

We already discussed the problems arising from using perspectives or roles, because the sender of the message always takes the subjective decision. SUBJECTOPIA allows us to redefine the decision strategy of perspectives modeling the decision of which perspective has to be chosen at the side of the receiver of the messages. We can polymorphically change the new decision strategy to the default perspective decision strategy to experiment if that suits better the problem domain.

4.4 Case Studies in Moose

In this section we discuss the use cases extracted from subjectivity requirements taken from the *Moose* platform. Before we start, we mention those parts of Moose that concern our use cases.

4.4.1 Short Introduction to Moose

Moose is an open source and language independent reengineering environment providing facilities to analyze, query, visualize and navigate object oriented software systems. Moose uses a generic meta-model to describe applications developed in object oriented languages. In Moose we use models for the analysis. Applications developed in Smalltalk and Java can be directly imported into Moose. Moose also supports other languages by extending Moose with plugins. A third option is to define your own parser to extract the model [19].

The model of the software system represents the object oriented artifacts, such as methods or classes, as a set of entities. Entities model software artifacts in Moose [20]. Examples of entities are `FAMIXClass` representing classes and `FAMIXMethod` representing methods. `FAMIXClass` and `FAMIXMethod` are entities of FAMIX a language independent meta-model for object-oriented systems [21]. Each kind of entity allows a set of actions *e.g.*, we can visualize a `FAMIXClass` as UML class diagram.

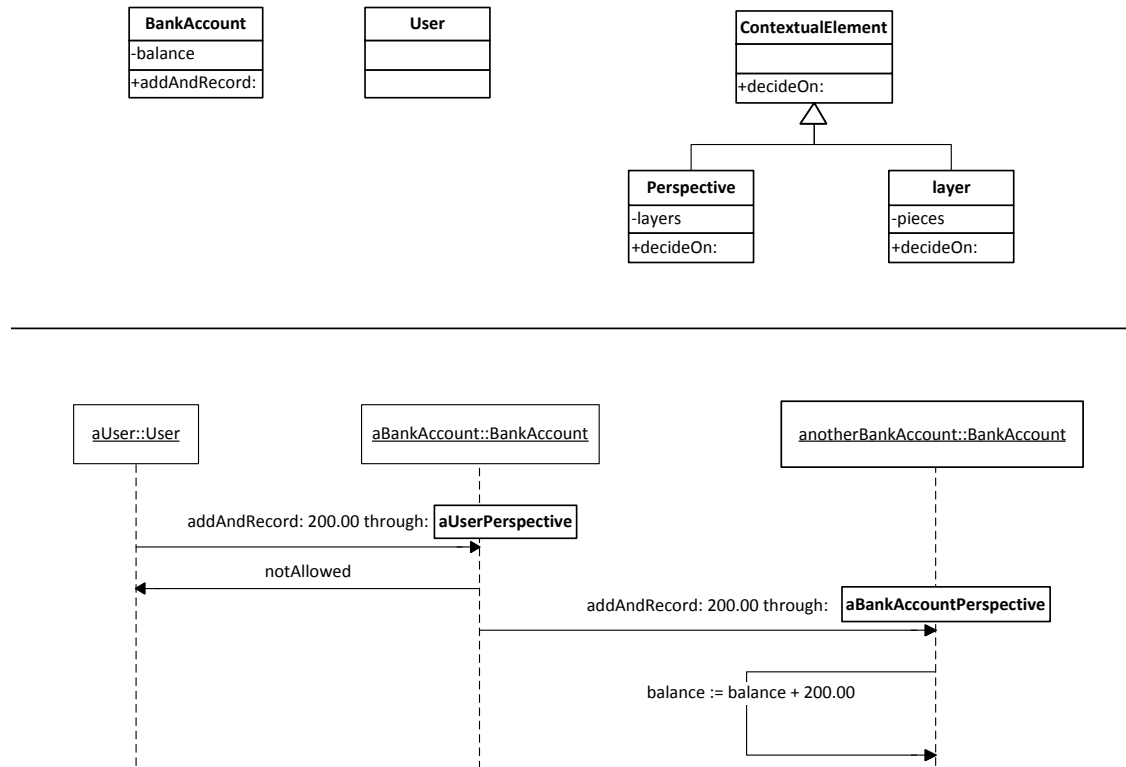


Figure 4.4: Sequence diagram and class hierarchy of the bank account use case.

Collections of entities, called `MooseGroup`, are also entities in Moose. Like any other entity, groups can have specific actions as well, *e.g.*, we can visualize a group of `FAMIXClass` as system complexity. The system complexity displays source code as a graph where each node represents a class and each edge represents the inheritance relationship [22].

Moose provides a graphical user interface to interact with the model of a software system. A list of all entities provides the possible set of actions depending on what they represent. Each listed entity has a different right-click menu with all of the actions.

Figure 4.5 describes how Moose handles its entities. In the uses cases we focus on the `FAMIXClassGroup` being a group of Moose entities where all of them are of class `FAMIXClass`. Concretely we concentrate on the following subjective behavior treated in more details in their corresponding sections:

viewSystemComplexity. A Moose group containing `FAMIXClass` entities only defines a behavior for the message `viewSystemComplexity`. All other Moose groups do not. Treated in Section 4.4.2.

viewDuplicationComplexity. A `FAMIXClassGroup` does only provide a behavior for that message if we previously computed the code duplications. Treated in Section 4.4.3.

viewAsSelectionOnSystemComplexity. The method corresponding to the message

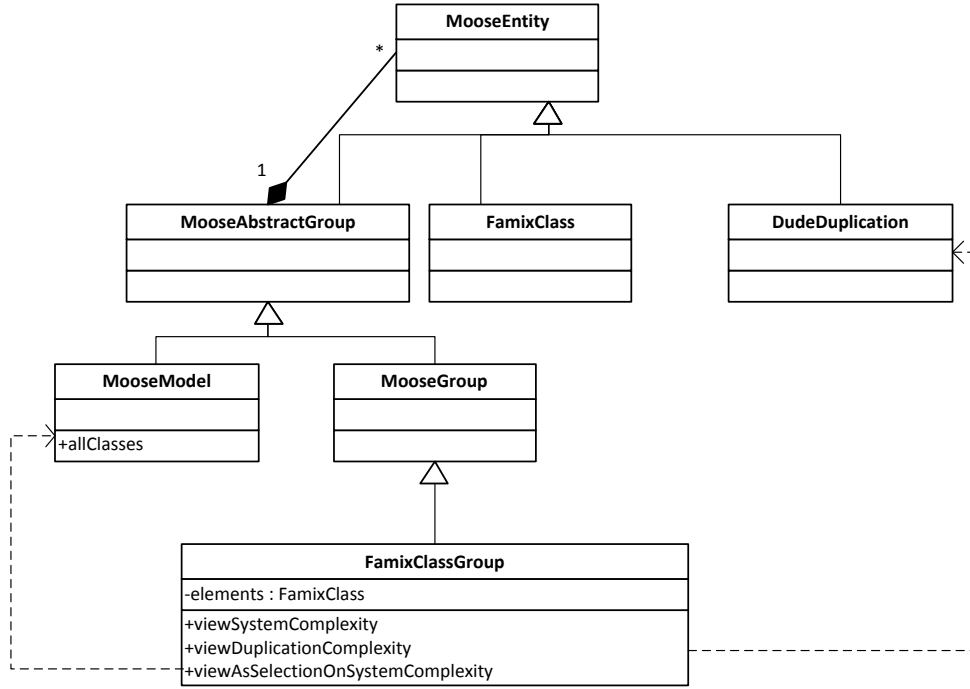


Figure 4.5: Class hierarchy of Moose entities.

`viewAsSelectionOnSystemComplexity` in `FAMIXClassGroup` sends the message `allClasses` to the `MooseModel`. The `allClasses` contains different information depending on the instance of the graphical user interface from which the message `viewAsSelectionOnSystemComplexity` was send. Treated in Section 4.4.4.

4.4.2 Case Study – Subjective Behavior of Group of Entities in Moose

Objects of class or subclass `MooseGroup` offer different behavior depending on the class of the collected entities. Hence, Moose groups behave subjectively for certain messages. As an example we take the subjective behavior for the message `viewSystemComplexity` showing the system complexity view of a software system. The system complexity view is polymetric *i.e.*, depends on multiple code metrics. The number of methods or lines of codes, for example, is a code metric. The size of the nodes representing the classes in the system complexity view depends on the metric values of the classes. The number of methods, for example, is a metric obtainable from a class. The edges of the system complexity view for the boxes depend on the class hierarchy of the system [23]. When a `MooseGroup` receives the message `viewSystemComplexity` it only visualizes the system complexity if all its entities are of class `FAMIXClass`. Moose only offers the action of viewing the system complexity if the Moose only contains entities representing classes.

Figure 4.6 describes our solution of modeling `MooseGroup` as subjects behaving subjectively for the message `viewSystemComplexity`. We separately model the subjective decision, called *DecideActions*, and the behavior, called *systemComplexity*, as decision strategies. The *DecideActions* determines whether the

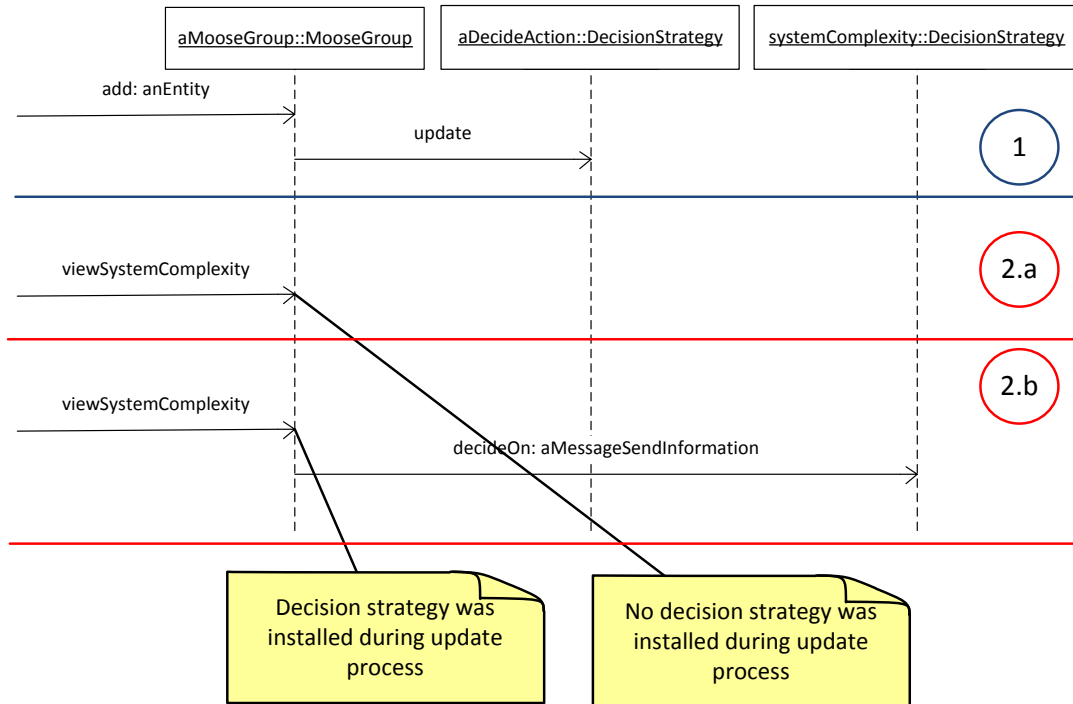


Figure 4.6: Step 1: Updating decision strategies when the entities in the Moose group are changed. Step 2.a and 2.b: Two scenarios for the same message `viewSystemComplexity` with subjective behavior depending if decision strategy for that message was installed in step 1 or not.

`MooseGroup` has a behavior for `viewSystemComplexity` or not. If a `MooseGroup` contains only `FamixClass` entities the *DecideActions* attaches the *systemComplexity*. Each time we manipulate the list of entities Moose evaluates the decision strategy *DecideActions*. The evaluation determines the actions made available to the Moose group.

The `MooseGroup` provides a method for the message *updateTypeAccordingToEntities* called each time we manipulate the list of entities. We override the method to evaluate the *DecideActions* decision strategy:

```

MooseGroup>>updateTypeAccordingToEntities
(self decisionMetaObject at: #updateTypeAccordingToEntities) update: self

```

The *DecideActions* decision strategy has the same responsibilities as the *updateTypeAccordingToEntities* method. In contrast to the original responsibility we add the necessary decision strategies to the Moose group instead of changing the class of the object. In this use case we attach the decision strategy modeling the behavior for the message `viewSystemComplexity` if all entities are of type `FamixClass`.

Up until now Moose uses the subclassing of `MooseGroup` to model subjective behavior. For example `MooseGroup` entities containing only `FAMIXClass` entities are of class `FAMIXClassGroup`. Therefore, changes in the list of entities can result in a change of the runtime class of the group. Additionally, the hierarchy chain of `MooseGroup` subclasses becomes complex due to shared behavior. Defining new entities with new actions can even break the hierarchy of `MooseGroup`. Consider that we introduce the imagi-

nary Moose group `MinimalFAMIXClassGroup` containing only `MinimalFAMIXClass` entities. We define `MinimalFAMIXClass` as a stripped down version of `FAMIXClass` containing less information. Because of missing information `MinimalFAMIXClassGroup` cannot provide all actions of `FAMIXClassGroup`. Nevertheless, `FAMIXClassGroup` can reuse all actions of `MinimalFAMIXClassGroup`. Thus the `FAMIXClassGroup` now subclasses `MinimalFAMIXClassGroup` instead of the `MooseGroup`. The changes in the original class hierarchy may have severe impacts on already instantiated objects of class `FAMIXClass` and also on extracted models. Figure 4.7

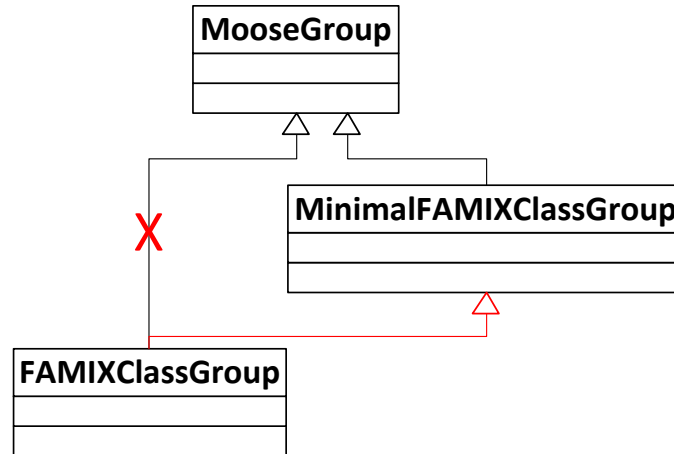


Figure 4.7: Changes – red path – in the original class hierarchy – black path – of Moose when defining the `MinimalFAMIXClassGroup`.

Our approach allows us to easily extend `MooseGroup` to implement other subjective behavior without depending on the class hierarchy. We simply change the *DecideActions* to decide the new case and model the new behavior as a decision strategy. 10 decision strategies can be combined in $2^{10} - 1 = 1023$ different ways. Finding a class hierarchy modeling this without duplication of behavior implies tedious work.

Moose is a legacy system and hard to change. To minimize the impact of the changes we used a different model than the traditional subjectivity approaches. For example, using COP implies to translate a big part of the system to layers which means a considerable effort. The elements contained in the `MooseGroup` influence its subjective behavior. Hence, we need to define an activation protocol for the layers. Splitting the contextual behavior of `MooseGroup` into a set of layers also implies a high effort because of the shared behavior between the different kinds of groups. For example, future adaptations may reuse the behavior for the message `viewSystemComplexity` in other `MooseGroup` not only containing `FamixClass` entities. The layer activation protocol needs to model the selection of multiple layers even if we model each `MooseGroup` behavior as a separate layer.

4.4.3 Case Study – Subjective Behavior Influenced by Third-Party Entities in Moose

The first case study of Moose depends on an intrinsic property of the Moose group. Moose groups may also provide subjective behavior based on entities that are not in the Moose group but in the Moose model. As an example we take the subjective behavior for the message `viewDuplicationComplexity` showing the

duplication complexity view of a software system. The duplication complexity view is a polymetric view in which boxes represent the degree of internal, *i.e.*, within the class, and external code duplication [22]. `DudeDuplication` entities model code duplication computed based on the `FAMIXClass` entities. A Moose model not containing any `DudeDuplication` entities has two possible reasons. Either the duplications were not yet computed or the model does not have any code duplication. A Moose group, consisting of `FAMIXClass` entities should only show the duplication view if the model contains `DudeDuplication` entities.

Up until now, `FAMIXClassGroup` entities did always provide the duplication complexity view. If no `DudeDuplication` entities were in the model when sending the message `viewDuplicationComplexity` the Moose group computed them. If all `FAMIXClass` of the Moose model do not contain any code duplication Moose shows an empty duplication complexity view.

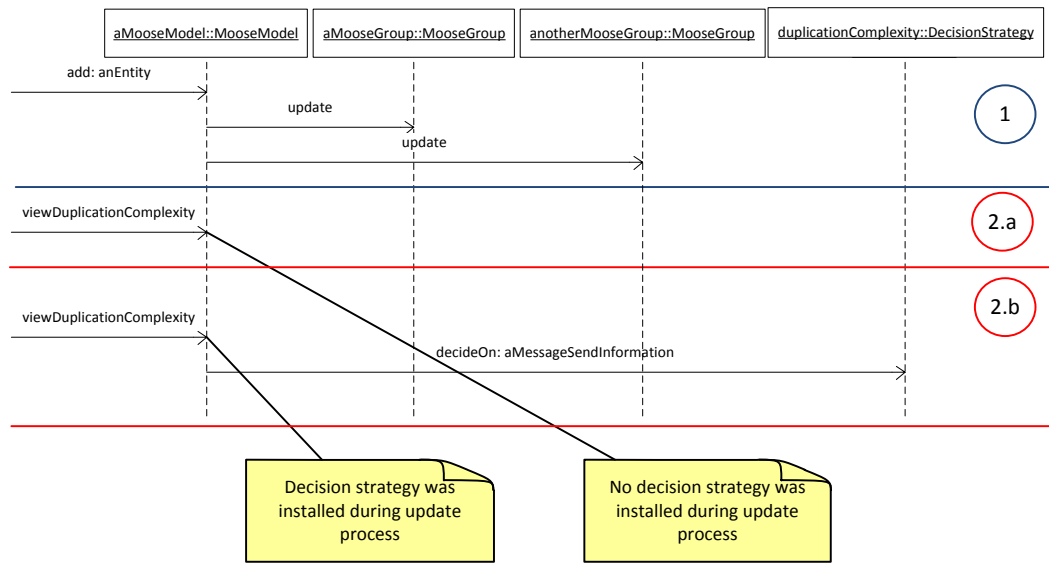


Figure 4.8: Step 1: Updating decision strategies when the entities in the Moose model are changed. Step 2.a and 2.b: Two scenarios for the same message `viewDecisionComplexity` depending if its decision strategy was installed during the update process or not..

Figure 4.8 describes our solution, which is similar to the solution proposed in Section 4.4.2. We model a decision strategy `duplicationComplexity`, which is only attached to the Moose model, if there are `DudeDuplication` objects in the model. We model the decision strategy *DecideDuplication* deciding whether to attach the `duplicationComplexity` decision strategy to the Moose model or not. In contrast to the first Moose problem the update is not performed when adding or removing entities in the Moose group. When the entities, normally Moose groups, in a Moose model change, the Moose model evaluates the *DecideDuplication* decision strategy. The evaluation determines for each Moose group whether to attach the `viewDuplicationComplexity` or not. The attaching depends if the computed dude duplication entities belong to a particular Moose group in the model. To this end we override the method for the message `update`: in the *DecideDuplication* decision strategy:

```
DecideDuplication>>update: aSubject
((aSubject mooseModel allDuplications) size) > 0)
  ifTrue: [aSubject register: (ViewDuplicationStrategy new) for: #viewDuplication.
aSubject viewDuplication.]
```

As well as in the first solution we used a different model than traditional subjectivity approaches due to Moose being a legacy system. Legacy systems are hard to change without affecting the overall stability. Using SUBJECTOPIA we can iteratively model subjective behavior in the application, as we can change the decision strategy.

4.4.4 Case Study – Subjectives menus in Moose

Moose provides a generic graphical user interface to interact with the model of the software system. Figure 4.9 lists the `MooseGroup` entities of the Moose model. A right click on a group opens the contextual menu listing the possible actions. For example a group of `FamixClass` entities shows the action *Visualize* → *System complexity*. The selection of a menu entry sends a message to the selected group. For example, selecting *Visualize* → *System complexity* sends the message `viewSystemComplexity` to the selected `FAMIXClassGroup`.

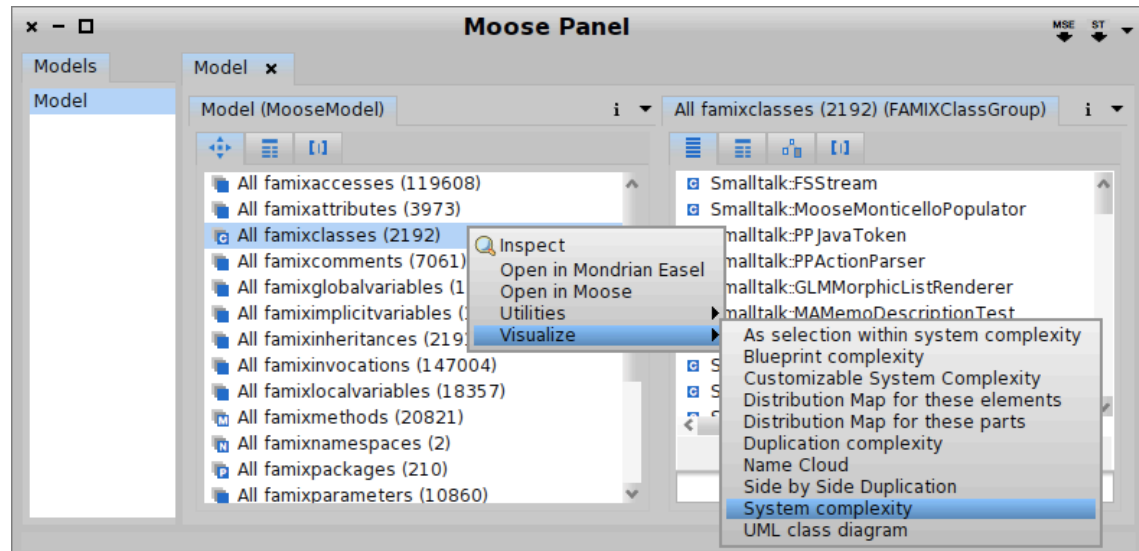


Figure 4.9: User interface provided by Moose. Selecting the entry `System Complexity` results in sending the message `viewSystemComplexity` to the selected group of classes.

The problem is that some visualizations may require contextual information not retrievable from the objects and subjects involved in the communication. Consider that we select a group of classes that we highlight on the overall system complexity. In Moose this is achieved by sending the message `viewAsSelectionOnSystemComplexity` to a `FamixClass` Moose group. To show the selected Moose group highlighted in the whole system complexity we require *all* other `FamixClass` entities of the model to create this visualization. Nevertheless, in different analysis contexts we only see a subset of all classes as a basis for the visualization. Hence, the simple action of `viewAsSelectionOnSystemComplexity` requires both the receiving group and the reference group. Moose uses a global variable to store the system wide `FamixClass` entities. The problem is that each new instance of the graphical user interface of Moose overrides the value of that global variable and this results in unwanted side effects.

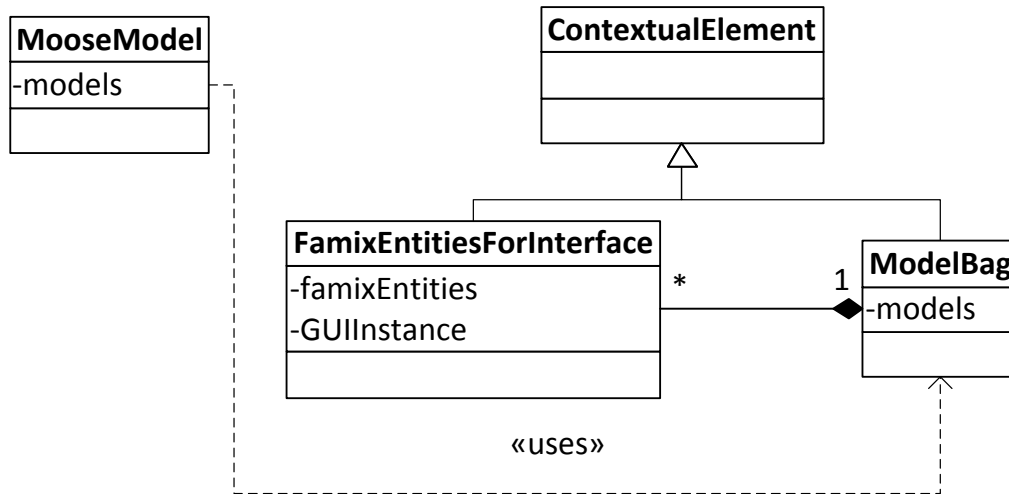


Figure 4.10: Class hierarchy of the contextual elements used to provide user interface instance related class support.

Figure 4.10 describes our solution that uses contextual elements to model the additional, context-sensitive information. The context influencing the behavior of the selected `FamixClass` group is *all* `FamixClass` entities of that model. Therefore, each model creates and maintains its own set of contextual elements holding *all* of its `FamixClass` entities for each user interface. We call the contextual element containing the `FamixClass` entities for one user interface instance `FamixEntitiesForInterface`. Each new interface initialization makes the model generate a contextual element holding the `FamixClass` entities for that particular interface. The composed contextual element `ModelBag` contains a `FamixEntitiesForInterface` for each user interface instance. We use a decision strategy modeling the behavior for the message `viewAsSelectionOnSystemComplexity`. The decision strategy has access to the contextual element `ModelBag` of its model *i.e.*, all `FamixClass` entities of the model. The decision strategy determines, using the meta-information of the message, which interface has sent the message and accordingly uses that contextual element.

With the SUBJECTOPIA approach we can model context-dependent behavior while not all other approaches cannot. For example, using roles does not suit this problem domain, as roles model different behaviors and not a way of reflecting on the context. The Moose groups behave subjectively depending on contextual information which is not included in the default message object. Roles also assume that the sender determines through which role it knows the `MooseGroup`. In reality the `MooseGroup` determines the active roles for this problem domain.

5

Conclusion

The analysis of the previous subjective behavior use cases demonstrated the lack of a unified subjectivity model. Up until now all subjective behavior models have a particular point of view towards subjectivity. We showed that the subjective model behavior is subjective to the problem domain.

In this thesis we introduce a unified approach to model subjectivity called SUBJECTOPIA. We surveyed prior work and identified a lack of generality when modeling different problem domains. The prototype implementation of SUBJECTOPIA allowed us to prove that we can model different subjectivity models such as perspectives, roles, COP and SMB. Hence, SUBJECTOPIA does not force the developer to use a particular subjectivity model when developing a software system. We prove in different use cases that SUBJECTOPIA adapts the model of subjective behavior to the problem domain.

SUBJECTOPIA reifies subjects, decision strategies and contextual elements to provide a more general approach to subjective behavior modeling. Due to explicitly defining these abstractions we can experiment with different subjectivity models in SUBJECTOPIA in the same application. As SUBJECTOPIA supports perspectives, roles, COP and SMB as subjectivity models we can reflect upon them accordingly in the validation. Moreover, we are able to define our own subjectivity model, thus adapting a legacy system like Moose is possible.

Explicit decision strategies and contextual elements allow us to define all prior subjectivity models. Hence, in SUBJECTOPIA we can define subjectivity models independently from the location of the subjective decision, either at the sender or the receiver of the message. Additionally, we can also reify an existing subjectivity model to change the location of the subjective decision. We reified the decision strategy of the perspective based approach, for example, to choose the perspective for the communication by the receiver of the message. Prior approaches did not model this and thus did not suit all problem domains. In SMB, for example, the lack of letting the sender take the subjective decision in the group programming use case is the reason it is not suited for that particular problem domain.

To prove that the SUBJECTOPIA approach allows us to model all existing subjective approaches as well as custom new ones we developed a fully working prototype written in Smalltalk. We also showed that other approaches cannot model all use cases while our approach can adapt and represent them. For this purpose, we presented the implementation of different non-trivial subjective use cases.

SUBJECTOPIA also allows us to selectively implement subjective models on a legacy systems such as Moose. If we use COP on a legacy system, for example, the entire application has to be rewritten to suit

the subjectivity model.

SUBJECTOPIA is a novel approach to model subjectivity by explicitly modeling subjects, decision strategies and contextual elements. The reification of these abstractions avoids the need to impose a particular paradigm for modeling subjective behavior on the developer.

5.1 Future Work

Introducing subjective behavior in legacy applications might have a considerable impact on the overall behavior of the application. For example using COP in Moose implies to rewrite the entire Moose environment as we cannot translate only a part of the system to COP. We explored this by using our own subjectivity model in Moose. Being able to scope the subjective changes to specific objects helps in controlling this impact. Especially the way we transform an object into a subject by sending the message `becomeSubject` can break a legacy application. In the current SUBJECTOPIA approach sending the message `becomeSubject` changes the class hierarchy of the the object receiving the message.

We plan to analyze reflection frameworks for SUBJECTOPIA to perform subjective adaptations on certain objects only. An example of such a reflection framework is Bifröst (formerly Albedo) [24]. Albedo uses meta-objects to provide a range of reflective features and allow application models and environments to evolve at runtime. Structural meta-objects, for example, offers to add, remove or replace methods at runtime on a single object instance. The message `becomeSubject` manipulates the methods of an object at the class side. Hence, extending SUBJECTOPIA with Albedo allows us to use structural meta-objects to manipulate an object at runtime to be able to add the missing responsibilities of a subject.

The current implementation of SUBJECTOPIA models decision strategies for perspectives, roles, SMB and COP. We need to adapt the implementations of the subjectivity models to fit better with the original intention. Moreover, some of the implementations do not take the latest research into consideration. The decision strategy for COP, for example, only implements explicit layer activation. Also we need to define more subjectivity models in SUBJECTOPIA.

The IDE of Pharo does not support the SUBJECTOPIA approach. For example, in Pharo, a Smalltalk environment, the representation of methods differs from classes even if following the idea of “everything is an object”. Pharo handles decision strategies and contextual elements as common classes. Nevertheless, this does not correspond to the responsibility of decision strategies and contextual elements. To use the SUBJECTOPIA approach, we need to adapt the IDE to support subject oriented programming. The finding of a proper IDE implementation goes beyond the scope of this thesis.

The last remaining point is to further evaluate the SUBJECTOPIA approach with new subjectivity use cases.

Bibliography

- [1] Beck, K.: Smalltalk Best Practice Patterns. Prentice-Hall (1997)
- [2] Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: Proceedings OOPSLA '93, ACM SIGPLAN Notices. Volume 28. (October 1993) 411–428
- [3] Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPoS special issue on Subjectivity in Object-Oriented Systems **2**(3) (1996) 161–178
- [4] Kristensen, B.B.: Object-oriented modeling with roles. In Murphy, J., Stone, B., eds.: Proceedings of the 2nd International Conference on Object-Oriented Information Systems, Springer-Verlag (1995) 57–71
- [5] Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, New York, NY, USA, ACM (October 2005) 1–10
- [6] Darderes, B., Prieto, M.: Subjective behavior: a general dynamic method dispatch. In: OOPSLA Workshop on Revival of Dynamic Languages. (October 2004)
- [7] Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates (2009)
- [8] Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. Volume 22. (December 1987) 227–242
- [9] Madsen, O.L., Møller-Pedersen, B.: Part objects and their location. In: Proceedings of the seventh international conference on Technology of object-oriented languages and systems, Hertfordshire, UK, UK, Prentice Hall International (UK) Ltd. (1992) 283–297
- [10] Kristensen, B.B.: Subjective method interpretation in object-oriented modeling. In: In Proceedings of the 5th International Conference on Object-Oriented Information Systems, Springer-Verlag (1998) 9–11
- [11] Kristensen, B.B.: Subjective behaviour. Comput. Syst. Sci. Eng. **16**(1) (2001) 13–24
- [12] de Champeaux, D., Anderson, A., Feldhousen, E.: Case study of object-oriented software development. In: Proceedings OOPSLA '92, ACM SIGPLAN Notices. Volume 27. (October 1992) 377–391
- [13] Kristensen, B.B., Osterbye, K.: Roles: Conceptual abstraction theory & practical language issues. In: Special Issue of Theory and Practice of Object Systems (TAPoS) on Subjectivity in Object-Oriented Systems. (1996) 143–160
- [14] Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology **7**(3) (March 2008)

- [15] Knight, A., Dai, N.: Objects and the web. *IEEE Software* **19**(2) (March 2002) 51–59
- [16] von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: Beyond layers. In: *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, ACM Digital Library (2007) 143–156
- [17] Costanza, P.: Context-oriented programming in context!: state of the art. In: *Celebrating the 50th Anniversary of Lisp. LISP50*, New York, NY, USA, ACM (2008) 4:1–4:5
- [18] Dozsa, A., Gîrba, T., Marinescu, R.: How Lisp systems look different. In: *European Conference on Software Maintenance and Re-Engineering (CSMR 2008)*, IEEE Computer Society Press (2008) 223–232
- [19] Gîrba, T.: *The Moose Book*. Self Published (2010)
- [20] Ducasse, S., Lanza, M., Tichelaar, S.: *The Moose reengineering environment*. *Smalltalk Chronicles* (August 2001)
- [21] Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, IEEE Computer Society Press (2000) 157–167
- [22] Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, New York NY, ACM Press (2005) 1–10 Invited paper.
- [23] Lanza, M., Ducasse, S.: Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)* **29**(9) (September 2003) 782–795
- [24] Ressia, J., Renggli, L., Gîrba, T., Nierstrasz, O.: Run-time evolution through explicit meta-objects. In: *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*. (October 2010) 37–48
- [25] Goldberg, A., Robson, D.: *Smalltalk-80: The Language*. Addison Wesley (1989)

Appendices

A.1 Installation Guide

We implemented SUBJECTOPIA in Pharo version 1.1.1, a Smalltalk environment. Smalltalk is a pure object-oriented, dynamically typed, reflective programming language [25]. Pharo is an open source implementation of the Smalltalk programming language and environment. Pharo consists of a Virtual Machine (VM) and images that contain the Smalltalk environment and the code. The VM in Pharo is completely written in Smalltalk and available for different OS like Windows, Linux and MacOS [7].

Before installing SUBJECTOPIA we need the Pharo VM. You can download the latest version of the Pharo VM at <http://www.pharo-project.org/>. For the VM to work we also need the source files located in the same source.

Once we have the VM, we can get started using SUBJECTOPIA by either using a ready-made image (suggested), or try to load the code manually.

Ready-Made Image You can download the ready-made image from <http://scg.unibe.ch/research/subjectopia>. Unzip all the files and drag the file named `subjectopia_v1.0.image` and drop it on the Pharo VM.

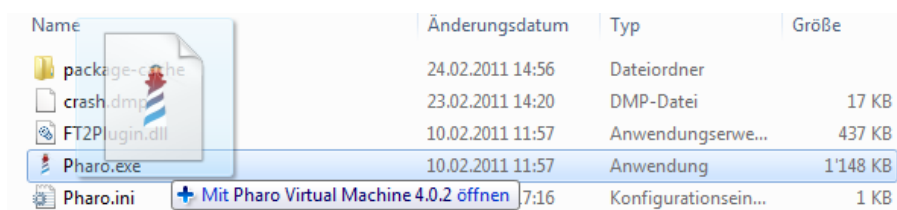


Figure 1: Dragging `subjectopia_v1.0.image` and drop to Pharo Virtual Machine in Windows.

The current image of SUBJECTOPIA uses the Moose image to reflect about the use case studies of Moose introduced in Section 4.4.

Manual installation We developed and tested SUBJECTOPIA in the PharoCore image version 1.1.1. We recommend to use that or later versions, downloadable from <http://www.pharo-project.org/>. The SqueakSource repository hosts the code of SUBJECTOPIA at <http://www.squeaksource.com/Decisionstrategy>. Use Monticello to download the code of SUBJECTOPIA from squeaksource. From the *World* menu select the entry `Monticello Browser`. In the opened window select `+Repository` and pick HTTP. Paste the following text in the new window:

```
MCHttpRepository
  location: 'http://www.squeaksource.com/Decisionstrategy'
  user: ''
  password: ''
```

You can ignore the warning, as only the Moose use case studies depend on the missing classes, SUBJECTOPIA does not. If you intend to have a look at the implementation of the Moose example you have to manually install Moose.

Moose Use Case Examples If you intend to run the example implementation of the use cases of Moose you can do so by installing SUBJECTOPIA in the Moose environment. You can download Moose from <http://www.moosetechnology.org/>. In the *Workspace* write `Subjectopia start` to start Moose with decision strategies. At any time you can write `Subjectopia stop` in the *Workspace* to stop subjective Moose. We have to point out that models extracted in subjective Moose may not work in Moose.

Also Moose models may not work in subjective Moose. We retain that subjective Moose is not thought to be used in productive scenarios, as only being a proof-of-concept solution.

A.2 Introductory Example for SUBJECTOPIA

In this section we introduce a simple use case of modeling private variables of objects in Smalltalk. The use case aims to model objects whose instance variables are only accessible by other objects of the same class. In Smalltalk, instance variables are always private to the object's instance, thus other objects of the same class cannot access them as in Java, for example.

We model a decision strategy that only returns the instance variable in some contexts. To return the instance variable, the sender and receiver of the message have to be of the same class. The message selector corresponds to the name of the private variable.

We create a new category `Subjectopia-Private` in Pharo by opening the `World` menu. To open the `World` menu we click with the mouse on the background of the main window. Next, we open the `System Browser` by selecting the entry `System Browser` in the `World` menu. The `System Browser`, shown in Figure 2, lists all available categories in the image. To add a new category we right-click in the list of categories and select the entry `add category...`

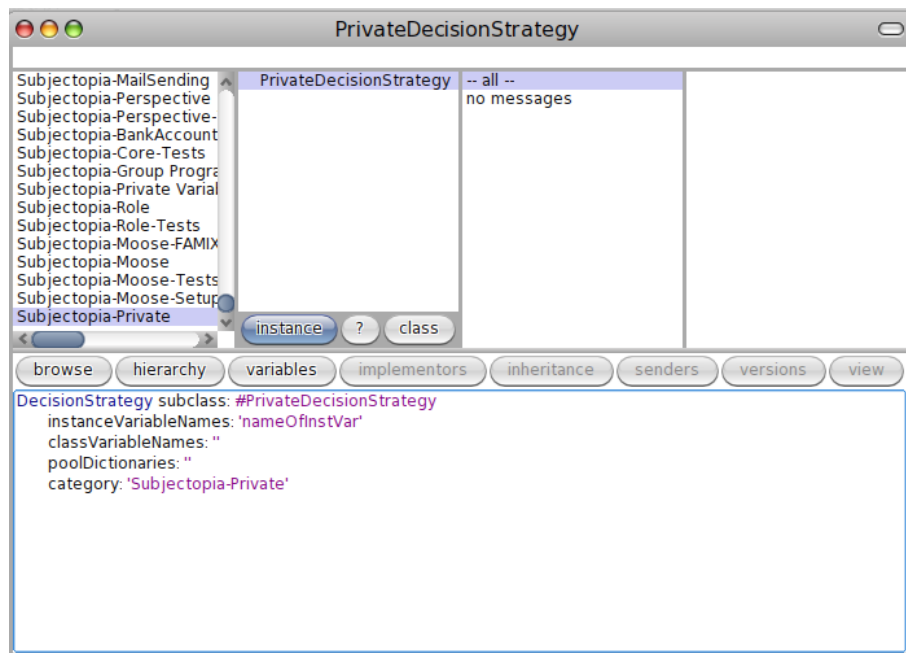


Figure 2: System Browser listing the categories, also the category `Subjectopia-Private`.

In the newly created category we model our simple decision strategy checking whether the sender and the receiver of the message are instances of the same class. We first create the decision strategy by subclassing `DecisionStrategy`. Figure 2 shows, at the bottom text panel, how to subclass `DecisionStrategy` to create the class

`PrivateDecisionStrategy:`

```
DecisionStrategy subclass: #PrivateDecisionStrategy
  instanceVariableNames: 'nameOfInstVar'
```

```
classVariableNames: ''
poolDictionaries: ''
category: 'Subjectopia-Private'
```

Pressing CTRL + S generates the class `PrivateDecisionStrategy` in the image. The class `PrivateDecisionStrategy` has an instance variable called `nameOfInstVar`. The instance variable saves the name of the instance variable for which we use the decision strategy.

In the class `PrivateDecisionStrategy` we override the method `decideOn:` to implement the following subjective behavior:

```
decideOn: aMessageSendInformation
  ^((aMessageSendInformation sender class) = (aMessageSendInformation receiver class))
  ifTrue:
    [
      (aMessageSendInformation receiver) instVarNamed: nameOfInstVar.
    ]
  ifFalse:
    [
      self error: self class asString, ' does not understand: ',
      aMessageSendInformation selector asString.
    ].
```

The message `instVarNamed:` is a reflective method returning the value of the instance where the argument suits the instance variable's name.

The decision strategy corresponds to one particular instance variable. For this purpose we create the following method in the class `PrivateDecisionStrategy`.

```
nameOfInstVar: aBinaryString
  nameOfInstVar := aBinaryString
```

We create a simple subject `MockSubject` to test our application by subclassing `Subject`:

```
Subject subclass: #MockSubject
  instanceVariableNames: 'privateVariable'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Subjectopia-Private'
```

We call the private variable of the created class `privateVariable`.

We override the method `initialize` in the class `MockSubject` to set the value of the private variable:

```
initialize
  super initialize.
  privateVariable := 1.
```

For testing purposes we also define the message `privateVariableOf:` in the `MockSubject` as follows:

```
privateVariableOf: aSubject
  ^aSubject privateVariable.
```

We write a test to prove that our decision strategy models exactly what we expect. We therefore define the class `PrivateTest` subclassing `TestCase`.

```
TestCase subclass: #PrivateTest
  instanceVariableNames: 'nameOfInstVar'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Subjectopia-Private'
```

We create the method `testPrivate` as follows:

```

testPrivate
  |aSubject anotherSubject|
  aSubject := MockSubject new.
  anotherSubject := MockSubject new.

  "Registering decision strategy on subject"
  aSubject register: (PrivateDecisionStrategy new nameOfInstVar: #privateVariable)
    for: #privateVariable.

  "As the test object is the sender of the message the instance an exception is thrown
  ."
  self should: [aSubject privateVariable] raise: Error.

  "anotherSubject does not have a decision strategy for the message Variable, hence it
  is not understood."
  self should: [anotherSubject privateVariable. self assert: false] raise: Error.

  "As the subject itself sends the message, the value of the instance variable is
  returned."
  self assert: ((aSubject privateVariableOf: aSubject) = 1).

  "A mock subject sends the message, the value of the instance variable is returned."
  self assert: ((anotherSubject privateVariableOf: aSubject) = 1).

```

We can now right-click on the created method and select `run test`. If you followed the instructions the test should be green.

We also give an introductory example for using contextual elements based on the same use case. **SUBJECTOPIA** models a perspective ready to use for instance variable access, called `InstanceVariableLayerAccess`. We use the perspective based approach to model this problem domain. An object can only access the instance variable if it uses the right perspective. In the `PrivateTest` class we create the method `testPerspectives`. We start with the setup initializing a `MockSubject` subject and the perspective `instVarPerspective`:

```

testPerspectives
  |aSubject instVarPerspective |
  aSubject := MockSubject new.
  instVarPerspective := InstanceVariableLayerAccess new instanceVariableName:
    #privateVariable.

```

We mentioned that **SUBJECTOPIA** models a decision strategy for perspectives. We use the perspective decision strategy for the subject. To test this we append the following to the `testPerspectives` method:

```

aSubject register: (Perspective new) for: #privateVariable.

```

Only if we use the perspective when sending the message `privateVariable` the subject returns its instance variable. We define the following tests in the method `testPerspectives`:

```

[aSubject privateVariable. self assert: false.] on: Error do: [self assert: true.].

self assert: ((aSubject privateVariablethrough: instVarPerspective) = 1)

```

The complete method `testPerspectives` in the class `PrivateTest` is:

```

testPerspectives
  |aSubject instVarPerspective |
  aSubject := MockSubject new.
  instVarPerspective := InstanceVariableLayerAccess new instanceVariableName: #
    privateVariable.

```

```
aSubject register: (Perspective new) for: #privateVariable.  
[aSubject privateVariable. self assert: false.] on: Error do: [self assert: true.].  
self assert: ((aSubject privateVariablethrough: instVarPerspective) = 1)
```

If you followed the instructions and run the tests, they both become green.