Unifying Subjectivity*

Daniel Langone, Jorge Ressia, Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

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 somehow 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.

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 20^{th} 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 various elements that collaborate to achieve a common goal may need to adapt their behavior when specific events or conditions are met.

Object-oriented languages follow the *objective* approach. An object behaves always the same way when receiving the same stimulus. To faithfully model the real-world domains we need mechanisms to model subjectivity. We can characterize the key approaches that have previously been proposed as follows:

Perspectives. Smith and Ungar proposed adding multiple *perspectives* to an object, where each perspective implements different behavior for that object [1]. When an object sends a message through a perspective the receiver

* In *Objects, Models, Components, Patterns, 49th International Conference, TOOLS 2011*, Zurich, Switzerland, June 28-30, 2011. LNCS 6705, pp. 115–130, 2011. doi:10.1007/978-3-642-21952-8_10

behaves differently depending on this perspective. Therefore, an object behaves subjectively depending on the perspective through which other objects see it.

Roles. Kristensen introduced the concept of *roles* to model subjective behavior [2]. People behave differently depending on the role they are playing. For example, the same person may behave differently as a father, an employee or a shopper. A role is attached 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. *Context-oriented programming* (COP) was introduced by Costanza *et al.* [3]. The behavior of an object is split into layers that define the object's subjective behavior. Layers can be activated and deactivated to represent the actual contextual state. When a message is sent, the active context determines the behavior of the object receiving the message.

SMB. Darderes and Prieto proposed *subjective message behavior* [4]. The different behaviors for a message are split into a set of independent methods and combined with a tree-based decision mechanism, called a *force tree*.

Although formally the approaches are equivalent in expressive power, they are not equally suitable in all circumstances. 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 wants to send an email using a mobile device [4]. If the network is available the email should be sent immediately, otherwise the email should be saved and sent when possible. Modeling the network with either roles or perspectives does not make sense. This subjective problem is not about roles of networks or emails, or about perspectives through which they may be seen, but rather about whether the network is available in the current context. Whereas COP or SMB might be more appropriate for modeling subjectivity in this domain, perspectives or roles would be more suitable to model behavior that varies with respect to the sender of a message.

Furthermore, the responsibility of determining which subjective behavior should be selected 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 perspective or role to be used. Consider communicating with a person who might be at work or on holidays, thus triggering completely different responses. In such a case it would make more sense for the receiver and not the sender to determine the subjective behavior.

Our approach. To alleviate the problem of having a fixed subjectivity model, 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 may be turned into a subject. Subjective behavior is modeled by a *decision strategy*. A decision strategy determines the appropriate subjective behavior based on the value of a set of *contextual elements*. Decision strategies can be

configured 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.

Section 2 presents a review of previous approaches to modeling subjective behavior. In Section 3 we explain how Subjectopia models the subjective behavior of objects and discuss our implementation. Section 4 validates our approach by showing the drawbacks of previous approaches in solving subjective problems and demonstrates how Subjectopia circumvents these shortcomings. In Section 5 we summarize the paper and discuss future work.

2 State of the Art

Subject-oriented programming was first introduced by Harrison and Ossher [5]. They advocated the use of subjective views to model variation, thus avoiding the proliferation of inheritance relations. Up to that point subjective behavior was modeled in an *ad hoc* fashion using idioms such as self-delegation and multiple dispatch. Various researchers subsequently proposed dedicated approaches to model subjective behavior in a more disciplined way. We briefly survey the key approaches and discuss their limitations.

2.1 Perspectives

Smith and Ungar [1] proposed to model subjective behavior through a set of possible views of an object. These views are called *perspectives* and are composed of zero or more hierarchically ordered *layers*. Each layer is composed of pieces modeling one behavior for one message and one object. For the approach to be deterministic a layer should never have two or more pieces corresponding to the same message and one object. An object sending a message selects the perspective through which it views the subject. Smith and Ungar developed a prototype called *US* on top of the *Self* [6] programming system.

The approach forces the developer to translate a given problem in terms of perspectives, which may not always suit the problem domain. Consider again the use case in which a user wants to send an email using a mobile device [4]. (If the network is available the email should be sent immediately, otherwise the email should be saved and sent when possible.) Network availability is a property of the current context of the user, not a "perspective" through which sending of email can be viewed.

A further difficulty is that the object initiating an interaction is responsible for selecting the current perspective. By contrast, in this use case it would be more appropriate for the mobile device to decide how to behave.

A general problem of this approach is that there is no way to overrule the process that decides the subjective behavior to be executed for a method. This decision is hardcoded in the internals of the approach.

2.2 Roles

Kristensen [2] stressed the importance of roles in the subjective behavior of entities: *"we think and express ourselves in term of roles"* when dealing with the real world. The notion of a role can be deduced from psychology as a set of connected behaviors, rights and obligations as conceptualized by actors in a social situation.

A role object is attached to a regular object, called an *intrinsic object*, and adds, removes or redefines the latter's original behavior. An intrinsic object together with its role is called a *subject*. Roles have no responsibilities of their own, *i.e.,* they only have meaning when attached to an intrinsic object. The *is-part-of* relationship of the role to its intrinsic object refers to the location of *part objects* introduced by Madsen and Møller-Pedersen [7]. An object sending a message selects the role through which it knows the subject. There are implementations of role-based programming relying on BETA [8] and Smalltalk [8].

Role-based programming forces the developer to model domain entities as playing various roles. Let us consider the group programming example [1] of a system for registering changes on source code of an object-oriented application. In the original implementation changes were modeled as perspectives, allowing us to have different views of the source-code. However, modeling changes as roles does not reflect reality. The source code does not play a particular role but rather is viewed differently by different developers.

As with perspectives, it is the sender of a message that decides which role the subject plays in an interaction. Scenarios in which the subjective behavior should be selected by the subject cannot be modeled directly.

2.3 Context-oriented Programming

Context-Oriented Programming (COP) refers to programming language mechanisms and techniques that support dynamic adaptation to context [9]. COP was first introduced by Costanza and Hirschfeld [3]. The behavior of an object in COP is split into several layers (not to be confused with the layers introduced by perspectives). Each layer models the behavior associated to a particular context. Every definition not explicitly placed in a user-defined layer belongs to a default root layer. When an object receives a message, its behavior depends on the active layer, representing the current context.

ContextL [9] extends CommonLisp with layers and PyContext [10] does the same for Python. Implementations also exist for Java, JavaScript, Smalltalk and Scheme[1].

With COP the developer is required to model subjective behavior in terms of contextual layers. Consider again the use case where a user wants to send an email using a mobile device [4]. If the receiver of the email is in the same room as the sender then the email is sent with high priority. The mail deliverer is responsible for delivering the emails with a given priority. With layers, we will

[1] `http://www.swa.hpi.uni-potsdam.de/cop/implementations/index.html`

have two implementations for the send mail responsibility, one with high priority and the other without. The default layer activation of COP, using explicit layer activation, does not allow us to faithfully model this problem domain. We require a mechanism from the sender to activate the appropriate layer before sending the message. Thus a perspective approach would model this problem better.

2.4 Subjective Message Behavior

Darderes and Prieto [4] proposed to represent subjective behavior by modeling the forces that might influence an object to behave subjectively. There are four types of *forces*: (i) the *sender force* is the object sending the message, (ii) the *self force* is the receiver of the message, *i.e.,* the subject, (iii) the *collaborator force* is any object collaborating with the subject, and (iv) the *acquaintance force* is any other object influencing the message.

Subjective Message Behavior proposes to split all possible behaviors for one message into a set of behaviors. The decision process is realized by a dispatch mechanism called a *force-tree*, consisting of *determinant nodes*, each consisting of a condition to be fulfilled and corresponding behavior. The *method determinant node* models one possible behavior for a given message of the object. The *force determinant node* models a boolean condition based on one force to decide which determinant has to be evaluated next. When an object receives a message, the root determinant of the force tree corresponding to that message is evaluated. The force tree has to be complete, acyclic and free of simultaneously active determinants in order for its evaluation to result in a unique possible behavior for a given message in a particular invocation context. Leaf nodes should always be *method determinant*.

Subjective Message Behavior requires the developer to model subjective behavior using a force tree, which may be overly complex for certain domains: Consider again the group programming example [1] of a system for registering changes to source code of an object-oriented application. Perspectives naturally model the behavior of a developer who wants to see his version of the source code. Casting this use case in terms of forces and force trees introduces unnecessary complexity.

3 Modeling Subjective Behavior

In this section we introduce Subjectopia[2] Both Subjectopia and the examples presented in this paper are implemented in Pharo Smalltalk[3]. Objects with subjective behavior are explicitly modeled as *subjects*, emphasizing the difference to common objects. A subject needs to select its correct behavior from a set of possible behaviors. We use *decision strategies* to explicitly model the way subjective decisions are taken. Finally, *contextual elements* model context-dependent

[2] `http://scg.unibe.ch/research/subjectopia/`
[3] `http://www.pharo-project.org/`

information, which can influence the behavior of a subject. Our model allows us to change the subjective behavior of a subject by changing its decision strategy. Decision strategies and contextual elements allow us to model perspectives, roles, context-oriented programming, subjective message behavior and other subjectivity models. Hence, Subjectopia does not force the developer to use a fixed modeling paradigm.

To explain the Subjectopia model we use the bank account example from the perspective approach [1]. The use case 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 the bank account the money should be transferred to. The bank account object changes its balance by the amount of money transferred and sends the message `addAndRecord:` to the bank account receiving the money. However, a user should not be able to directly send the message `addAndRecord:` to a bank account to guarantee that only bank accounts can trigger a transfer and maintain the balance invariant of the banking application. As a consequence the message `addAndRecord:` has two different behaviors for a bank account, depending on whether a user or a bank account object sends the message.

The following subsections introduce the concepts of *subject*, *decision strategy* and *contextual element* and describe how they can be used to model the example

3.1 Subjects

A *subject* is an object that behaves differently under different contextual circumstances. A subject may be fully subjective or only present subjective behavior for certain responsibilities. To transform a regular object into a subject we send the message `becomeSubject` to the object. For example, we can tell the bank account object `aBankAccount` to become a subject:

```
aBankAccount becomeSubject.
```

The transformation of `aBankAccount` into a subject adds the necessary behavior to enable it to behave subjectively for certain messages. The bank account object can also directly inherit from Subject, which will have the same effect. The bank account subject will only change its balance if the sender of the message `addAndRecord:` is a bank account. We therefore define the message `addAndRecord:` in the `aBankAccount` subject as being subjective:

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

The original behavior of the `aBankAccount` subject for the message `addAndRecord:` is replaced by a decision strategy which models the subjective decision process. Modeling subjects explicitly has the advantage that the subjective parts

[3] Readers unfamiliar with the syntax of Smalltalk might want to 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`.

of an application can be detected and thus reflected upon accordingly. Otherwise, this information would be encoded in the application source code and we would have to use *ad hoc* mechanisms to detect the subjects.

3.2 Decision Strategies

A decision strategy models the process of deciding how a subject has to behave when it receives a specific message. Because we use explicit decision strategies we can define our own or reuse existing decision models such as those that express perspectives, roles, context-oriented programming or subjective message behavior. We can also directly implement behavior in a decision strategy.

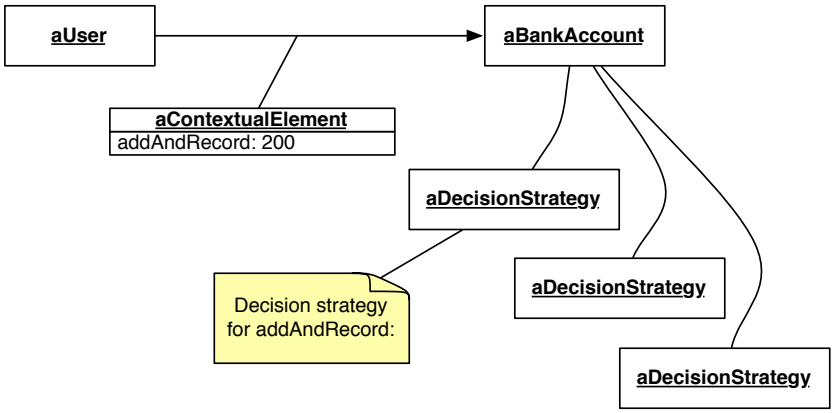

Fig. 1. The object `aUser` sends message `addAndRecord:` with argument 200 to `aBankAccount`. The subject performs a lookup and finds the subjective method. The method evaluates the decision strategy selecting the appropriate behavior for the current context.

Figure 1 shows the process of the subject `aBankAccount` receiving the message `addAndRecord:` from `aUser`. The subject performs a traditional method-lookup. Since `addAndRecord:` was defined as a subjective method, the method's behavior is adapted to evaluate the decision strategy:

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

The subjective method uses two steps to make the subject behave subjectively. The first step consists in the subject creating a contextual element representing the meta-information of the message. In Figure 1 this message object contains the following information:

- The message selector `#addAndRecord:`

- The argument 200.
- The sender of the message, the `aUser` object.
- The receiver of the message, the `aBankAccount` subject.

The second step consists in evaluating the decision strategy with the contextual information provided by the message object. The decision strategy determines which information provided by the message object is used. The evaluation of a decision strategy may be resolved as:

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

In Figure 1 the user object `aUser` wants to change the balance of the bank account `aBankAccount`, increasing it by 200 Fr. The decision strategy examines the message object and denies the request to change the balance because the sender is a user object.

Decision strategies can be replaced, in case the paradigm for modeling subjective behavior needs to be adapted over time. It is even possible to use multiple decision strategies within a single subject, thus allowing, say, role-based and perspective-based approaches to be combined, if the problem domain demands it.

3.3 Contextual Elements

Contextual elements model information available to a decision strategy for selecting subjective behavior. We have already seen the example above where a message object reifies the meta-information of a communication, to be used by the decision strategy.

Other examples of contextual elements are perspectives, roles or context layers. These abstractions are contextual objects that can affect the decision strategy depending on the subjective model we are in. We can also directly implement behavior in a contextual element, for example to simulate roles. A contextual element can be passed to a decision strategy in two ways: either the decision strategy has direct access or it is sent together with the message.

Figure 2 describes the process of how a decision strategy uses a contextual element to model subjective behavior. We can use contextual elements to model the bank account using perspectives. Bank account subjects send the message `addAndRecord:` through the perspective, modeled as `aMessageContextualElement`. Since `addAndRecord:` was defined as a subjective method, the method behavior is adapted to allow the use of contextual elements:

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

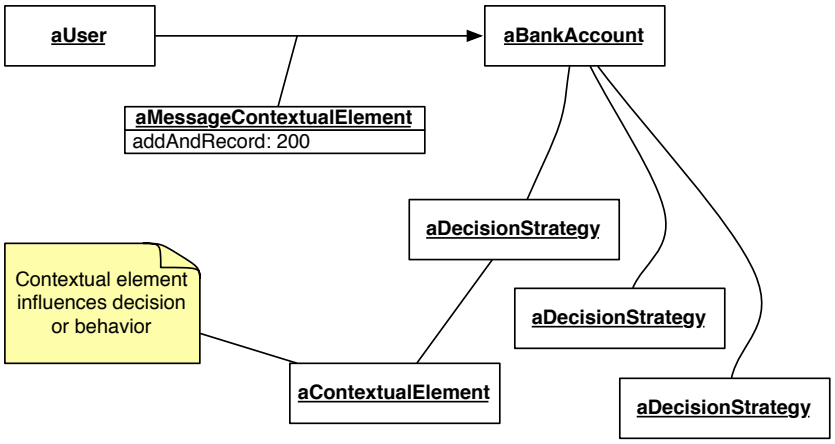

Fig. 2. Two ways of using contextual elements.

Because the decision strategy is modeled explicitly we can change the way the decision is taken. For example we can let the decision strategy automatically determine, using the message object, which perspective has to be used. In this way we do not have to send the contextual element together with the message. In Figure 2 this corresponds to the green contextual element, modeling the perspective, which is directly accessed by the decision strategy.

Sometimes we need composed contextual elements, for example when modeling perspectives. One contextual element models one layer of the perspective. The layers as contextual elements are hierarchically composed to one perspective. The evaluation order of the composed contextual elements is determined by the decision strategy.

3.4 Implementation

The proof-of-concept implementation of Subjectopia is written in Smalltalk, due to its advanced support for run-time reflection. At present, a subject must directly inherit from the class `Subject` to be able register subjective behavior. We transform existing objects to subjects by sending the message `becomeSubject` which adds the necessary behavior to the object receiving the message.

Each subject has a special decision strategy, called *decision meta-object*, which maps subjective message names 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 the bank account subject for the message `addAndRecord:` we send the message `register: aDecisionStrategy`

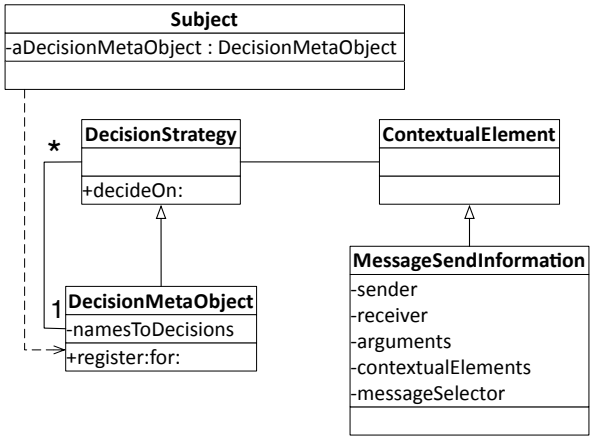

Fig. 3. Class diagram of Subjectopia.

`for: #addAndRecord:`. First, the subject creates an entry with key `addAndRecord:` and value `aDecisionStrategy` in its decision meta-object. Second, the subject generates the following method automatically:

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

The object `thisContext` represents the communication context, which is automatically generated in Smalltalk. If an object sends the message `addAndRecord:` to a bank account subject, it evaluates the decision strategy corresponding to the message `addAndRecord:`.

Prior to the evaluation of the decision strategy, the subject generates an object representing the meta-information of the message with the help of the `thisContext` object. Next, the subject sends the message `decideOn:` to the decision strategy with the meta-information object as argument, which triggers the evaluation. Currently Subjectopia models decision strategies for perspectives, roles and subjective message behavior.

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 will look for the decision strategy corresponding to the message 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.

Consider an object that sends the message `addAndRecord: 200 through: aBankAccountPerspective` to a bank account subject. Since this message is not defined for the subject the `doesNotUnderstand:` method of the class `Subject` will be evaluated. The subject performs a decision lookup to get the decision strategy for the message `addAndRecord:` from the decision meta-object. The contextual

element `aBankAccountPerspective` will be included in the object representing the meta-information of the message. The decision strategy can take the contextual element `aBankAccountPerspective` into consideration available through the meta-information object.

4 Validation

Subjectopia does not force the developer to use a fixed subjectivity model. Because we can choose among different subjectivity models we can model where the subjective decision is taken, whether it is the sender or receiver of the message. In this section we demonstrate this flexibility through four use cases. Two of them were used by previous approaches as examples of subjective behavior. The other two use cases are subjectivity requirements taken from the *Moose* platform for software and data analysis[4].

4.1 Mobile mail application

Let us consider the mobile mail application introduced by Darderes and Prieto [4]. The use case is about having users sending emails from their mobile device. A user can only send emails from his own device. The user collaborates with a mail deliverer which can only send the email if the the device is connected to a network. Otherwise, the mail deliverer retains the email until a connection is established.

The mail deliverer behaves subjectively for the message `deliver: aMail`, as users may only send emails from their own device. The original implementation models the subjective behavior for the message `deliver:` as a force tree associated to the mail deliverer. Our implementation follows the original approach since Subjectopia can model subjective message behavior.

Modeling the `deliver:` message's subjective behavior with perspectives delegates the decision to the sender. However, in reality the user does not choose through which perspective he sees the mail deliverer, but the mail deliverer chooses how to react to the message depending on the context. Modeling this problem domain with perspectives is not natural.

Perspectives are not suitable to model the mail deliverer problem due to the sender-oriented context definition. However, since Subjectopia models the decision taking process explicitly, we can modify it. We can make 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 mails being sent by users.

4.2 Group programming

The group programming application is introduced by Smith and Ungar to explain perspectives [1]. In this use case a system keeps track of all the changes to the

[4] `http://www.moosetechnology.org`. A ready-made image with Subjectopia and Moose can be found at: `http://scg.unibe.ch/research/subjectopia/`

source code of an object-oriented application. We can either see the changes performed by a single developer or the merged changes of several 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 thus we install a perspective decision strategy for the message `getSourceCodeFor:`. A single perspective defines the changes that a developer performs to the system. The changes of the source code for a particular method are modeled as contextual elements which represent layers. The perspectives are modeled as composed contextual elements which are sent by the developer together with the message `getSourceCodeFor:`. The textual representation of the source code is different depending on the chosen perspective.

The group programming use case can be modeled by using a perspective decision strategy with Subjectopia. Other approaches are not well suited for naturally solving this problem domain. For example, Subjective Message Behavior would model changes to the source code as forces. This is not natural because forces influence the behavior of objects 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 want 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 Subjective behavior regarding types of objects in Moose

Moose is a platform for software and data analysis providing facilities to model, query, visualize and interact with data [11,12]. For analyzing software systems, Moose represents the source code in a model described by the FAMIX languages-independent meta-model [13]. For example, the model of the software system consists of entities representing various software artifacts such as methods (through instances of `FAMIXMethod`) or classes (through instances of `FAMIXClass`).

Each type of entity offers a set of dedicated analysis actions. For example, a `FAMIXClass` offers the possibility of visualizing its internal structure, and a `FAMIXMethod` presents the ability of browsing the source code. These actions are renderable as a contextual menu.

A group of entities is modeled through a `MooseGroup`, and is also an entity. Like any other entity, groups can support specific actions as well. For example, a group of `FAMIXClass` can be visualized using a System Complexity View [14], a visualization that highlights the number of attributes, methods and lines of code of classes within a class hierarchy.

We want to solve the problem of offering different behavior depending on the type of the collected entities. As an example we take the subjective behavior for the System Complexity View. When a `MooseGroup` receives the message `viewSystemComplexity` it should only display the contained entities that are of the type `FAMIXClass`. Thus, ideally, we should offer the possibility of viewing the system complexity only if all contained entities are classes.

Our solution models a `MooseGroup` as a subject that behaves subjectively when receiving of the message `viewSystemComplexity`. We separately model the decision, called *decideAvailableActions*, and the behavior, called *systemComplexity*, as decision strategies. The *decideAvailableActions* strategy determines whether the `MooseGroup` has a behavior for `viewSystemComplexity` or not. If a `MooseGroup` contains only `FamixClass` entities, the *decideAvailableActions* strategy attaches the *systemComplexity* strategy. Each time the list of entities is manipulated the decision strategy *decideAvailableActions* recalculates subjectively which actions are available.

Up until now, subjective behavior in Moose is currently realized by subclassing `MooseGroup` (see Figure 4). For example a group of classes is of type `FAMIXClassGroup`, while a group of methods is of type `FAMIXMethodGroup`. Therefore, changes in the list of entities can result in a change of the runtime type of the group. The decision which type to choose for a given group is currently implicit and it is based on names. For example, we cannot easily introduce a decision of defining actions for a mixed group containing both classes and methods.

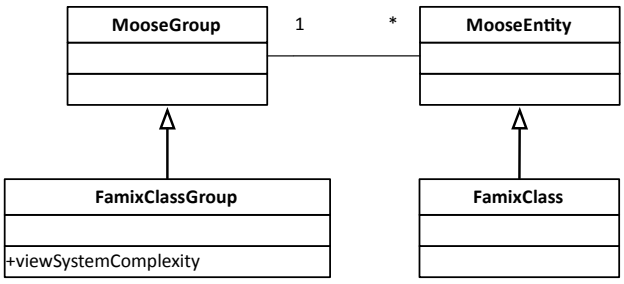

Fig. 4. Current class hierarchy of Moose elements.

Using our approach, we extend `MooseGroup` to implement subjective behavior, without depending on the class hierarchy. We simply change the *decideAvailableActions* strategy to decide the new case and model the new behavior as a decision strategy.

Moose is a large system with many extensions defined on top. Thus, any change to the core should limit the impact on the other parts. This would imply significant effort with other subjectivity approaches. For example, using COP would have implied to translate a large part of the system to layers which entails a considerable engineering effort. The subjective behavior is influenced by the elements contained in the `MooseGroup`, thus we would need to define an activation protocol for the layers. Splitting the contextual behavior of `MooseGroup` into

several layers also implies a high effort because of the shared behavior between the different kinds of groups.

4.4 Subjective behavior depending on the Moose environment

Moose provides a generic graphical user interface to interact with the model of the software system. In Figure 5 the `MooseGroup` entities of the model are listed. 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*. By selecting a menu entry a message is sent to the selected group. For example, selecting *Visualize* → *System complexity* sends the message `viewSystemComplexity` to the selected `FAMIXClassGroup`.

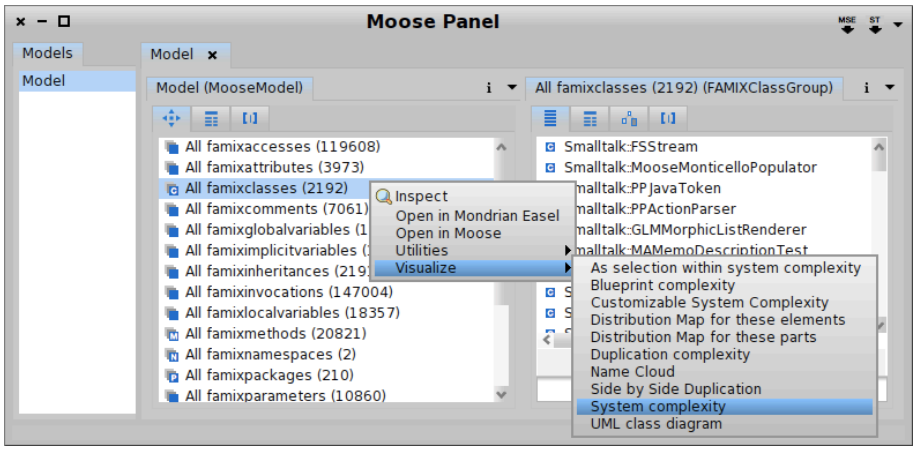

Fig. 5. User interface provided by Moose. Selecting the entry `System Complexity` results in sending the message `viewSystemComplexity` to the selected group of classes.

The problem is that some visualizations may require contextual information not retrievable from the objects and subjects involved in the communication. Let us consider that we select a group of classes and that we want to view them as highlighted on the overall system complexity. This can be achieved by sending the message `viewAsSelectionOnSystemComplexity` to the group. This behavior also requires *all* other `FamixClass` entities of the model to create this visualization. However, in different analysis contexts we want to see only a subset of all classes as a basis for the visualization. Thus, the simple action of `viewAsSelectionOnSystemComplexity` requires both the receiving group and the reference group. Moose currently uses model-wise global variables to store this information. The problem is that each new instance of the graphical user interface of Moose can override the value of that global variable and this results in unwanted side effects.

Our solution 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 use a decision strategy modeling the behavior for the message `viewAsSelectionOnSystemComplexity`. The decision strategy has access to the contextual elements 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 other approaches cannot. For example, using roles would 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`, whereas it is the `MooseGroup` that determines its roles.

5 Conclusion

In this paper we have presented Subjectopia, a unified approach to modeling subjectivity. Specifically, our contributions are the following:

1. We surveyed prior work and identified a lack of generality when modeling different problem domains.
2. We presented a novel approach to subjectivity that explicitly models subjects, decision strategies and contextual elements. The reification of these abstractions avoids the need to impose on the developer a particular paradigm for modeling subjective behavior.
3. We developed a fully working prototype of the Subjectopia system and presented the implementation of non-trivial subjective use cases.
4. We demonstrated that our approach can model all other existing subjective approaches as well as new, customized strategies. Moreover, we showed that other approaches cannot model all use cases while our approach can adapt and represent them all.

Introducing subjective behavior in legacy applications might have a considerable impact on the overall behavior of the application. Being able to scope the subjective changes to specific objects helps in controlling this impact. We plan to analyze reflection frameworks to allow Subjectopia to perform object-specific subjective adaptations.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Synchronizing Models and Code” (SNF Project No. 200020-131827, Oct.

2010 – Sept. 2012). We thank Tudor Gîrba for providing the Moose case studies which greatly helped in the development of Subjectopia. We thank Tudor Gîrba, Lukas Renggli and Fabrizio Perin for their feedback on earlier drafts of this paper. We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

1. 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
2. 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
3. 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
4. Darderes, B., Prieto, M.: Subjective behavior: a general dynamic method dispatch. In: OOPSLA Workshop on Revival of Dynamic Languages. (October 2004)
5. 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
6. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. Volume 22. (December 1987) 227–242
7. 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
8. 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
9. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology **7**(3) (March 2008)
10. 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
11. Gîrba, T.: The Moose Book. Self Published (2010)
12. 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.
13. 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
14. Lanza, M., Ducasse, S.: Polymetric views—a lightweight visual approach to reverse engineering. Transactions on Software Engineering (TSE) **29**(9) (September 2003) 782–795