Modeling Variability in Design Patterns using ACL

by

Philip Eagan

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Computer Science

at

School of Computer Science
Carleton University
Ottawa, Ontario

Abstract

Design patterns were created to promote reusability of solutions to problems found in software design. Unfortunately, the absence of a systematic approach to the categorization and organization of design patterns has hindered overall reusability by allowing several catalogues to overlap. In this thesis, I propose two approaches that use Another Contract Language (ACL) for the specification of such patterns. My goal is to demonstrate that it is feasible to capture the variability inherent in design patterns while producing a testable model. My two modeling strategies are compared with two published, non-ACL approaches in a case study on the well-known Observer pattern. This case study illustrates the benefits of my proposed strategies in dealing with pattern variability *and* scenario capturing.

Acknowledgements

I would like to thank all the members of my family. For it is through their encouragement and support that I was able to get through and complete this thesis.

I would also like to thank my thesis supervisor, Dr. Jean-Pierre Corriveau, whose guidance and support has helped me throughout the preparation and the writing of this thesis. Finally, I also want to thank my co-supervisor, Professor Wei Shi, for her help and funding throughout this research.

Table of Contents

List of Figures

List of Appendices

1 Chapter: Introduction

1.1 Context of Thesis

The use of design patterns is common in software development, but what exactly is a design pattern? Christopher Alexander in his seminal work on design patterns states, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" (Alexander, et al., 1977, p. x; Alexander, 1979). Essentially, Alexander argues that a design pattern is a solution to a common problem that occurs in a specific context that has the goal of increasing reusability. However, in this thesis, I do not address such context. The concept of design patterns was further popularized with the work of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their book "Design Patterns Elements of Reusable Object-Oriented Software" (1994). The authors of this book are commonly referred to as the 'Gang of Four' (hereafter Go4). Their foundational work, which details a series of patterns (also called a pattern catalogue), has been cited 4,586 times (ACM Digital Library, 2014), illustrating the importance and fundamental nature of their work in the field of software development. I will assume the reader is familiar with it.

Design patterns are important to the field of computer science because they help promote reusability of software and speedup development. The use of an existing pattern, whether it is a Go4 pattern, some other design pattern or another kind of pattern (e.g., analysis, testing, etc.), speeds up the development process of software because the solution has already been tried and tested to solve the problem (Bishop, 2007).

The patterns specified by the Gang of Four in their book cover three different types of patterns: Creational, Structural, and Behavioral. Each of the three types of patterns group together for a specific aspect of object-oriented software development to form highly reusable solutions for small frequent problems relevant to that aspect. Creational patterns present solutions to the problem of how classes create objects. Structural patterns address class organization for a specific problem by showing how the relevant classes are connected to each other and with abstract classes and/or interfaces. For example, the structure of a *proxy* contains a class that works as a placeholder representing another class or object when communicating with other classes in the piece of software. Lastly, behavioral patterns focus on how sets of classes behave. For example, a *mediator* handles the communication between a set of classes so that any two of these mediated classes do not have to know anything about each other. Instead, all communication is performed indirectly through a mediator.

As is implied in Alexander's definition of a design pattern, there are innumerable design patterns in existence. Unfortunately, the absence of a standardized specification technique for design patterns has led to the proliferation of catalogues in which 'claimed' patterns are, in fact, mere (possibly overlapping if not redundant) variants of a genuine design pattern (that is, I repeat, a reusable key idea at the core of a space of solutions). The lack of a systematic approach to the categorization and organization of design patterns has worsened the problem (and has hindered overall reusability) by allowing several catalogues to overlap. A general question to ask is how the reusability of design patterns can be improved through better specification, but 'better' with respect to what? Moreover, how would one demonstrate 'improved reusability' without carrying multi-

variable experiments involving a large number of software developers (which are typically not realizable within the scope of a Master's thesis). We need to scope this general problem.

1.2 Problem and Thesis

When looking at any design pattern, it is important to conceptualize it, I repeat, as *a space of solutions*: a pattern includes a core idea of a solution and some *variability* in the realization of this core idea. The concept of variability[1] is not new or unique to design patterns. The concept of variability can be seen as far back as Alexander's original work on patterns (Alexander, et al., 1977; Alexander, 1979). Alexander states that the use of design patterns is analogous to the individual components that make up a cityscape, from the small windows on a house to the high points of the city. These individual components are just the building blocks. Then someone who is making a house or some other project can pick and choose from these blocks to create their house or city. This customizability of Alexander's patterns instantly introduces variability into his patterns.

This notion of variability is inherent to the Go4 design patterns. For example, Freeman et al. (Freeman, Freeman, Bates, & Sierra, 2004) introduce two variants for Go4's Observer pattern. Both versions have the Subject notify all registered Observers when it changes. However,

[1] Variability refers to the differences (in structure or behavior) that can occur inside of a design pattern when it is implemented while maintaining the essential parts of the pattern.

1) in the *pull* variant, which is the one described in Go4, each observer then must ask the subject to send it the information that concerns this specific observer, whereas

2) in the *push* variant, as it notifies its Observers, the Subject includes in this notification its complete state, leaving each observer to retrieve from that state whatever information is relevant to it.

Two immediate remarks are in order: a) these two variants are not the only possible variants for the Observer pattern and b) these two variants can be applied to the Mediator pattern as well (for which many other variants have been published).

Variability in design patterns is certainly not limited to the Go4. Consider, for example, the catalog of patterns presented by Buschmann et al. in their book "Pattern-Oriented Software Architecture" (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996). One of the patterns they discuss is the well-known Model-View-Controller (MVC) pattern that allows multiple variations in its design, ranging from a small variation in which views are either created dynamically or statically, to more significant variation. For example, the authors mention one such major variant in their book, namely the "Document-View" in which the separation between the view and controller classes is relaxed to the point that these classes share responsibilities between each other (*Ibid*.). Another major variant of the MVC consists of using controllers organized into one or more hierarchies.

Variability is found even in antipatterns (i.e., patterns capturing bad designs to be avoided) (Brown, Malveau, McCormick, & Mowbray, 1998). For example, the Blob antipattern is found in designs where one class monopolizes the processing, and other

classes primarily encapsulate data. One specific variant of the Blob is known as a 'god-object', that is, an object that sets the state of other objects (thereby violating encapsulation). More general definitions of a 'god-object' view it as an object that knows too much or does too much, a formulation that overlaps with the one of a Blob. As an application grows, if it adopts an MVC architecture, typically its controllers will grow to the point they may be viewed as blobs. Thus, considerably different designs can 'fall under' the same antipattern, as is also the case for design patterns.

Advocates of Generative Programming (see section 2.1.4) have demonstrated that modeling variability promotes reusability. This is the postulate on which this thesis rests. My research question then could be: how to model the variability inherent to design patterns? This is still too general a question inasmuch as criteria to evaluate a modeling notation can be quite contentious. I need to further refine it. To do so, I observe that Generative Programming uses *features diagrams* as its modeling tool for capturing variability in programs. An example of a feature diagram for a security profile is given in Figure 1 and is from the paper "Staged Configuration Using Figure Models" (Czarnecki, Helsen, & Eisenecker, 2004). Without going into details, this model captures a security profile consisting of two features: the password policy and the permission set. Each feature is specified in terms of subfeatures or sets of possible values.

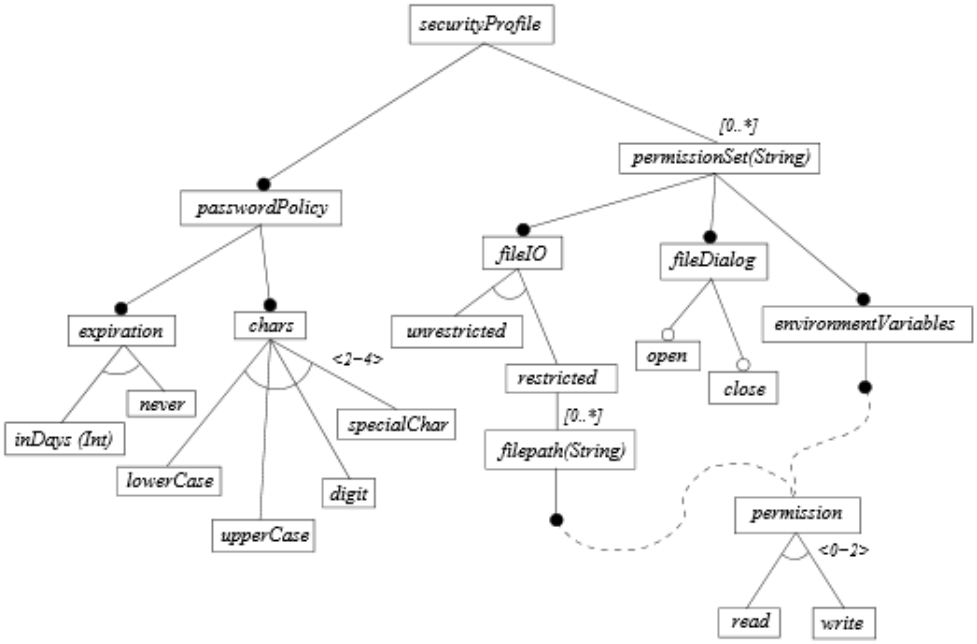

Figure 1: Feature Diagram Example (Czarnecki, Helsen, & Eisenecker, 2004)

To the best of my knowledge, feature diagrams have not been developed for design patterns (except for the initial work of Radonjic et al. (Radonjic, Bashardoust, Corriveau, & Arnold, 2011) on creational patterns). The reason is simple: they capture variability at a very abstract level and are purely descriptive. That is, it is not obvious what constitutes a 'good' feature model but, more importantly, such a model is disconnected from code (in which a design pattern is ultimately realized). Consequently, as a specification, this model is not *testable* (in the sense Binder (1999) uses for this word): no tests can be generated from a feature diagram. The inadequacy of feature diagrams for capturing variability in design patterns in a verifiable way leads to the specific research question this thesis asks, namely: is it feasible to model the variability

of design patterns in a way that would enable some form of verification[2] of this specification against an actual implementation?

The research question tackled in this thesis is whether it is feasible to both model the variability that occurs in design patterns as well as verify such patterns against an implementation of a design pattern variant. It is my thesis that by using Another Contract Language (ACL), a contract language created by Arnold and Corriveau (see section 2.2), augmented with generative syntax, it is possible to capture the variability inherent in a design pattern *and* 'check compliance'[3] of an implementation under test (hereafter IUT) with this specification.

This thesis tackles the issue of pattern verification and not pattern detection. That is, the proposed solutions are for verifying the presence of a design pattern in an implementation and not for searching for the presence of patterns in a system. This is due to the nature of the tool used and as such there is no need to worry about false positives in the context of the proposed approach.

1.3 Methodology

In order to support my thesis, I will present (in this thesis and on its associated website) several examples of how different variants of some Go4 patterns can be captured in ACL and *augmented ACL* (hereafter aACL, to distinguish it from the existing

---

[2] By 'verification' I am referring to the ability of having i) a specification for a variation of a design pattern and ii) an implementation to be tested, that are bound together and checked against each other

[3] What I mean by 'checking compliance' will become apparent as I discuss ACL at length in the next chapter (section 2.2). How compliance is checked in the two proposed approaches is described in Chapter 3 in sections 3.1.4.1.4 and 3.1.4.2.4.

ACL/VF (found at vf.davearnold.ca) system. I focus specifically on some of the Go4 patterns because this catalogue is not only widely reused but also because a) variants of some of its patterns have been explicitly discussed, and b) several Go4 patterns constitute the building blocks of so-called 'compound' patterns such as the MVC (Freeman, Freeman, Bates, & Sierra, 2004) and Douglas Schmidt's patterns for concurrent and distributed systems (Schmidt, Stal, Rohnert, & Buschmann, 2000). My work will not only contribute models but also an in-depth comparison of each selected design pattern with two other approaches, namely the one of Elaasar (see section 2.1.2), and the one by Jézéquel (see section 2.1.3). Elaasar's approach was chosen because it is a descriptive approach to the Go4 patterns that tackles the issue of variability, while the approach of Jézéquel was chosen because it is a modeling approach to the Go4 patterns that does support a form of correspondence checking between the specification and the code realizing a pattern.

For this thesis there are two approaches proposed for the modelling of the Go4 design patterns. One is referred to as the "Refine Approach" and the other as the "Generative Approach," both of which are expanded upon in Chapter 3. In the proposed Generative approach I have made additions to the ACL syntax in order for it to allow the Generative Programming principles that are discussed in Chapter 2 to be followed in a specification. One such addition I made to the ACL syntax is the use of a statement inside of square brackets that is placed before observabilities and responsibilities. This is discussed in sections 2.2.1.4 and 2.2.1.5 respectively. Another addition of mine for the Generative Programming Approach is the use of a Parameters block and Choice statements inside interaction classes, as can be seen in sections 2.2.2 and 2.2.2.1.

1.4 Overview of the Thesis

Chapter 2 will provide an overview of techniques related to the specification of design patterns, as well as a more in-depth look at the techniques developed by Elaasar and Jézéquel. This chapter will also discuss Generative Programming and the tool that will be used for this thesis, ACL. In particular, an example of ACL running in its Validation Framework (hereafter VF) will be provided to show how it works. In Chapter 3, I will systematically compare my proposed approach with the ones by Elaasar and Jézéquel through the use of the Observer design pattern and provide a summary of my modeling for the Mediator, Factory, and Memento patterns. Chapter 4 contains the conclusion to the thesis.

2 Chapter: Related Work, Generative Programming and ACL

This chapter will present a review of work related to the topic of this thesis. In particular, there will be a general overview of the approaches taken by Elaasar and Jézéquel. The ACL language will be presented, and an example of an ACL contract running in the Validation Framework will be provided.

2.1 Related Work

In this section, a general overview of the available related approaches will be provided. Among the topics covered are the two approaches being compared with the thesis's proposed technique, as well as an overview of the Generative Programming paradigm.

2.1.1 General Overview of Other Approaches

While completing this literature review, a variety of approaches for the detection and representation of design patterns in software systems were examined. There has been a significant amount of research done in this area, which has resulted in a number of techniques proposed for the detection of design patterns. The more relevant of these varied approaches are reviewed below.

2.1.1.1 Diagram Specification Approaches

One technique for the specification of patterns makes use of Unified Modeling Language (UML) diagrams. In this approach, France *et al.* propose the use of a more complex class diagram combined with structure and sequence diagrams as a way of presenting the design patterns (France, Kim, Ghosh, & Song, A UML-Based Pattern

Specification Technique, 2004). They do tackle variants of a pattern through the making of a diagram for each variant. While they do allude to the end goal of being able to somehow verify the patterns using these diagrams, this goal is only included as part of their future work. That is, for now, they only model patterns with no concern for a subsequent verification step, making their work less relevant to my goal.

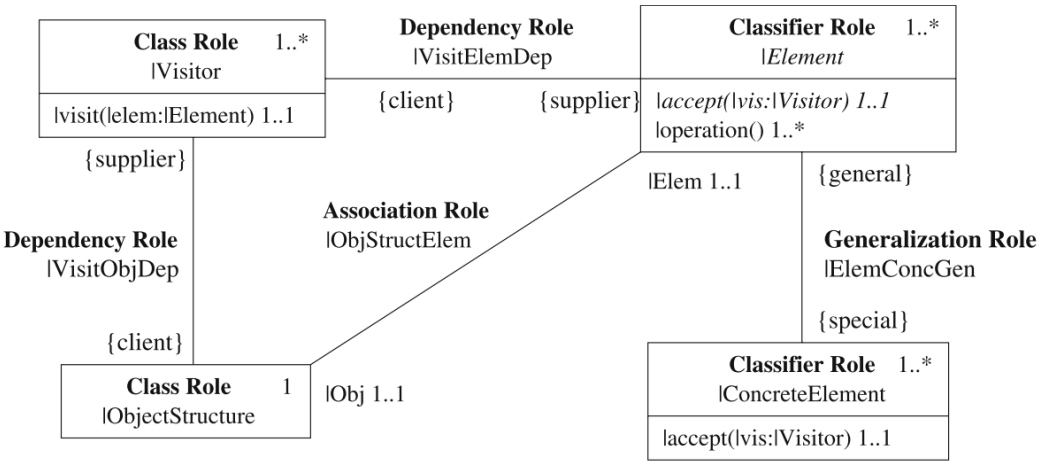

Figure 2: Static Pattern Specification (Kim & Shen, 2008)

A second approach for design pattern specification that uses a UML based approach is one by Kim and Shen (Kim & Shen, 2008). They used a language called Role-Based Meta-modeling Language (RBML) in order to specify the solution domain. RBML was presented by France *et al* in their 2004 paper (France, Kim, Ghosh, & Song, 2004). In this approach, these authors use a particular RBML pattern specification called Static Pattern Specification to represent the structure of a design pattern. An example of a Static Pattern Specification can be seen in Figure 2. This approach takes this Static Pattern Specification and uses it to determine the presence of the pattern in a UML Class diagram. Again, verification is not addressed.

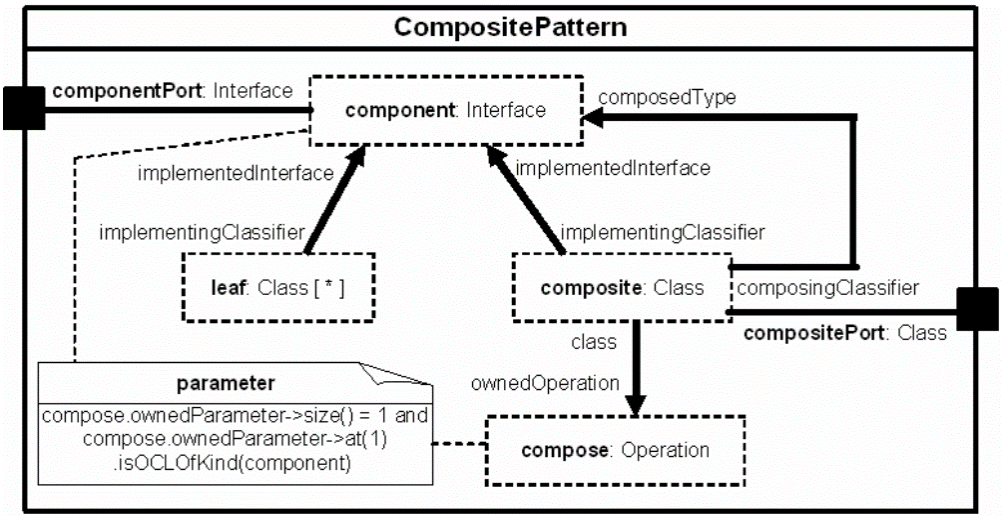

Figure 3: EPattern Specification Example (Elaasar, Briand, & Labiche, 2006)

In early work on this problem, Elaasar, Briand, and Labiche used their Pattern Modeling Framework as a metamodeling approach for pattern specification. More specifically they use EPatterns to represent patterns. An example of an EPattern can be seen in Figure 3. The EPattern diagram contains all the relevant information about the pattern, from variables to methods, to how the classes relate to each other. The goal of their approach was to create an algorithm that would use these pattern models to detect their existence in user models (Elaasar, Briand, & Labiche, 2006). Elaasar would go on to use a similar approach in his Ph.D. thesis for detecting patterns, but would use a different type of diagram to represent the patterns. This will be discussed in section 2.1.2.

In summary, there are a variety of approaches for *representing* design patterns in UML. (e.g., France et al. (2004), Kim and Shen (2008), Le Guennec et al (2000), and

Mak et al. (2004)). However, most of these approaches currently are not testable[4] and in many cases do not address how to capture variability. While there were a few tools that would identify patterns inside of UML diagrams (e.g. Kim and Shen (2008) as well as Le Guennec et al (2000)) in the diagrammatic approaches I reviewed there was only one approach that provided a tool for taking their model and verifying it against a realization of the pattern in code, that approach is one by Elaasar and it is discussed in section 2.1.2 of this chapter. There were a couple of approaches that provided a tool for checking that a pattern was present in a UML diagram, these approaches were the ones by Kim and Shen (2008)

2.1.1.2 Code Specification Approaches

There are techniques available that represent patterns through a code specification and, in some cases, use that representation to verify that a pattern is present in a given piece of software.

```
p_singleton(Singleton):-
%Structural Aspect
   association(Singleton, Singleton, UniInstance,'1'),
   is_static_attr(UniInstance),
   dependency(Singleton, Singleton, GetInstance,
   'depReturn'),
   is_static_op(GetInstance),
%Behavioral Aspect
   has_unique_instance(Singleton).
```

Figure 4: Prolog Representation Example (Huang, Zhang, Cao, & Duan, 2005)

[4] By 'not testable' I am referring to a method that cannot use its diagrams to verify the existence of a pattern in a given implementation and/or does not offer a tool to do so.

For example, Huang *et al.* provide a representation of design patterns in Prolog (see Figure 4 for an example). Then using a tool that they developed called “PRAssistor," they can recover patterns from an existing piece of software (Huang, Zhang, Cao, & Duan, 2005). While the use for this technique is primarily focused on recovering patterns from legacy code it is also used for the representation of patterns in code. As well, it does suffer from a lack of precision as false positives are frequently detected.

Another approach that uses a code representation is the one developed by Kramer and Prechelt in 1996. Their approach detected the design information directly from the C++ header files and stored it inside a repository. Kramer and Prechelt expressed the design patterns using Prolog rules to query the repository in which they had put the design information that they had obtained from the C++ header files. Using their approach, they had a 14 to 50 percent success rate of detecting Adapter, Bridge, Composite, Decorator, and Proxy patterns, which they believed to be acceptable (Kramer & Prechelt, 1996).

```
C := ∅
for each class c do
  Y := ∅ // intermediate result
  for each variable v in c do
    for each class u := class(v) do
      if c=u ∨ c ⊂ u
        Y := Y ∪ {(c,v,u)}

for each (c,v,u) ∈ Y do
  for each method m in u do
    if "… v.m …" in body(m) in c
      C := C ∪ {(c,v,u,m)}
```

Figure 5: Static Analysis of Chain of Responsibility (Heuzeroth, Holl, Hogstrom, & Lowe, 2003)

A third approach in this category is the one of Heuzeroth et al. (Heuzeroth, Holl, Hogstrom, & Lowe, 2003). They created a tool that does a static analysis as well as a dynamic analysis on legacy code to determine the presence of patterns in that code. The static analysis was used to look for the structure of the patterns. An example of the code they used for the static analysis of the chain of responsibility pattern can be seen in Figure 5. The dynamic analysis was done during the execution of the legacy code to look for the behaviour caused by the different patterns, while paying close attention to the possible pattern matches that were found during the static analysis of the code. Through the use of their tool, they produced several possible candidates for design patterns in the legacy code, which they had to manually verify. Beside this major drawback, there were also some other minor problems with their approach.

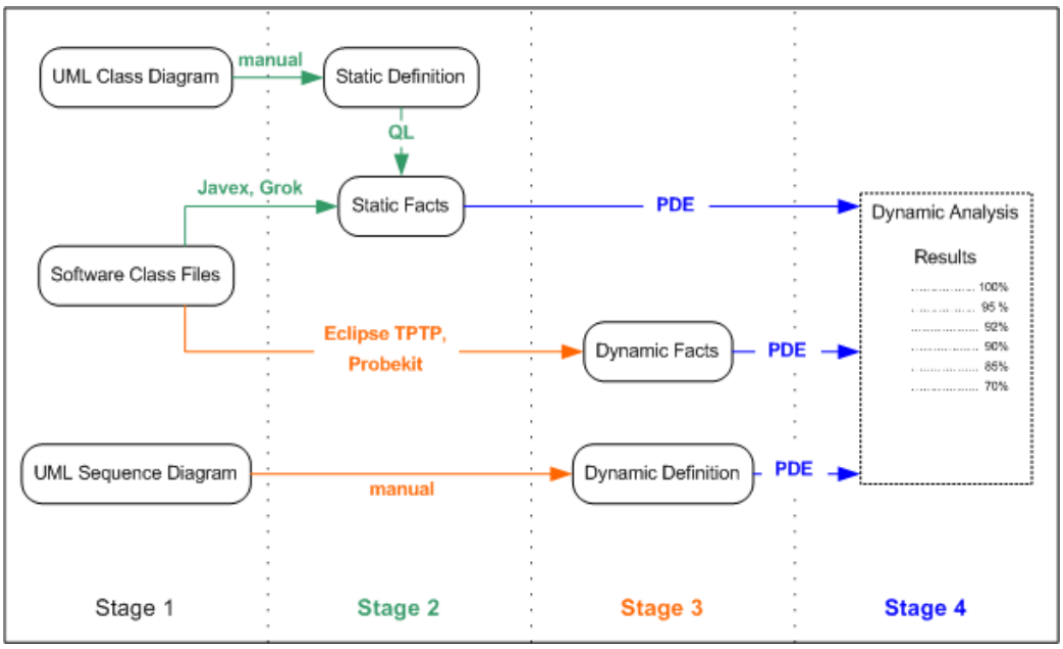

Figure 6: Overview of Steps in Approach by Birkner (Birkner, 2007)

For his thesis, Birkner developed a tool for the sole purpose of identifying patterns in code. The steps of his technique can be seen in Figure 6. While similar to the

previous approach, Birkner examined the usage of static and dynamic checks as a way of better identifying design patterns. Through the use of other tools, Birkner converted Software Class Files and a Static Definition created from UML diagrams into Static Facts that were then inputted into the tool. He also used yet another tool to convert the software class files into a set of dynamic facts to be inputted into his tool. These facts were represented using XML. His tool then proceeded to identify the patterns present in the software based on these facts. However, during experimentation, Birkner found that false positives were possible and that many patterns were missed when the tool was looking for 100% accuracy when checking against the predefined facts about the pattern (Birkner, 2007).

2.1.1.3 Graphical Specification Approaches

In this subsection, I will look at approaches that represent and identify patterns through the use of graphs for representing both the system and the pattern. An important part of these approaches is the method they use to determine how they will convert both the patterns and the software being examined into graphs.

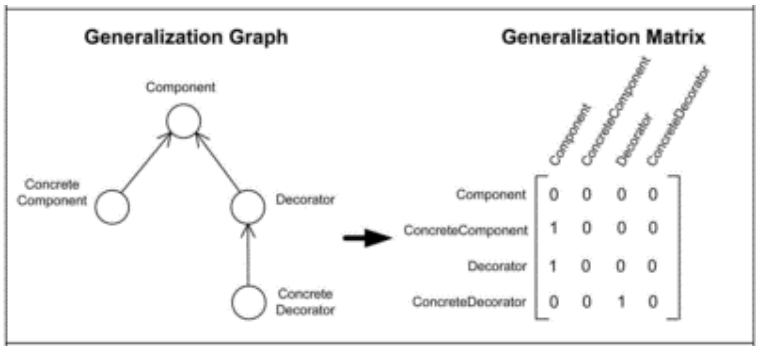

Figure 7: Representation of Pattern Structure as Graphs and Matrices (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006)

For example, the approach by Tsantalis et al. converts a pattern and the system being checked into graphical representations that show the relations between classes (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006). They then use an algorthim that compares the graphs of the system with the graphs of the patterns in order to determine if a close representation of a pattern can be found. An example of their graphical representation can be seen in Figure 7. This enables them to detect variations of patterns, but can lead to false positives.

A second graphical approach is advocated by Pettersson and Löwe (Petterson & Löwe, 2006). In this approach, they first need to take a graphical representation of both the pattern and the software being examined. They then suggest getting the contents of the graph related to the pattern down to the bare minimum. In this case, bare minimum refers to certain aspects of patterns that are deemed to be unnecessary and thus are removed. A year later, the same authors then decided to include aspects that they had deemed unnecessary in their first approach (Pettersson & Löwe, 2007)!

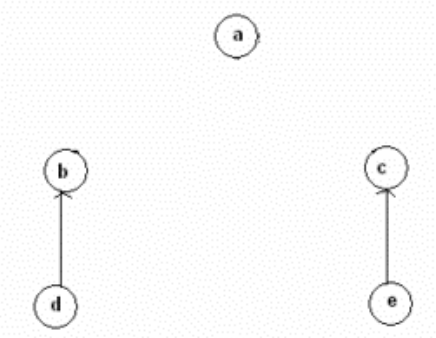

Figure 8: Generalization Relationship Graph for Abstract Factory (Gupta, Pande, & Tripathi, 2011)

SOP(gen1) = d.b	SOP(gen 2) = e.c	3(b)
SOP(d.a.) = a.b + a.c		3(a)
SOP(dep) = d.e		3(c)

Figure 9: Sum of Product Expression for Abstract Factory (Gupta, Pande, & Tripathi, 2011)

Gupta et al. proposed a graphical approach in their paper (Gupta, Pande, & Tripathi, 2011). The generalization relationships of the classes of the Abstract Factory pattern in graph form can be seen in Figure 8. This graph is then converted into a sum of product (SOP) form of Boolean expressions. The SOP expressions for Abstract Factory is given in Figure 9. In their approach, with both the system being reviewed and the design patterns converted to a sum of product, they can look for the occurrences of design patterns in the software by comparing the two SOP expressions. One of the limitations they identify in this approach is the inability to identify any pattern that has a relationship with itself, such as the Singleton pattern. But, in fact, as with other reviewed approaches relying on 'conversions', there are two major difficulties:

1) conversions do not guarantee semantic equivalence between the source and the target. In fact, a (typically significant) loss of information is frequent if not unavoidable.
2) The target representational schemes are not only very far from the level of conceptualization found for design patterns in Go4, they also generally do not address variability.

In fact, none of the approaches reviewed up to this point directly address the issue of variability. Thus, we will now move on to the two approaches I have selected for comparison with my proposal.

2.1.2 Elaasar's Approach

In his Ph.D. thesis, Maged Elaasar sought to create a new unified approach to detecting a couple of types of patterns for MOF-based modeling languages. In Elaasar's approach, he models a pattern using the Visual Pattern Modeling Language (VPML) and then detects the pattern's occurrence in a model with a corresponding QVT-Relations (QVTR) transformation (Elaasar, 2012). The key feature of this method, which motivates my decision to compare it to my own, is the type of approach it uses: it is descriptive and it deals with variability. Throughout this subsection I will go over the details pertaining to how Elaasar is modeling the Go4 patterns.

2.1.2.1 Modeling a Pattern in VPML

In order to model a pattern, Elaasar uses VPML to represent such a pattern visually. VPML is a language he designed for his thesis, and the concepts and relations defined in it come directly from the pattern domain (Elaasar, 2012). In this section, his representation of the Go4 Adapter Pattern will be used to review how he represents a pattern and its variants in VPML. I will assume the reader's familiarity with these Go4 patterns.

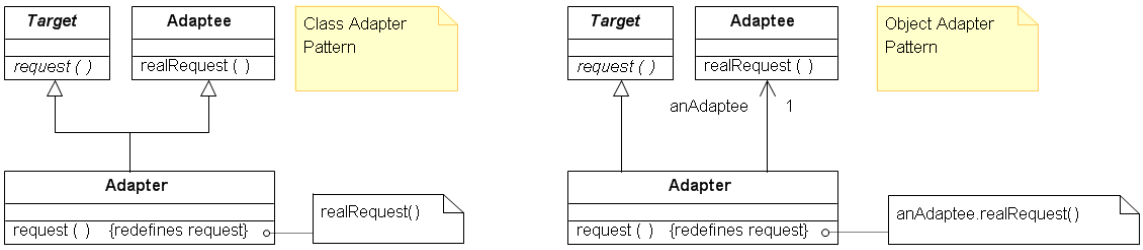

Figure 10: Adapter Pattern and Variation (Elaasar, 2012)

Elaasar uses the UML representation of the pattern seen in Figure 10; this UML diagram contains the two variants that Elaasar looks at for the Adapter pattern. The VPML representation of the pattern and its variants can be seen in Figure 11. The VPML diagram contains representations for classes, operations, and properties, all of which are referred to as *roles*. A role is represented as a box. Inside the box on the right side of the colon, one can find the role's type and its name can be found on the left side of the colon. If a box is made with a solid line, it means that it is visible. If a box is made with a dashed line then, it is invisible. The relationships between roles are represented through the use of arrows. Inside the diagrams, there can be an idiom. These idioms are represented through the use of an oval. Idioms are an action that occurs routinely inside of a class. The idioms covered by Elaasar, that are shown in this thesis, are ones for "Redefinition," "Delegation,' and "Conglomeration" and are discussed in upcoming paragraphs. The separation of the idioms from the VPML diagrams is done to increase readability.

Elaasar's approach allows capturing variations in the structural aspect of the patterns. To describe the variations, he creates a diagram for the base representation of the pattern and then a diagram for each variation (or equivalently, variant). The diagram for the variants both alters existing roles and adds new roles to the base diagram to show what is different in each variation. In the VPML diagrams for the variants, Elaasar allows for the introduction of new classes, properties, operations, and idioms, as well as for different relationships between the roles. The variations are marked by their name in the top left corner of their diagram. The name to the right of "->" is the diagram that the variation pattern is being applied to.

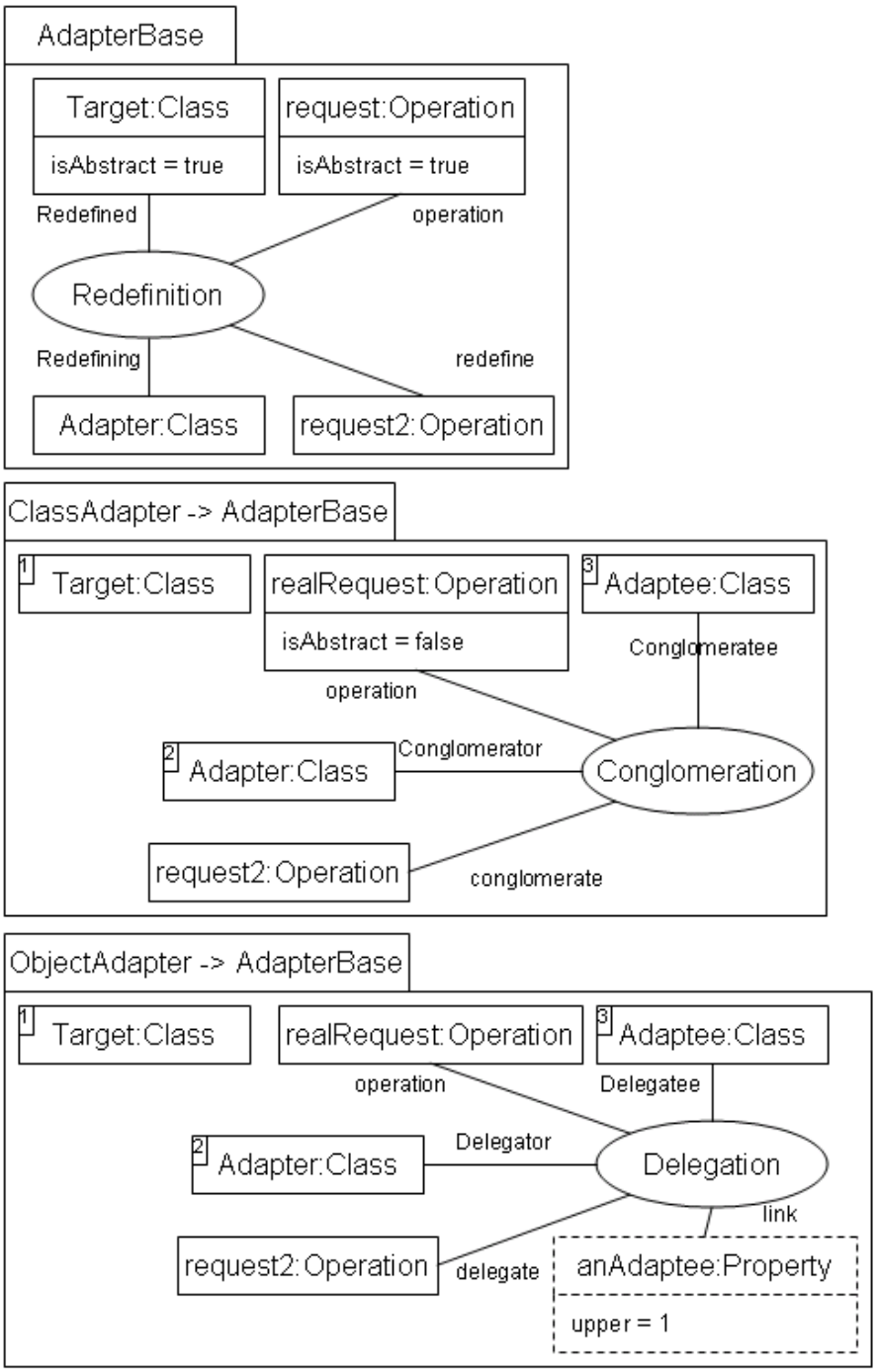

Figure 11: Adapter Pattern modelled using Elaasar's approach (Elaasar, 2012)

At the bottom of the VPML representation, there is a place for conditions that are to be placed on the pattern. VPML allows for limited support of scenarios. Scenarios refer to a series of actions that are to be completed in a specified order. In this approach, the use of scenarios is extremely limited and can only show a simple series of events. An example, which can be seen in Figure 12, shows that when the delegate operation is called, that it is called through a link.

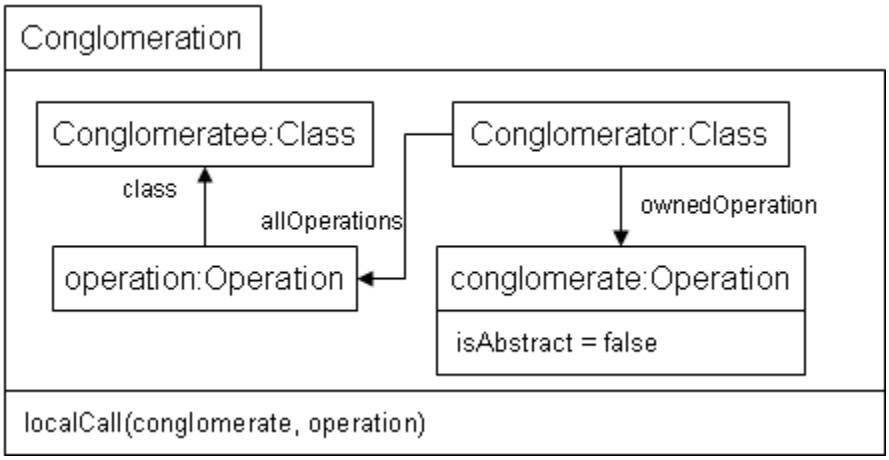

Figure 12: Elaasar's Conglomeration Idiom

In Elaasar's representation of the Adapter pattern, we can see that in the diagrams for the variants both of them add an Adaptee class and an operation called "realRequest" while the Object Adapter adds an invisible property named "anAdaptee." Another difference between the two patterns is that the Class Adapter uses the conglomeration idiom, whose details can be seen in Figure 12, and the Object Adapter uses the Delegation idiom. The Conglomeration idiom adds the operations from the Adapter class to the Adaptee class.

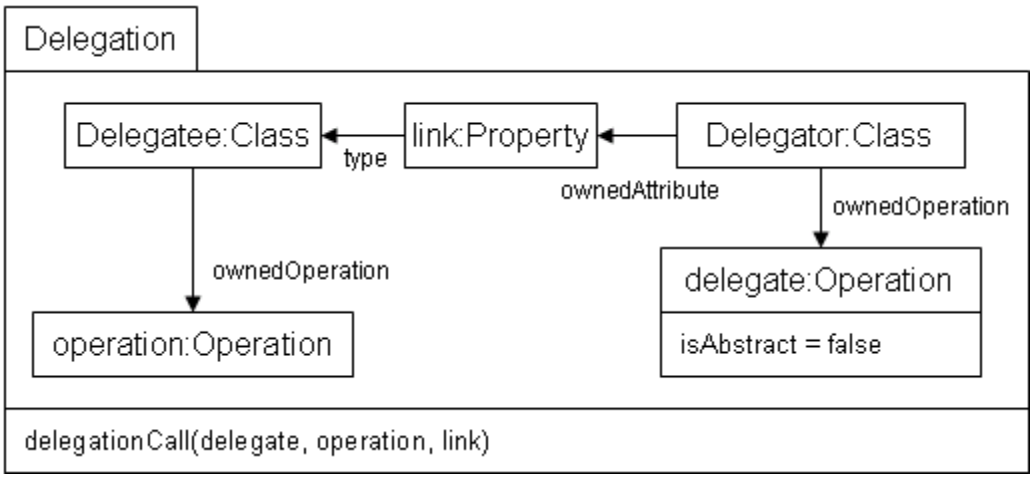

Figure 13: Elaasar's Delegation Idiom

Elaasar's representation of the ObjectAdapter variant makes use of the Delegation Idiom. The Adapter class contains a property named "anAdaptee." This property is a link to the Adaptee class and is denoted as such in the metadata detailing its relationship. The Adapter class is described as the delegator and the Adaptee class as the delegatee while the "request2" operation in the Adapter class is listed as the delegate. The oval in the diagram for the variant connects a group of roles and works as a placeholder for the Delegation idiom. Figure 13 illustrates how the delegation idiom works and how the components of the idiom relate to each other. For example the role "Delegatee:Class" would read as "Adaptee:Class," "link:Property" would read as "anAdaptee:Property," "operation:Operation" would read as "realRequest:Operation," and "delegate:Operation" would read as "request2:Operation" after the delegation idiom is applied to the strategy pattern. The purpose behind this delegation idiom is to show, in VPML, where there is an operation that is to be replaced by an operation in another class.

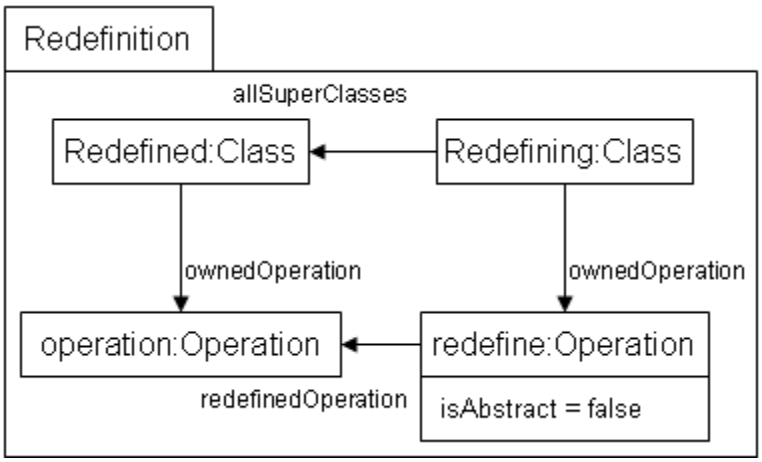

Figure 14: Elaasar's's Redefinition Idiom

In the base diagram for the adapter seen in Figure 11, there is another placeholder. This placeholder is for a Redefinition idiom. The Redefinition idiom occurs where there is a method that replaces a method in another class at run time. In the VPML diagram, it happens between the Target and Adapter classes. The method "request2" of the Adapter class replaces the one in the Target class. A diagram of this idiom can be seen in Figure 14.

2.1.2.2 Modeling Patterns using QVTR

In order to compare his representation to a piece of software, the VPML diagram of a pattern needs to be transformed into a QVTR[5] representation of the same pattern. In his thesis, Elaasar provides the information on how the VPML maps to the QVTR representation. He uses the metadata from the VPML diagram and uses a tool to convert

[5] QVTR (Query/View/Transformation-Relations) is a standard of the Object Management Group (OMG) for defining transformations between MOF-based models. More information can be found on the website http://www.omg.org/spec/QVT/1.1/

them to a QVTR representation of the pattern. Once he has the diagrams converted to QVTR he needs the software being examined to be converted to a QVTR representation. When this is done, the two can be compared, and the patterns searched for using Elaasar's approach.

2.1.2.3 Summary of Elaasar's Approach

The basic steps behind Elaasar's approach to the specification and verification as described above are: Step 1: convert the UML diagram into a VPML representation of the pattern. Step 2: convert the VPML diagram into a QVTR representation of the pattern. Step 3: convert the system being verified to a QVTR representation of the system. Step 4: run the comparison algorithm to check for occurrences of specified patterns in the software. In Elaasar's case study on the detection of Go4 patterns, his approach precisely identified patterns an average of 72% of the time. However, some of the identified patterns were incorrectly identified by the approach. The approach would identify a pattern being present because its structure in the system matched that of the pattern. However, when Elaasar manually went to verify the existence of the patterns it was found that they were incorrectly labeled because they lacked both the behaviour of the pattern and intention to use the design pattern.

2.1.3 Jézéquel's Approach

Jézéquel uses a modeling approach for the specification and verification of the Go4 patterns. His approach is included because it allows for some form of correspondence check between the specification and the pattern code.

Jézéquel uses the Eiffel programming language to capture the patterns. The next subsection (2.1.3.1) will present an example of his approach. The example being used is

his representation of a system using the Strategy pattern. Eiffel is a programming language that allows the user to specify conditions for the code that are checked to see if the method or class execution matches the requirements. The conditions put on the methods can range from pre and post conditions to enforcing variable types. The contracts for a class can contain information for invariants, methods, variables, and constructors. The standard documentation for Eiffel can be found on ECMA's website (ECMA International, 2006). In the next subsection, I will provide an overview of a couple of classes written in Eiffel and what their contents mean.

The example that Jézéquel uses is one in which he prints figures on a variety of devices. In the example, the FIGURE class takes the place of the Strategy pattern's Context class while the DEVICE class is the abstract Strategy class and the specific devices the Concrete Strategy classes.

The initial parts of the FIGURE class can be seen in Figure 15. First Jézéquel states that the class name is FIGURE and on the second line he declares that there are two different constructors in this class "make_default" and "make." The next part of the class specifies what Eiffel needs to verify for the two constructors. The first constructor dealt with is the "make_default" constructor that does not contain any parameters, but contains the execution code for the constructor. The second constructor "make" does include a parameter. The specification for the second constructor contains a "require" statement, which works as a pre-condition and states that Eiffel must check that the parameter variable is not void before the method is run. The following line in the constructor states that there is a local variable of type integer named "index." The next part of the second

constructor is the execution code which contains a loop. The last couple of lines in Figure 15 deal with a class variable. This variable contains a link to a DEVICE class.

```
class FIGURE
creation make_default, make
feature {ANY} -- Creation
   make_default is
         -- build a default square
      do
         !!points.make
         points.add_last(new_point(1,1))
         points.add_last(new_point(1,5))
         points.add_last(new_point(5,1))
         points.add_last(new_point(5,5))
      end -- make_default
   make (pts : COLLECTION[POINT]) is
         -- make figure from COLLECTION of points
      require pts_not_void: pts /= Void
      local index : INTEGER
      do
         !!points.make
         from index := pts.lower until index > pts.upper
         loop
            points.add_last(pts.item(index))
            index := index + 1
         end -- loop
      end – make
feature {ANY} -- Queries
   device : DEVICE -- Where the figure must be drawn
```

Figure 15: Jézéquel Example - Figure Class Creation and Query (Jézéquel, Train, & Mingins, 1999)

The next part of the FIGURE class deals with the methods of the class, the specification for the "set_device" method can be seen in Figure 16. Sections of Eiffel classes are separated by the statement "feature {ANY}." This statement declares that the content in this block is viewable to everyone. If the statement said "feature {NONE}" then the contents of the block would only be visible inside the class. The method in Figure 16 is "set_device." It contains a parameter and has a "requires" statement making sure that the value being handed into the method via the parameter is not null. However, the specification for this method contains the "ensures" statement. The "ensures" statement works as a post-condition. This means that what is stated there is checked after

the execution has occurred to verify that the method ended with the right result. In this method, the "ensures" statement makes certain that the class variable "device" has been set to the value that was handed in via the parameter.

```
feature {ANY} -- Commands
   set_device (d : DEVICE) is
         -- define current figure device
      require device_not_void : d /= Void
      do
         device := d
      ensure
         device = d
      end -- set_device
```

Figure 16: Jézéquel Example - Figure Class "set_device" Command (Jézéquel, Train, & Mingins, 1999)

```
feature {NONE} -- Private
   new_point (x,y : INTEGER) : POINT is
         -- Factory method to create a new POINT
      do
         !!Result.make(x,y)
      ensure created: Result /= Void
      end -- new_point
   points : LINKED_LIST[POINT]
invariant
   initialized : points /= Void
end – FIGURE
```

Figure 17: Jézéquel Example - Figure Class Private and Invariant (Jézéquel, Train, & Mingins, 1999)

The final section of the FIGURE class contains an invariant and information on a method and variable that are visible only from inside this class. This part can be seen in Figure 17. The method "new_point" returns a value of type POINT and has an "ensure" statement that verifies that the value being returned is not void. The class variable "points" is a linked list of type POINT. The last part of the contract is for an invariant. An invariant is a statement that must be true at all times outside of a method execution. In this case, the invariant states that the variable "points" cannot be void. In other words, the "points" variable must contain a reference to a linked list.

The next class covered in this section is the DEVICE class, which can be seen in Figure 18. It is a deferred class; which means that it is an abstract class whose contents are expanded in other classes that implement or inherit it. A deferred class contains all the shared information for the classes that inherit it. This deferred class contains three methods: “move_to,” “draw_line,” and “draw_point.” The contents of these three methods are the keyword “deferred.” Where the keyword “deferred” occurs in methods, it means that the subclasses must fill in the information. If there were any checks to be done that are included in all subclasses, they could be included in this deferred class.

```
deferred class DEVICE
feature {ANY}
   move_to (x,y : INTEGER) is
         -- set current position
      deferred
      end -- move_to
   draw_line (p1, p2 : POINT) is
         -- emit commands to draw a line from p1 to p2
      deferred
      end -- draw_line
   draw_point (p1 : POINT) is
         -- emit commands to draw a point
      deferred
      end -- draw_point
end -- DEVICE
```

Figure 18: Jézéquel Example - DEVICE Class (Jézéquel, Train, & Mingins, 1999)

There are other classes made by Jézéquel for this example, such as the devices and a POINT class. The classes used for this example including the ones not included in this section can be found in the file “Jézéquel – Strategy” on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/).

In conclusion, the specifications made for the example are very closely linked to the classes being looked at and cannot be easily reused. This approach lacks the ability to

specify a sequence of events and to verify that the sequence occurs, which hinders the approach from showing an interaction between classes.

2.1.4 Generative Programming

The goal behind Generative Programming is the design and implementation of software modules that can be combined in order to generate specialized and highly optimized systems fulfilling specific requirements (Eisenecker, 1997) (Czarnecki & Eisenecker, Generative Programming Methods, Tools, and Applications, 2000). Generative Programming aims to decrease the conceptual gap between program code and domain concepts, achieve high reusability and adaptability, simplify managing many variants of a component, and increase efficiency (Czarnecki, Eisenecker, Gluck, Vandevoorde, & Veldhuizen, 1998) (Czarnecki & Eisenecker, Generative Programming Methods, Tools, and Applications, 2000).

In order to meet these goals, Generative Programming makes use of multiple principles, which are detailed in Czarnecki's 1998 paper (Czarnecki, Eisenecker, Gluck, Vandevoorde, & Veldhuizen, 1998). The first principle is called "Separation of Concerns". This principle means that the written code should only deal with one issue at a time and each issue should be separated into different modules that can be combined in order to create a needed object. The second principle is called "Parameterization of Differences" in which families of components can be compactly represented through the use of parameters. A family is a set of objects or classes that share common content. The third principle deals with how the parameters affect one another. This principle is labeled as "Analysis and Modeling of Dependencies and Interactions". The fourth principle is

"Separating Problem and Solution Spaces" in which the problem space consists of the domain-specific abstractions for application programmers and the solution space contains the implementation components that can be combined to make the software (Czarnecki & Eisenecker, Generative Programming Methods, Tools, and Applications, 2000). The final principle is "Eliminating Overhead and Performing Domain-Specific Optimizations." This principle stems from the fact that in Generative Programming the components are created statically at compile time, during which most of the overhead that can be created by unused code, run time checks, and unnecessary levels of indirection can be eliminated (*Ibid*). These principles are very relevant to my approach, as will be explained later. That is, in this thesis, my interest in Generative Programming lies in what it can do for the specification of design patterns. In particular how it can represent the variability of a design pattern in its specification. One of the approaches I propose in this thesis will use the ACL programming language in a generative way in order to specify design. In the next section (2.2), the different aspects of the ACL programming language will be examined to show how it allows for such a Generative Programming approach.

2.2 Another Contract Language (ACL)

The specification language that will be used for this thesis is called Another Contract Language (ACL). ACL is a high-level contract language that is closely tied to the concept of requirements. The language was developed by Arnold and Corriveau (Arnold, 2009; Arnold, Corriveau, & Shi, 2010a, 2010b, 2010c). It contains constructs for the specification of goals, beliefs, scenarios, as well as several lower-level constructs, such as pre and post-conditions. I will go into greater detail about these constructs in this

section. One reason ACL was chosen for this thesis is because initial experimentation with it suggested that it is indeed well suited to capture both variability and design patterns. A second reason for choosing ACL is the fact that in combination with the Validation Framework it is feasible to verify that the realizations of the design patterns match the ACL specification.

2.2.1 Contracts

ACL makes use of Contracts to represent an existing class in the implementation under test (hereafter IUT). A contract consists of several different parts that will be discussed further in the upcoming subsections. Contracts make use of both dynamic and static checks. Dynamic checks are done during run time and typically address behaviour, while static checks are done before execution of the IUT and verify a system's structure. A contract can be abstract, meaning that it contains information common for the contracts that inherit it and cannot be used as a complete contract by itself. Inside the contract anything can be labeled as abstract by preceding the construct type by the keyword *abstract*, which causes that line of code to work as a placeholder for the construct to be filled in by the subcontracts. A contract is declared through the use of either "Contract" or "MainContract" followed by the name of the contract. For example "Contract Memento" followed by an opening bracket would start the contract for a memento class.

2.2.1.1 Parameters

```
Parameters
{
      Scalar Boolean InstanceBind OriginatorAccessOnly;
}
```

Figure 19: ACL Parameter Section

The first section in a contract typically contains a block of code for parameters. An example of how a parameter variable is specified can be seen in Figure 19. Any variables specified in the "Parameters" block of code work in a similar fashion as a constant variable in Java. The value in which the variable is set to can be done in one of three different ways. The first way is for it to be set directly in the contract. The second way is for it to be set by the binding tool. The final way it can be set is at run time (Arnold, Corriveau, & Shi, Modeling and Validating Requirements using Executable Contracts and Scenarios, 2010b). The binding tool will be discussed in section 2.2.3.1.

The way that the parameter variable is set in the example shown in Figure 19 is at run time. The keyword "InstanceBind" that can be seen preceding the variable name denotes this. The use of parameters enables the use of ACL in a Generative Programming way as will be explained later. The keyword "Scalar" means that the parameter can hold only one value. For a variable that can hold multiple values, the keyword "List" would be used instead.

2.2.1.2 Structure

```
Structure
{
	choice(Parameters.enforcingInterfacePresence) == true
	{
		HasInterface(true);
	}
}
```

Figure 20: ACL Structure Block

The "Structure" block is where all the static checks occur. A static check typically checks the structure of the program in the IUT. This enables the user to verify that the correct structure is in place and that it matches that of a design pattern. In the example of the "Structure" block located in Figure 20, the "choice" statement can be seen. The "choice" statement works similarly to an "if" statement. If there were an "else" statement, it would be marked using the "alternative" keyword. The "choice" statement is essential for using ACL as a Generative Programming approach because it enables the user to set the conditions for when part of the specification is to be included when the contract is being run. This is important because it enables part of a specification to be included or not included when the specification is being executed at the same time as the IUT

2.2.1.3 Invariant

```
Invariant ObserverHasSubject
{
	Check(subject().length() > 0);
}
```

Figure 21: ACL Invariant

One of the constructs available in ACL is the invariant. An invariant is any statement that must be true at all times outside of a responsibility execution. In ACL, an

invariant is contained in a block of code marked by the use of the word “Invariant.” An example of an invariant can be seen in Figure 21. The example shows an invariant for an observer contract that states that an observer must have at least one subject at all times.

2.2.1.4 Observabilities

```
[Parameters.OriginatorAccessOnly == true] Observability tOriginator
originator();
Observability S state();
```

Figure 22: ACL Observability

The use of “Observability” marks a safe (i.e., side-effect free) method. A safe method is one that does not change anything about the IUT. An “Observability” can contain parameters, pre and post-conditions, and beliefs just like responsibilities do.

In the example for observabilities, which is shown in Figure 22, one can see two different observabilities. The first observability named “originator,” which returns an object of type tOriginator, contains a statement preceding the word “Observability.” That statement, which uses a parameter named “OriginatorAccessOnly,” must be true for that observability to be included when both the specification and the IUT are run. This is an addition that is being made for aACL. As discussed in section 2.2.1.1, the value for the parameter variable is set when the specification is run. This enables the contracts to have observabilities that are determined to be included at runtime. This works similarly to the conditionally generated code that is present in a Generative Programming approach. The second observability returns a value of type “S” which is set when the contract is bound to an IUT.

2.2.1.5 Responsibilities

Responsibilities are tied to the methods of a class located in the implementation under test. Unlike observabilities, these can be unsafe methods. Unsafe methods can make changes to aspects of the implementation such as variables.

```
Responsibility setState(S newState)
{
	choice(Parameters.OriginatorAccessOnly) == true
	{
		Pre(originator() not= null);
		Belief("Only Originator who created the memento
can access this");
	}
	Pre(state() not= null);
	Pre(newState not= null);
	Execute();
	Belief MementoStateChanged("Memento state has been
changed to newstate")
	{
		Post(newState == state());
	}
}
```

Figure 23: ACL Responsibility

The example of a responsibility seen in Figure 23 focuses on the “setState” method located in the memento pattern. The first part of this responsibility deals with a "choice” statement. This statement delineates a block of code to be included in the responsibility should the parameter equal a specific value, which aids in the use of Generative Programming techniques in ACL. The “choice” block makes use of a pre-condition and a “Belief” statement. A “Belief” statement details an aspect of the contract that is to be true at that point in time and can contain a block of code to show that it occurs. All responsibilities must contain an “Execute()” statement that shows when the method in the IUT is run. Inside the “Belief” statement, which is located after the

"Execute()" statement, there is a post-condition. This post-condition states that after the method is executed the state in the memento must have been set to the new state.

One aspect of a responsibility not shown in this example is that a responsibility in aACL, just like an observability, can be preceded by a statement that determines if it is to be included in the contract at runtime. In this case, the statement would have looked like this:

"[Parameters.OriginatorAccessOnly == true] Responsibility setState(S newState)"

and it would have meant that this responsibility is to be included only when that parameter is true. This aspect of a responsibility helps in writing contracts in the Generative Programming style. It enables the contracts to be broken up into components that are either included or excluded at runtime based on the value set for the variables in the "Parameters" block.

There are two special responsibilities in ACL; they are "new()" and "finalize()." Both of these methods contain no return type or parameters. The "new()" responsibility deals with what happens when the contract is first created while the "finalize()" responsibility deals with what happens when the contract is destroyed.

2.2.1.6 Scenarios

The use of scenarios allows for the modelling in a contract of an expected series of events that can occur during the execution of the IUT, allowing a contract to represent the class's behaviour. A scenario is started by a triggering event, which is followed by an optional set of execution grammar, and by the termination statement. The scenario shown in Figure 24 is simple as it is triggered by the "new()" responsibility. The execution grammar is that the memento's responsibilities "setState()" and "getState()" are executed

an unknown number of times and then finally the “finalize()” responsibility is executed by the IUT, which causes the scenario to terminate.

The “choice” statement can be used in a scenario, which allows for variability to be modelled as will be illustrated later. The scenario’s execution grammar also allows for blocks denoted by the keywords “atomic” and “parallel”. A block of code preceded by the keyword “atomic” means that only the responsibilities located in the block can be executed. If any other responsibility occurs during that block in the scenario, the latter will fail. “Parallel” allows for a block of code to be executed at the same time as other responsibilities are occurring. In Figure 24, the scenario declaration contains an empty set of parameters. However, a scenario can contain parameters that are to be handed into and be used by the scenario. An example of this will be seen when the Observer pattern is being discussed.

```
Scenario MementoLifetime()
{
      Trigger(new()),
      (setState()*, getState()*)*,
      Terminate(finalize());
}
```

Figure 24: ACL Scenario

2.2.2 Interaction Classes

An interaction class in ACL allows for the specification of how the different contracts relate with one another. In aACL as in contracts, an interaction class can include a parameter section to declare constant variables that are to be used. An interaction class is started similarly to a contract but uses the word “Interaction” as opposed to “Contract.” An example of this can be seen In Figure 25. Figure 25 also shows the capability for an interaction class to contain instance variables. These variables

are usable by all the relations and are used to show that the same instances of contracts are used between relations.

```
Interaction MediatorInteraction
{
	Parameters
	{
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind IsDynamic;
	}

	Instance Mediator med;

	Instance List Colleague col;
```

Figure 25: ACL - Interaction Class

2.2.2.1 Relations

```
Relation ColleagueCreation
{
	Contract Colleague newColleague;

	Instance c;

	c = newColleague.new(),

	col.Add(c)+;

	choice(Parameters.IsDynamic) == true
	{
		med.Register(c)+;
	}
}
```

Figure 26: ACL Relation

Relations constitute the technical content of the interaction class; they capture the details of the contract relationships. They can contain local variables as seen in Figure 26. The variables in an interaction class or in a relation refer to contracts located in the ACL project. In Figure 26, the local variable refers to the Colleague contract used for the Mediator pattern. The keyword “Contract” means that the variable is tied to a type of contract. The relation has a variable that uses the keyword “Instance” to get at a specific contract instance. After the variables are declared, whether they are for the whole

interaction class or just a specific relation, the relation includes a series of responsibilities or scenario calls that are to be included to show the interactions between the classes. As is shown in Figure 26, it is possible to include a "choice" block here to allow for variability. The relation seen in Figure 16 shows what happens when a Colleague is created. After the Colleague is created, the Colleague is added to the instance variable seen in Figure 25. Finally if the "choice" statement is true the relation shows that the Mediator registers the Colleague.

2.2.3 Validation Framework

The Validation Framework was developed in combination with ACL by Arnold and Corriveau (Arnold, 2009). This tool is used for the running of ACL contracts and the binding of the contracts to an IUT. After the binding process has occurred the Validation Framework produces a report of the results. These reports will be further discussed in section 2.3.

2.2.3.1 Binding Tool

Once the contracts are written for an implementation that is being tested, the contracts need to be bound to this IUT. This allows for ACL to be IUT independent. That is, the binding process also allows a) for responsibilities to be mapped to (possibly sequences of) procedures of the IUT and b) contracts to set variable types and return types at the time of binding.

The Validation Framework tool provides a method to bind the ACL contracts to an implementation. The tool does this through the use of a graphical tool. The screenshot seen in Figure 27 shows the binding window from the Validation Framework. In it the users have the option (either automatic or manual) of how they want to bind the contracts

to the IUT. There are two types of automatic binding currently done by the Validation Framework. The first type of auto-binding is done by matching the responsibility name to the IUT method or class name and then by matching the types of parameters. The second type of auto-binding does what the first auto-bind does but also tries to match the parameter values. If an item has been successfully bound it will show up as green, and if it was unsuccessful it will show up as red.

One of the options for setting the values of parameters is for it to be done at the time of binding. Figure 28 shows a screenshot where one can set a parameter value during the binding process. This can be achieved by selecting the parameter from the list on the left. This would set the value of the parameter variable for all instances.

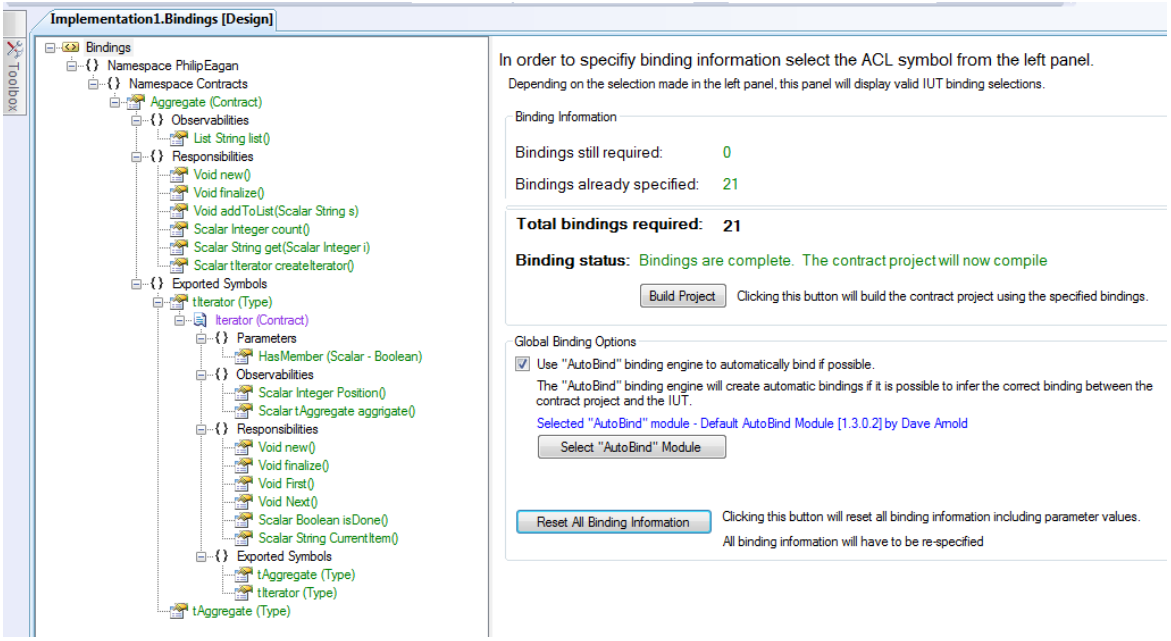

Figure 27: VF Binding

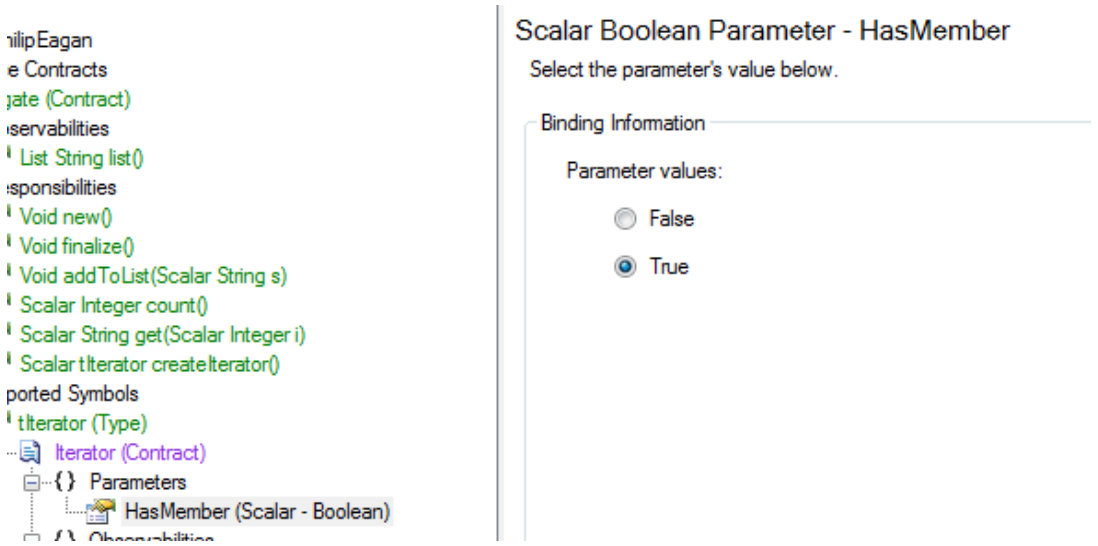

Figure 28 VF Binding - Parameter Selection

2.2.4 ACL Overview

It is my intent to demonstrate later that ACL allows for the specification of the Go4 design patterns in a Generative Programming way that represents not only the pattern but variations of it as well. Furthermore, through the use of the Validation Framework, it will be feasible to verify that the realization of the design pattern matches the specification.

2.3 Running an ACL Contract in the Validation Framework

In this section, an ACL contract running in the Validation Framework will be presented. For the contract being run inside the Validation Framework please see "ACL – VF Example" on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/).

After a contract has been successfully bound to an IUT a report can be generated. The binding of this example can be seen in section 2.2.3.1. This report shows how well the IUT conforms to the contract. For an example of the layout of the report, please refer to Figures 29 and 31. Figure 29 shows an example in which one of the scenarios failed.

The part that failed is shown in red on the left. In the overall report, the result is listed as "Fail", and it is shown that one scenario and one contract failed. This contract failed because the scenario inside it failed. In Figure 30, the reason the scenario failed is shown. In this case, it is because the scenario failed to run correctly. The report seen in Figure 31 succeeded and, therefore, in the overall result it is declared as having passed, and everything on the left is green.

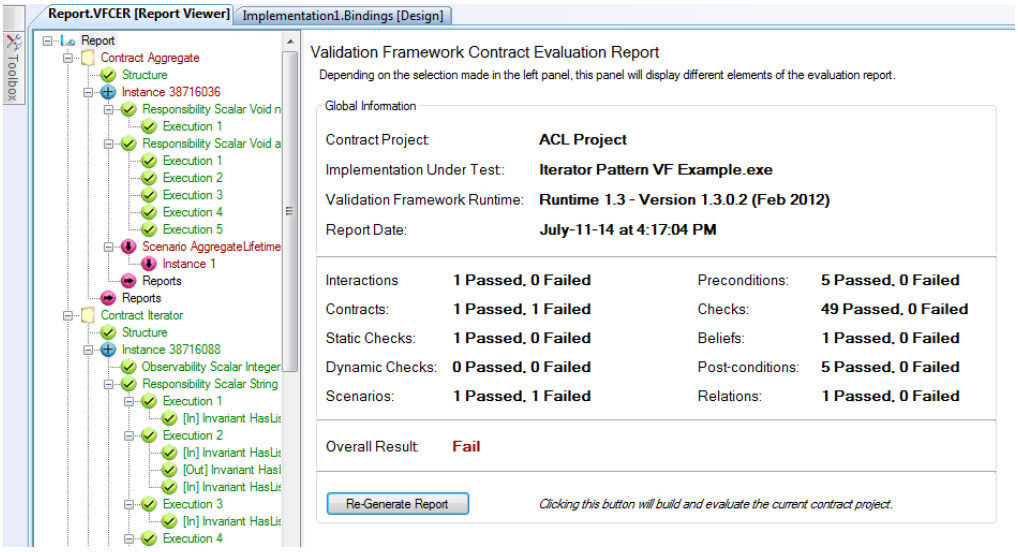

Figure 29: VF Report – Failure

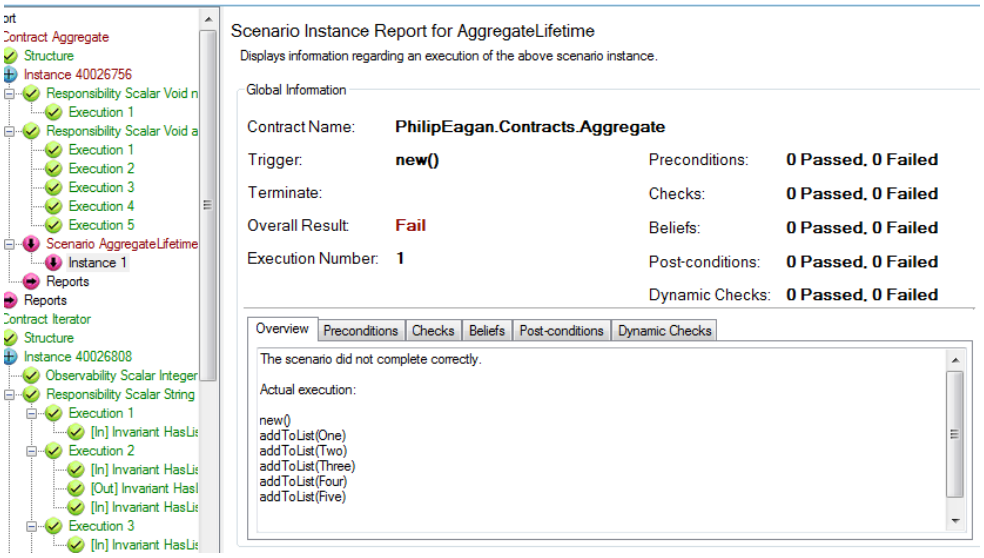

Figure 30: VF Report - Failure Drill Down

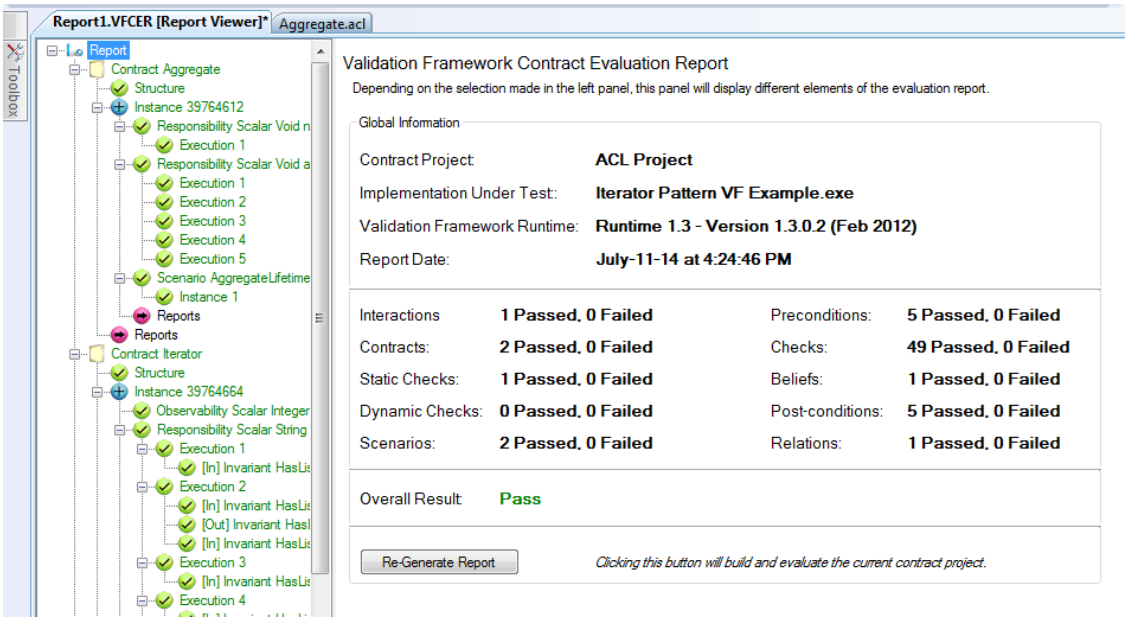

Figure 31: VF Report – Pass

My claim is that ACL allows a user to write the specifications of Go4 patterns that deals with variability. Then the use of the Validation Framework will allow its user to verify that a specification of the pattern matches its realization. We explore the specification of a design pattern using ACL in the next chapter.

2.4 Summary

In this chapter I discussed a variety of techniques that have been proposed in the literature for the detection and modelling of patterns. These approaches range from the use of diagrams to graphical and code-based representations. Some of the discussed approaches provide a tool that is capable of detecting design patterns. However the ones that do often lack the ability to either handle variations or the ability to show the behaviour of the pattern, which decreases their comparability with my proposed approach.

This chapter also offers an overview of Generative Programming as well as of the two existing approaches that I use to compare to my approach in order to best show the benefits of my proposal. Finally in this chapter I provided an overview of the language used in my two approaches, as well as of the tool used to bind the proposed contracts to an implementation. Because of the nature of the Validation Framework my proposed approaches are for pattern verification and not pattern detection. This is due to the fact that the Validation Framework binds ACL contracts to an IUT and therefore the part of the IUT that a contract is being bound to must already have been identified to be a design pattern candidate by the user.

3 Chapter: Comparison of Approaches

In this chapter, my proposed approach to modeling design patterns will be introduced and compared to the approaches proposed by Elaasar and Jézéquel. I will focus on the Observer pattern, for which well-known variants exist, and specifically on how to capture variability. The modeling of relevant scenarios will be discussed. I will consider two different 'styles' of ACL for my modeling purposes in order to illustrate what a generative style achieves.

Two criteria will be used in comparing my approaches to the abovementioned ones. The first of the criteria that I will use for comparing the approaches is the ability to capture variants, in particular variability in the core (or 'gist') of the pattern. The second of the comparison criteria I will use is the ability to model scenarios relevant to the pattern at hand. The ability to model scenarios is important because it enables the patterns to be identified with more confidence. Scenarios show that the IUT behaves in a manner congruent with the design pattern.

3.1 Observer Design Pattern

3.1.1 Pattern Overview

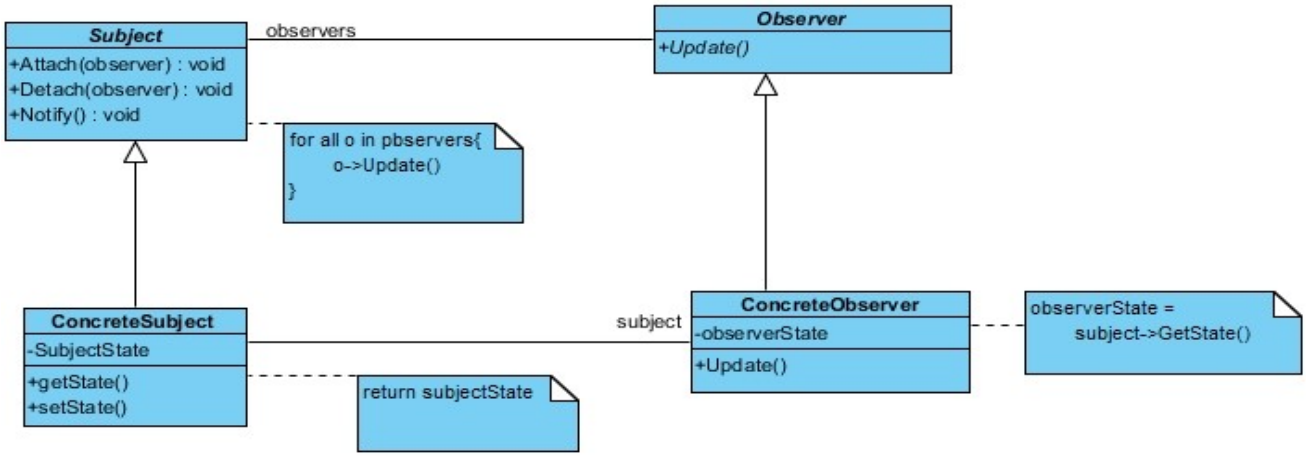

Figure 32: Observer Pattern – Structure

The Observer pattern is one of the more well-known design patterns discussed by the Gang of Four. The structure of the Observer pattern can be seen in Figure 32. An Observer pattern is used when some objects (instances of ConcreteObservers) require getting updated information from a specific object (an instance of a ConcreteSubject). The former are referred to as the Observers, and the latter as the Subject. In the Observer pattern, when a Subject changes, it notifies all of its Observers of the change. This sequence of actions can be seen in Figure 33.

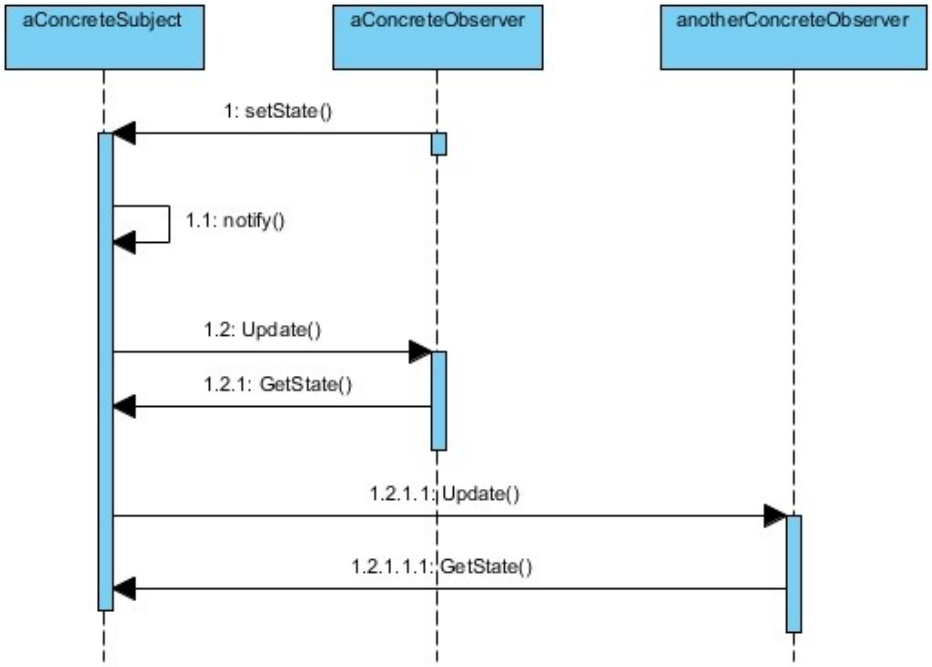

Figure 33: Observer Pattern - Sequence Diagram

3.1.2 Elaasar's Observer Pattern

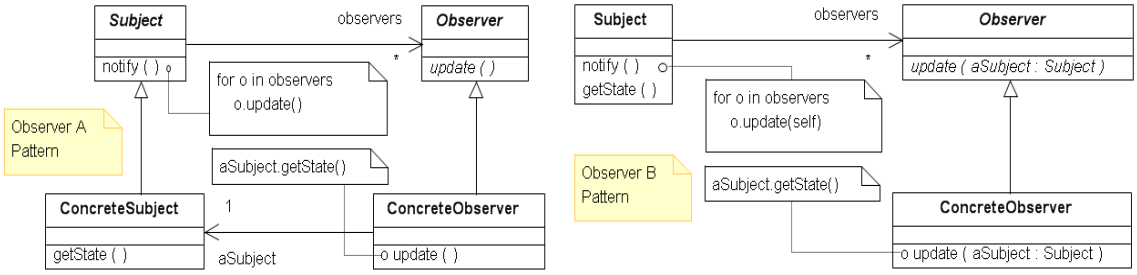

Figure 34: Elaasar's UML Description of the Observer Pattern (Elaasar, 2012)

Elaasar's VPML diagram of the Observer design pattern will be discussed in this subsection. The UML diagram for the two variations of the Observer pattern that Elaasar deals with can be found in Figure 34. The first variation contains a Concrete Subject class that has an abstract parent, and the Observers contain only one Subject. The second

variation of the Observer pattern that Elaasar deals with does not use an abstract Subject class. Instead, the "update" method must be handed in a Subject that has been updated through a parameter.

3.1.2.1 Overview of VPML Representation

The VPML diagram for the Observer pattern can be seen in Figure 35. Elaasar uses three diagrams to show the pattern with the two variations. There is one diagram that contains the information shared between the two variations and a diagram for each variant showing what is unique in each one. The VPML diagram for the Observer pattern makes use of the previously defined delegation and redefinition idioms, which can be seen in Figures 13 and 14 in section 2.1.2.1.

The base diagram, named "ObserverBase" contains several roles. It contains a role for the Subject class, which has roles attached to it for a property named "observers," and an operation named "notify." This part of the diagram uses the delegation idiom to show that when the "notify" operation occurs, its sole purpose is to call the "update" operation in the Observer class. The next role is for the Observer class, which contains a role for an operation named "update." Both the class Observer and the operation "update" are to be abstract. The last class shown in the base diagram is ConcreteObserver, which contains a role for an operation named "update2." This part of the diagram contains a redefinition idiom which states that the contents of the "update2" operation replaces the content for the "update" operation in the abstract Observer class when it is called.

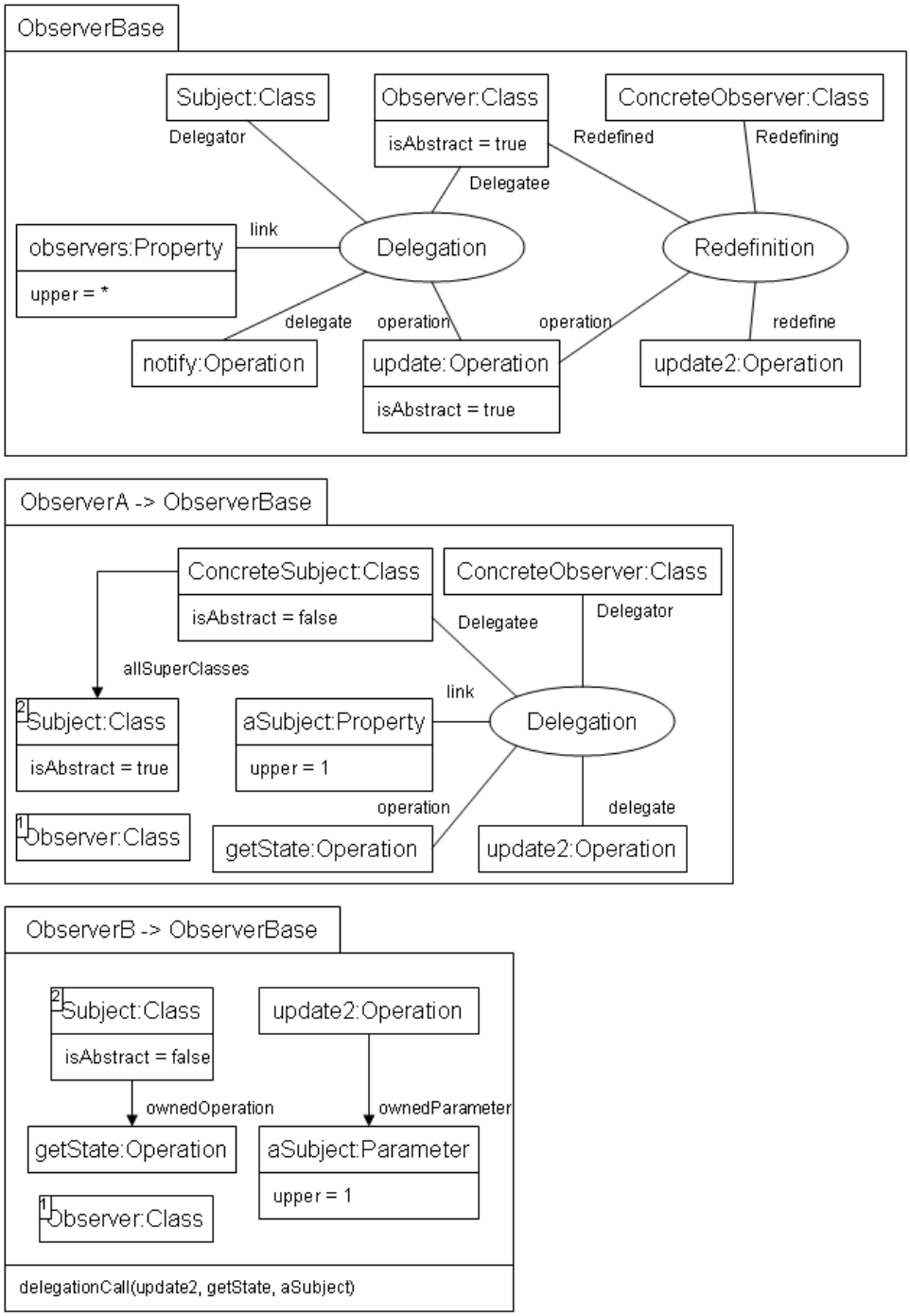

Figure 35: Elaasar's VPML Representation of the Observer Pattern (Elaasar, 2012)

The diagram for the first variation is titled “ObserverA." This variation contains a ConcreteSubject class and makes the Subject class found in the base diagram abstract. The class ConcreteSubject is shown to be a sub-class of the Subject class. The

ConcreteSubject class contains roles for an operation named “getState.” The class ConcreteObserver contains a new role. The added role is a property named “aSubject.” In this part of the diagram, there is a delegation idiom. This idiom shows that when the “update2” method is called, it gets the Subject’s state by calling the ConcreteSubject’s operation “getState” through the property “aSubject.”.

The second variant can be seen in the diagram entitled “ObserverB.” This diagram does not contain a new class, but it does contain a new operation and updates for the existing roles in the base diagram. The new operation is located in the non-abstract Subject class. The name of the operation is “getState.” The operation “update2,” which is located in the ConcreteObserver class in the base diagram, has been given a parameter. This parameter is called “aSubject,” and it uses this parameter to know which Subject has been updated. The bottom of this diagram contains a statement that states that the operation “update2” uses the parameter “aSubject” to call the Subject’s operation “getState.”

3.1.2.2 Handling of Variability

The approach by Elaasar deals with variability through the use of the base and sub-diagrams. While these diagrams let Elaasar show differences in structure, they have minimal ability to show differences in interaction between the classes. This approach struggles in capturing actual variants in code that differ in some way from their representation, whether it is in the parameters, naming of roles or specifically in the behaviour. In his thesis, Elaasar addresses how to solve this problem in the future. In summary, he plans to do this through the use of an algorithm that checks for structures

similar to the pattern being looked for and returns the ones it deems to be highly similar to the pattern being searched for in the implementation.

3.1.2.3 Handling of Scenarios

The approach by Elaasar offers a limited representation of scenarios. A scenario is a way of showing the sequence of events that occur between the classes based on a set of conditions. Elaasar's approach for pattern representation, as seen in the VPML diagram for the Observer pattern, contains a way of showing the order in which a series of events occur. At the bottom of the diagram for variant "ObserverB" one can find that when the "update2" operation is called, that the subject's "getState" method is then called through the use of the parameter "aSubject." However, in large part, this approach suffers from the limited ability of showing interactions between classes and the behaviour that occurs in a class.

3.1.3 Jézéquel's Observer Pattern

Jézéquel's approach as discussed in section 2.1.3, uses the Eiffel programming language to write the specifications for his contracts, including the Observer pattern. Over the next few subsections, I will provide an overview of his Observer contracts and discuss how the topics of variability and scenarios are covered.

3.1.3.1 Overview of the Eiffel Specification

3.1.3.1.1 Observer Contract

Jézéquel provides contracts of a few variations for the Observer class of the Observer pattern. One of the given contracts acts as a parent contract for the other two

Observer contracts. This parent contract provides the essential methods, queries, and variables that are used and expanded in the subcontracts. The two variations of the Observer contract deal primarily with having only one Subject per Observer or with a Subject that contains multiple Observers.

The base contract for the OBSERVER class (see Figure 36) is used as a parent contract for any variation of the Observer pattern. The given Eiffel contract ensures that the bare essentials for an Observer class are present in the subcontracts. The given contract contains rules for a method named "update" that is in charge of updating the Observer's view of the Subject; this Eiffel requirement ensures that the "update" method's parameter must contain an existing Subject object.

```
deferred class OBSERVER
feature
   update (s : SUBJECT) is
       -- update the observer's view on 's'
      require not_void: s /= Void
      deferred
      end -- update
end -- OBSERVER
```

Figure 36: Jézéquel - Base Observer Contract

The main variation that is covered in Jézéquel's method for the OBSERVER class is one that deals with an OBSERVER class containing only one SUBJECT. This Eiffel contract can be seen in Figures 37 and 38. The contract builds on the base OBSERVER contract that was covered in the previous paragraph. The contract for the constructor requires that the value given in the parameters is not null. The execution code for the constructor is then included, the actions being completed are: the parameter value is assigned to the "subject" variable, and the Observer is attached to the Subject.

```
deferred class MONO_OBSERVER
inherit
   OBSERVER
      redefine update end;
feature -- Creation
   make (s : like subject) is
       -- Creation and initialization
      require s_not_void: s /= Void
      do
       subject := s
       subject.attach(Current)
      end -- make

feature {ANY} -- Queries
   subject : SUBJECT
```

Figure 37: Jézéquel - Mono Observer Contract – Creation and Public Queries

The specification of the operations and invariants is shown in the next part of the Eiffel contract (see Figure 38). Jézéquel provides details on the various methods that occur in the OBSERVER class including further details about the operation that deals with an updated Subject. Both the description for the "update" method and the "detach" method have added the code being executed for the method and do not add any additional checks to the methods. The contract also contains one invariant. The invariant states that if the "subject" variable is not void then there exists a Subject that the Observer has been attached to.

```
feature {ANY} -- Commands
   update (s : like subject) is
       -- take actions because 's' did change state
      do
       check s_is_my_subject: s = subject end -- check
       update_observer
      end -- update
   update_observer is deferred end
      -- real action, no check needed on the subject identity
   detach is
       -- detach from its subject
      do
       subject.detach(Current)
       subject := Void
      end -- detach
invariant
   subject_observer: subject /= Void implies
subject.is_attached_to(Current)
end -- MONO_OBSERVER
```

Figure 38: Jézéquel - Mono Observer Contract – Commands and Invariants

3.1.3.1.2 Subject Contract

Jézéquel provides contracts for a couple of variations of the Subject class of the Observer pattern. One of the contracts given acts as a super contract that provides the essential methods and variables that are used in the subcontracts. The two variations of the Subject contract deal primarily with whether a) the Subject has a direct reference to an Observer or b) all the Observer related activities are done through a Mediator.

The base contract for the SUBJECT class (see Figure 39) provides a basic overview of what is required in the Subject class of the Observer design pattern. The contract starts off with a query that ensures that the SUBJECT class is attached to an Observer via its execution. After this initial query, the Eiffel contract for the SUBJECT class contains specifications for the following methods: “attach,” “detach,” and “notify.” The “attach” specification checks to make sure that the parameter is not void and that the Subject is not already attached to the Observer. If not attached, then it ensures that the

Subject is attached after the method is executed. The specification for “detach” and “notify” is self-explanatory.

```
deferred class SUBJECT
feature -- Queries
   is_attached_to (o : OBSERVER) : BOOLEAN is
      -- Whether o belongs to the observers
      require o_not_void: o /= Void
      deferred
      end -- is_attached_to
feature -- Commands
   attach (o : OBSERVER) is
       -- add `o' to the observers
      require
       o_not_void: o /= Void
       not_yet_attached: not is_attached_to(o)
      deferred
      ensure attached: is_attached_to(o)
      end -- attach
   detach (o : OBSERVER) is
       -- remove `o' from the observers
      require o_not_void: o /= Void
      deferred
      ensure not_attached: not is_attached_to(o)
      end -- detach
   notify is
       -- Send an update message to each observer
      deferred
      end -- notify
end -- SUBJECT
```

Figure 39: Jézéquel - Base SUBJECT Contract

Jézéquel provides a couple of variations for the SUBJECT class. These variations will be discussed to show how they differ from one another. The first variation deals with a Subject that contains direct references to its Observers. The second variant deals with a Subject that deals with the existence of a class between the Subject and the Observer classes working similarly to a Mediator pattern. For the first variation, in Figure 40, Jézéquel shows that the SUBJECT class contains a safe operation that confirms that it is attached to a specific Observer. In Figure 41, AUTONOMOUS_SUBJECT contains a collection of OBSERVERS, whether it is in an array or a list, and an invariant that states that the collection of OBSERVERS cannot be void.

```
class AUTONOMOUS_SUBJECT
inherit
   SUBJECT
feature -- Queries
   is_attached_to (o : OBSERVER) : BOOLEAN is
      -- Whether o belongs to the observers
      do
       Result := observers.fast_has(o) -- identity comparison
      end -- is_attached_to
```

Figure 40: Jézéquel - Autonomous Subject – Queries

```
feature {NONE} – Private
observers : COLLECTION[OBSERVER]
       -- the collection holding the observers. Can be instantiated
       -- with either make_array_of_observers or make_list_of_observers
make_array_of_observers is
       -- Factory method to instanciate observers as an ARRAY
      do
       !ARRAY[OBSERVER]!observers.make(1,0)
      ensure observers_initialized: observers /= Void
      end -- make_array_of_observers

 make_list_of_observers is
       -- Factory method to instanciate observers as a LINKED_LIST
      do
       !LINKED_LIST[OBSERVER]!observers.make
      ensure observers_initialized: observers /= Void
      end -- make_list_of_observers
invariant
   observers_initialized: observers /= Void
end -- AUTONOMOUS_SUBJECT
```

Figure 41: Jézéquel - Autonomous Subject - Private Collection and Invariant

The way in which the second variation differs from the other variant is shown in Figures 42 and 43. The specification that is presented in Figure 42 shows a safe method that, in its execution, verifies that the Subject is attached to a specific Observer through its Mediator. Figure 43 shows that there is a private variable in this SUBJECT class that refers to a manager, which works as the Mediator between the Subject and the Observers. There is also an invariant that states that the variable “change_manager” cannot be void. This means that the variable should have been instantiated, and there should be a reference from the Subject to the Mediator class.

```
deferred class MANAGED_SUBJECT
inherit
   SUBJECT
   SINGLETON_ACCESSOR
      rename singleton as change_manager redefine change_manager end;
feature -- Queries
   is_attached_to (o : OBSERVER) : BOOLEAN is
      -- Whether o belongs to the observers
      do
         Result := change_manager.is_registered(Current,o)
      end -- is_attached_to
```

Figure 42: Jézéquel - Managed Subject – Queries

```
feature {NONE} -- Private
    change_manager : CHANGE_MANAGER is
         -- Singleton instance of a CHANGE_MANAGER
      once
         !!Result.make
      end -- change_manager
invariant
   change_manager_initialized: change_manager /= Void
end -- MANAGED_SUBJECT
```

Figure 43: Jézéquel - Managed Subject - Private variable and invariant

Only a portion of the contracts for these two variations has been covered in this section. Please refer to the file "Jézéquel – Observer" on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) for the entire Eiffel specification of the two variants.

3.1.3.2 Handling of Variability

Jézéquel's approach deals with variability 'manually.' That is, in Jézéquel's approach, the contract for the design pattern must be rewritten for each variant. The need to rewrite a contract for each variant makes it difficult for the contracts to look for unexpected variations because each contract is closely tied to a specific implementation. One aspect that is available in Eiffel that may help in modeling variability in a contract is

the ‘rename’ functionality, which allows for a feature that is present in one contract to be renamed in a subcontract to a name fitting the variant.

3.1.3.3 Handling of Scenarios

Jézéquel’s approach has minimal ability to handle scenarios in the contracts. The methods provide the actual execution of the operation. For example, the method, named ‘detach,’ first states that it detaches the Observer from the Subject, then assigns the value void to the “subject” variable. However, while this shows the sequence of event being executed by the actual system, Jézéquel’s approach lacks a method of verifying an expected sequence of events. Jézéquel’s approach does not have the ability to show scenarios inside the contracts nor does it have the ability to show any interaction between the classes.

3.1.4 ACL - Observer Pattern

This section will be looking at the specification of the Observer pattern using an approach in ACL. Two different techniques in ACL that can be used to model the Observer pattern will be covered.

The two techniques covered will be referred to as the Refine approach and the Generative approach. The Refine approach, which is presented in section 3.1.4.1, makes use of the *current* ACL to show both scenarios and the interactions between classes. The Refine approach makes use of sub-contracts to show the differences in each of the pattern’s variations. One disadvantage: the Refine approach can contain redundant information between variants. The Generative approach, which is presented in section 3.1.5.1, allows for the ACL contracts of variants to be written in a concise manner. This conciseness eliminates redundancies in the contracts present in the Refine approach and

helps promote reusability. The Generative approach accomplishes this by having a single contract that can contain components that are either included or excluded at runtime. Inclusion or exclusion of a component depends on which variant is being searched for in the contract. The Generative approach cannot be run in ACL at this time while the Refine approach can. Students in Corriveau's research group are currently working to add the Generative syntax to ACL/VF.

For this section, I will be looking at six variations of the Observer pattern and will model them in ACL using the Refine and Generative approaches. The *push* and the *pull* variations were introduced in the first chapter. A third variant that will be examined is one called the *safe* variant. In this variation, when the Subject notifies an Observer it checks that the Observer is not null, and when an Observer deregisters it checks that the Subject is not null. The fourth variation covered is referred to as the *unsafe* variant, in which the checks of the safe variant are not performed. The fifth and sixth variants deal with whether the Observer or Subject classes contain an interface.

3.1.4.1 Refine Approach

In this approach to writing a pattern contract in ACL, all the shared content for each of the variations is placed in abstract contracts. The abstract class is then refined for each variation containing details unique to the variant. The variants that do not conflict with each other have been combined. As a result, there is a contract for a safe Observer with the pull variation and a contract for a safe Observer with the push variant. A similar combination for the unsafe variant, for the Subject class and its variants, was also done.

3.1.4.1.1 Overview of the ACL Contracts

```
abstract Contract Observer<Type T>
{
	Parameters
	{
		Scalar Boolean InstanceBind CheckMembers;
		Scalar Boolean InstanceBind enforcingInterfacePresence;
	}
	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}

		Belief SubjectType("Subject being passed in method Update
is of type tSubject")
		{
			HasParameterOfType(Observer().Update(),1,T);
		}
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Subject being
Observed")
			{
				HasMemberOfType(T);
			}
		}

	}

	Observability List<T,S> S SubjectState();

	Observability List T subject();
```

Figure 44: Base Observer Contract - Parameters, Structure & Observabilities

The Observer pattern contains classes for the Observer and the Subject. The first that will be discussed is the Observer class. I make use of multiple contracts in order to represent the Observer class and its possible variations. The ACL contract that contains all the common information for the Observer class can be seen in Figures 44, 45, and 46. Figure 44 contains the common parameters, "Structure" block and observabilities.

There are two parameter variables in this contract. The parameter variable "CheckMembers" states that the user is looking to verify that the Observer contains a reference to a Subject. This check is done in the "Structure" block and is important to

check because an Observer is not part of the pattern unless it has a Subject. The second parameter variable is “enforcingInterfacePresence” and states whether the contract is to look for an interface attached to the Observer class in the IUT. The static check is located in the “Structure” block. This check, if run, will verify that the class in the IUT contains an interface. The “Structure” block also contains a static check that verifies that the method named “update” contains a parameter that is receiving a Subject object. This static check is included in all the variations because the Observer should be able to tell which Subject was updated. Static checks like the ones seen here help differentiate the ACL techniques from the approach made by Jézéquel because these checks on the structure are not done in the latter approach. These checks help to verify the structure of a design pattern.

The two observabilities present in this contract show that there needs to be a method for accessing the Subjects being observed and the state of each Subject. It is important that the observabilities be included both for their presence in the design pattern and for their use in the invariant and the pre and post conditions that are done to prove the existence of the pattern through its behaviour.

```
	Invariant ObserverHasSubject
	{
		Check(subject().length() > 0);
	}
	Responsibility new()
	{
		Belief ObserverCreated("Must be created with at least one
subject")
		{
			Post(Subject().length() > 0);
		}
	}
	Responsibility finalize()
	{
		Pre(subject() not= null);
		Belief SubjectRemoved("When Observer destroyed Subjects
have been removed")
		{
			Post(Subject() == null);
		}
	}
	Responsibility Register(T sub)
	{
		Pre(subject().contains(sub) == false);
		Execute();
		Post(subject().contains(sub) == true);
	}
	Responsibility Deregister(T sub)
	{
		Pre(subject().contains(sub) == true);
		Execute();
		Post(subject().contains(sub) == false);
	}
```

Figure 45: Base Observer Contract - Invariant and Responsibilities

The next part of the base contract, which is seen in Figure 45, contains an invariant and some responsibilities. The invariant found in this contract states that the Observer contains a Subject. This is included because when the Observer pattern occurs there is a relationship between the two classes and the invariant ensures that an Observer has a Subject that will notify it of any changes. The “new” responsibility ensures that after a class is created it contains a Subject, while the “finalize” responsibility ensures that when an instance of the Observer is destroyed no Subject is attached to it. The two other responsibilities included in Figure 45, “register” and “deregister”, deal with the attaching and detaching of Subjects to Observers. The pre and post-conditions on the

"register" responsibility show that the Subject being registered was not attached before execution but is attached afterwards. For "deregister," the opposite is true; the Subject is checked to see if it is attached to the Observer before execution and that it is no longer attached afterwards. The invariant and the three responsibilities are included here because they are essential for the Observer class. The invariant shows that the Observer needs a Subject while the two responsibilities show the ways in which a Subject is attached or detached from an Observer.

The "new," "finalize," invariant," and pre and post conditions are all aspects of these ACL contracts that are impossible to model using Elaasar's approach. His approach does not provide any way of representing statements that need to be true at a system's creation and destruction. Elaasar's approach also does not provide a way of representing an invariant or a method of representing statements that need to be true at the beginning and end of a method's execution. These are aspects of the specification that enable ACL to better represent the behaviour of the classes so that it can better determine the presence of a design pattern.

The final part of the base contract for the Observer class can be seen in Figure 46. This figure contains the information that is shared by all the variations for the "update" responsibility. This responsibility contains a pre-condition that verifies that the updated Subject is one being observed by this Observer. Then, before execution, it contains the belief that the Observer can be notified by all Subjects. This belief is important because the action of a Subject notifying an Observer is a significant component of the design pattern, and if the Observer cannot be notified by a class then those two classes do not implement the pattern. Then after execution, it checks that the state of the Subject has

been changed. This class also contains a placeholder for a scenario (to be in all subcontracts), named “updateObserver." This scenario takes in as a parameter a Subject object to be used in the scenario. The inclusion of a parameter in the scenario enables the ACL contract to show that a specific series of events takes place based on the parameter value.

```
	Responsibility Update(T updatedSubject)
	{
		once Scalar S oldState;
		oldState = SubjectState[updatedSubject]();

		Pre(subject().contains(updatedSubject));

		Belief Notifiable("Observer can be notified of a change
by all subjects");

		Execute();

		Belief ObserverUpdated("Observer has been Updated from
oldState")
		{
			Post(oldState not= SubjectState[updatedSubject]());
		}
	}

	abstract Scenario updateObserver(T updatedSubject);
```

Figure 46: Base Observer Contract - Update Responsibility, Scenario

The belief statements that are found in the ACL contracts allow for the capturing of the intent of the system at that point in time. The inclusion of beliefs in contracts enables the responsibilities and contracts to show the objective of the pattern at that time. This is an aspect of the design patterns that is not captured in Elaasar’s and Jézéquel’s approaches, as they do not have a method for representing the intent.

```
Contract SafeObserverPullVariant extends Observer<tSubject>
{

	Responsibility Deregister(tSubject sub) extends Deregister(sub)
	{
		Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
		{
			Pre(sub not= null);
		}
		Execute();
	}

	Responsibility Update(T updatedSubject) extends
Update(updatedSubject)
	{
		Pre(updatedSubject not= null);

		loop(0 to subject().length())
		{
			Pre(subject().At(counter) not= null);
		}

		Execute();
	}

	refine Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject)),
		Terminate( );
	}
```

Figure 47: Safe Observer Pull Variant – Responsibilities

The first variant of the Observer class shown is the safe Observer using the pull variation. The differences from the other variants can be seen in Figure 47. Its "Deregister" responsibility extends the responsibility in the base contract. The line "extends Deregister(sub)" states that the contents of this responsibility are executed as are the contents of the "Deregister" responsibility that are located in the abstract contract. This responsibility adds the belief and a check to see if the Subject being deregistered is not null. This is important for the safe variant. In the "update" responsibility, it adds a check that ensures that none of the Observers attached to the Subject are null before execution. For the pull variant, the "update" responsibility contains only one parameter, namely the Subject that was changed. This is because the Observer needs to request the

information from the Subject as the information is not pushed to the Observer. The "updateObserver" scenario in this case is a simple one triggered when the Observer's "update" method is executed. It is used primarily in the interaction class that will be discussed later.

The second variant of the Observer class being presented is the safe Observer with the push variation. The differences in this variant can be seen in Figure 48. It contains an addition to the "Structure" block that statically checks to see if the "update" method contains a second parameter for the receiving of the Subject's status. This is included because in the push variant, the Subject needs to pass the updated information to the Observer. This variant gets the same addition to the "Deregister" responsibility as in the safe pull variation.

Another difference between the safe push Observer variant's contract and the pull variant is in the "update" responsibility. The "update" responsibility contains the additions that were made for the safe pull variant, but it also contains additions made for the push variant. The new additions are a second parameter, where it takes in the Subject's new state, and a pre-condition that checks that the new state of the Subject is not the same as the old state. After execution, the "update" responsibility contains the belief that the information has been pushed to the Observer and that the state has been updated. This belief shows the important difference from the pull variant in that the Observer does not need to request any information from the Subject and is instead passed the information initially by the Subject. These belief statements are aspects that cannot be modelled by both Jézéquel and Elaasar while the pre and post conditions cannot be

modelled by Elaasar. Both of those statements enable ACL to show the detailed aspects of a design pattern.

```
	Structure
	{
		Belief 2ndParameterUpdate("Method Update() has a second
parameter")
		{
			HasParameterOfType(Observer().Update(),2,T);
		}
	}

	Responsibility Deregister(tSubject sub) extends Deregister(sub)
	{
		Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
		{
			Pre(sub not= null);
		}
		Execute();
	}

	Responsibility Update(tSubject updatedSubject, S state) extends
Update(updatedSubject)
	{
		Pre(SubjectState[updatedSubject]() not= state);
		Pre(updatedSubject not= null);

		loop(0 to subject().length())
		{
			Pre(subject().At(counter) not= null);
		}

		Execute();

		Belief PushedBySubject("In Push Variation Observer is given
new state by Subject without requesting it")
		{
			Post(SubjectState[updatedSubject]() == state);
		}
	}
```

Figure 48: Safe Observer Push Variant - Structure and Responsibilities

The Observer class also comes in two unsafe variations, one for the pull variant and one for the push variant. The difference between these two contracts and the two contracts for the safe variant is that the unsafe contracts do not check to see if a variable is null before a responsibility is executed. Essentially the unsafe contracts contain the

same content as was seen in the safe versions of the two variants except for any part that checked for null values in parameters or variables.

Similar to the Observer class, the Subject class contains a base Subject contract that contains all of the common information found in each of its variations. The base Subject contract can be seen in Figures 49, 50, and 51. The Subject contains a parameter variable and a static check that is also seen in the base Observer contract. In this contract, the static check verifies that the Subject class in the IUT contains an interface. The beginning of the Subject contract (see Figure 49) contains a couple of observabilities, one for the Subject's state and another for the Observers attached to the Subject. These observabilities are included for their use in pre and post-conditions to better identify the presence of the pattern and to show that the Subject has a state to be observed and an Observer waiting for an update. The 'new' responsibility checks to verify that the Subject has been created with an Observer already attached while the 'finalize' responsibility verifies that when the Subject is destroyed that it removes all the Observers attached to it. Both of these responsibilities are included to show that the Subject has an Observer watching it from the time of its creation and that any Observers that the Subject has when it is destroyed are dealt with appropriately.

```
abstract Contract Subject(Type S)
{
	Parameters
	{
		Scalar Boolean InstanceBind enforcingInterfacePresence;
	}
	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}
	}

	Observability Scalar T SubjectState();

	Observability List S observer();

	Responsibility new()
	{
		Post(observer() not= null);
		Belief SubjectCreated("Must be created with at least one
observer")
		{
			Post(observer().length() > 0);
		}
	}

	Responsibility finalize()
	{
		Pre(observer() not= null);
		Pre(observer().length() > 0);
		Belief ObserverRemoved("When Subject is destroyed the
Observers have been removed")
		{
			Post(observer() == null);
		}
	}
```

Figure 49: Base Subject Contract - Observabilities & new and finalize Responsibilities

The responsibilities common for the Subject class can be seen in Figure 50. The first responsibility is called “attach” and is in charge of adding an Observer to the Subject. This responsibility takes in the Observer being attached to the Subject and checks to see if, before execution, the Observer is not attached and that afterwards it is. The “detach” responsibility is in charge of removing an Observer from the Subject. This responsibility checks that the Observer is attached before execution and detached afterwards. The next responsibility is called the “notify.” It is in charge of letting the Observers know of an update. It checks that the Subject has an Observer to update and has the belief that it can notify all Observers of an update. This belief is important because it verifies that the essential behaviour of this class in the Observer pattern can occur. The essential behaviour of a Subject class in the Observer pattern is that it notifies all of its Observers of a change, and if it cannot do that then it is not part of the Observer pattern. The last responsibility is called “setState” in which the Subject is given a new state. In it there is a check that verifies that the new state is different from the old state before execution and that after execution the state of the Subject has been changed to the new one, with the belief that Observers should be notified. This responsibility is what starts the sequence of events that lead to an Observer being notified.

```
Responsibility attach(S o)
{
	Pre(observer().contains(o) == false);
	Execute();
	Belief ObserverRegistered("Observer has been Registered")
	{
		Post(observer().contains(o) == true);
	}
}

Responsibility detach(S o)
{
	Pre(observer().contains(o) == true);
	Execute();
	Belief ObserverDeRegistered("Observer has been DeRegistered")
	{
		Post(observer().contains(o) == false);
	}
}

Responsibility notify()
{
	Belief ObserversToUpdate("Subject has Observer(s) to Update")
	{
		Pre(observer() not= null);
	}

	Belief AllObservers("Subject must be able to notify all observers
of a change");

	Execute();
}

Responsibility setState(T newState)
{
	Pre(SubjectState() not= newState);
	Execute();
	Belief StateUpdated("Subjects State has been updated, Observers
should be notified")
	{
		Post(SubjectState() == newState);
	}
}
```

Figure 50: Base Subject Contract – Responsibilities

The last part of the base Subject contract can be seen in Figure 51. In it, there are two placeholders for scenarios. These scenarios will be filled out in the subcontracts. The purpose of the 'ChangeOccurs' scenario is to show the differences that occur when a change happens between the variations.

```
abstract Scenario ChangeOccurs();

abstract Scenario SubjectLifetime();

Exports
{
	Type T;
}
```

Figure 51: Base Subject Contract - Scenarios & Export

The first variant of the Subject class that will be discussed is the safe Subject with the pull variation. The most important differences between it and the other variations are shown in Figure 52. For the two safe variants' "notify" responsibility, it contains a check that verifies that all Observers are not null before execution. The pull variant of the Subject contract includes a responsibility that the Observer will use to retrieve the Subject's updated state after being notified of a change. The name of this responsibility is 'getState.' This responsibility is needed because in the pull variant, the Subject does not push the updated information to the Observers when it notifies them of a change. As a result the Observer needs a method to retrieve the updated state. The scenario 'ChangeOccurs' shows that when a Subject's state is updated, it goes through the list of Observers looking for null values, and if the Observer is null it detaches that Observer. This is done for the safe variation. After the Observers have been checked it then notifies all of the existing Observers of the update after which it fields requests from them for the updated state.

```
      Responsibility notify() extends notify()
      {
            Belief observerSafe("If Safe Variant, Checks that no
attached observer is null")
            {
                  loop(0 to observer().length())
                  {
                        Pre(observer().At(counter) not= null);
                  }
            }

            Execute();
      }

      Responsibility T getState()
      {
            Belief PullInformation("In Pull variant subject has a state
to send to the observer when asked")
            {
                  Pre(SubjectState() not= null);
            }
            Execute();
      }

      refine Scenario ChangeOccurs()
      {
            Trigger(setState()),

            loop(0 to observer().length)
            {
                  choice(observer().at(counter)) == null
                  {
                        detach()+;
                  }
            }

            notify()+,
            getState()+;

            Terminate( );
      }
```

Figure 52: Safe Subject Pull Variant - Responsibilities and Scenario

The second variation discussed here is the push variant of the safe Subject class. The important parts of it can be seen in Figure 53. Its “notify” responsibility contains the same content that appeared in the safe pull variant. The main differing factor here from the pull variant happens in the ‘ChangeOccurs’ scenario. This scenario contains everything that is in the safe pull variant version except since this variant does not have a ‘getState’ responsibility that line is missing from this scenario. That responsibility is not

present because this variation pushes the data to the Observers when it notifies them of a change.

```
	Responsibility notify() extends notify()
	{

Belief observerSafe("If Safe Variant, Checks that no attached observer
is null")
		{
			loop(0 to observer().length())
			{
				Pre(observer().At(counter) not= null);
			}
		}

		Execute();
	}

	refine Scenario ChangeOccurs()
	{
		Trigger(setState()),

		loop(0 to observer().length)
		{
			choice(observer().at(counter)) == null
			{
				detach()+;
			}
		}
		notify()+;

		Terminate( );
	}
```

Figure 53: Safe Subject Push Variant - Responsibility and Scenario

The contracts for the two unsafe variants of the Subject class, one for the pull variant and one for the push variant, are largely the same as what was seen in the safe variants. The difference in the two unsafe contracts is the missing statements looking for null values in the values handed in via the responsibility's parameters and observabilities before responsibility executions or in a scenario. The parts that were added specifically for the push and pull variants are included in the unsafe versions of those variants. The Pull variant contains the same "getState" responsibility as was seen in the safe variant

because that responsibility is important for the variant. Both the pull and push variants contain a different "ChangeOccurs" scenario from what was seen in the safe variants, and can be seen in Figures 54 and 55. Both of these scenarios do not check for null values. The pull variant of this scenario includes the "getState" responsibility in its sequence of events because the Observers will be requesting information from the Subject.

```
refine Scenario ChangeOccurs()
{
      Trigger(setState()),
      notify()+,
      getState()+;
      Terminate( );
}
```

Figure 54: Unsafe Subject Pull Variant - Scenario

```
refine Scenario ChangeOccurs()
{
      Trigger(setState()),
      notify()+;
      Terminate( );
}
```

Figure 55: Unsafe Subject Push Variant – Scenario

An interaction class, that shows how the Observer and Subject classes interact with each other, is included for the Observer pattern and can be seen in Figure 56. This class enables ACL to show the behaviour of a system that occurs between classes. This is a feature that was lacking in the approaches by Elaasar and Jézéquel. Elaasar's approach focused on the structure of a pattern while Jézéquel's approach did not have a way of representing the classes interacting with each other. The use of an interaction class enables ACL to show how the classes will behave and interact with each other in a given situation. This enables the ACL specification with the Validation Framework to show that the IUT behaves in a manner expected of the pattern. This helps provide more certainty in determining that the pattern is present.

In the class shown in Figure 56, there are a couple of class variables that are included at the beginning. These variables enable the interaction class to show that the same contract instances are being used in each of the different relations. In this interaction class, there are relations that show what happens when the Subjects and Observers are created, when a change occurs, and when the Observer is deregistered.

```
Interaction ObserverSubjectInteraction
{
	Instance Subject sub;

	Instance List Observer obs;

	Relation ObserverCreationtoRegistration
	{
		Instance s;
		Instance o;

		s = sub.new()+; (o = obs.new(), sub.attach(o))+;

		//Adding instances to class variable
		Sub = s;
		obs.Add(o);
	}

	Relation updateSubjectState
	{
		sub.notify()+, obs.updateObserver(sub)+;
	}


	Relation ObserverDeRegistrationtoDestruction
	{
		Contract Observer obs2;

		Pre(obs.contains(obs2))
		obs2.finalize()+, sub.detach(obs2)+;
	}
}
```

Figure 56: Observer Interaction – Relations

The first relation ‘ObserverCreationtoRegistration’ shows what happens when the classes are created. It also works to add values to the class variables. First the Subject is

created. Then, when the Subject is created, the created contract is stored in a local variable in the relation. After the Subject is created, the system expects an Observer to be made, which is also stored in a local variable. The final action in the relation that is expected of the IUT is that the Observer is attached to the Subject. After the sequence of actions in the relation the created Subject is added to the class variable for the Subject and the Observer is added to the list stored in the instance variable "obs." The second relation shows what happens when a Subject is updated and how it notifies the Observer. Note that the Observer is calling the 'updateObserver' scenario and passing the Subject to the parameter to be used in the scenario. This relation shows the interaction between the two classes at the time in which the two classes are completing the important task of the pattern, namely the Subject notifying its Observers of an update. The final relation shows what happens when an Observer is finished. In the relation, it verifies that the Observer being destroyed is one that is part of the list of Observers that were created in the "ObserverCreationRegistration" relation. These relations show the events between classes that are important to the pattern. An example of this is the action of a Subject notifying an Observer of changes. The inclusion of these relations in the interaction class is important because they enable ACL to verify that the interaction between the classes is occurring in a manner congruent with the Observer pattern. This enables there to be more certainty in determining that an Observer pattern is present in the implementation.

The contracts for all the Observer and Subject contracts can be found in Appendix A.1 and in the file "ACL – Observer – Refine Approach" on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) in their entirety.

3.1.4.1.2 Handling of Variability

This ACL technique shows the variations in the design patterns through the use of subcontracts, while keeping the sections that occur in all of the variations in an abstract base contract. One of the benefits in this approach is that in the specification, it is clear what is included in each variant and where they differ. This ACL technique shares in some of the problems found in Jézéquel's technique in that for each variation the user needs to create a separate contract that shows what is different about that variation. This ACL approach also contains a small amount of repetition in some of the variants. That is, some parts of a variant are repeated in other variations, which is why the Generative ACL approach is considered. One of the benefits of the ACL approach in comparison to Jézéquel, is that since the contracts are bound to an implementation through the use of a tool, where each contract and responsibility is bound to an implementation, the contracts can capture some unexpected variations. This occurs because the specific types for the parameters, return types, or variables can be set at the time of the binding. The binding tool also helps to create a more general specification for a pattern and enables the contracts to be applied to different implementations.

3.1.4.1.3 Handling of Scenarios

The concept of scenarios is important to the ACL language and the Validation Framework. One of the components available to each contract is the scenario block. This block allows for the representation of the ordering of the responsibilities during execution as well as how they interact with each other inside the contract. Not only does ACL contain a scenario block in its contract for specifying its behaviour, ACL also contains the option for interaction classes. Interaction classes show how each of the contracts

relate to each other in a given situation. The Observer pattern contains a scenario that shows what occurs in a Subject when a change occurs and how that differs for each variation. The Observer ACL project also contains an interaction class that shows how the contracts interact with each other at the creation of an Observer and Subject as well as when a change occurs to the Subject. The scenario block in the contracts and the interaction classes enable ACL to show the behaviour of a pattern. In comparison to Elaasar's VPML representation and Jézéquel's Eiffel representation, ACL allows for more flexibility and an easier way to show the interaction between classes and the expected behaviour of the pattern. This is done in ACL through the use of both interaction classes, seen in section 2.2.2, and the Scenario block of code, seen in section 2.2.1.6.

3.1.4.1.4 Checking Compliance of the Refine Approach

For this approach I checked the compliance of an Observer pattern specification written using the Refine approach by utilizing the Validation Framework as well as by hand. When I used the Validation Framework I bound the specification to an implementation of the Observer pattern manually, but this could have been done automatically using the tool. Since the Validation Framework requires binding the specification to an implementation, the user must be aware that the part of the implementation being bound to is a candidate for the Observer pattern. After the specification is bound to the candidate implementation, the Validation Framework will run the specification against the IUT and produce a Contract Evaluation Report, as was seen in section 2.3. For checking compliance by hand I took the IUT and verified that the various dynamic and static checks as well as the scenarios would execute as expected.

3.1.4.2 Generative Approach

The Generative Programming approach in ACL for the Observer pattern is showcased in this subsection. Two major benefits of this approach are how compact it makes the contracts and that it avoids redundancy in the contracts. The variants that were discussed in the Refine approach are now presented using this approach.

3.1.4.2.1 Overview of the ACL Contracts

The differences in this approach from the Refine approach and the approaches by Elaasar and Jézéquel are described in this paragraph. The Refine approach needs the creation of a subcontract for each variant. Jézéquel's technique needs the presence of a contract for each variant, while Elaasar's approach needs a diagram for each variant. However, the Generative approach allows for all the variant's information to be combined into a single contract thus more compactly representing variants and reducing redundancy in the contracts.

The Observer contract for the ACL Generative Programming approach can be seen in Figures 57 through 59. Figure 57 shows the "Parameters" and "Structure" blocks of code in the contract. The "Parameters" block contains variables that are used for the determination of which variants of the pattern will be looked for in the implementation. The values for each of these parameters are set at the time of binding. The variable "PushVariant" determines if the contract is looking for the push or pull variant. If the variable is set to true, it is looking for the push variant and if it is set to false, for the pull variant. The second variable, that is named "enforcingInterfacePresence," determines if the contract is to look for the presence of an interface attached to this class during the static check. The third and final variable, which is named "IsSafe," determines if the

contract is looking for the safe or unsafe variant. True means it is looking for the Safe variant and false for the unsafe variant. These parameter variables are used to determine which sections of the contracts are to be included at runtime. This will be further discussed in the upcoming paragraphs.

```
Contract Observer
{
	Parameters
	{
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind enforcingInterfacePresence;
		Scalar Boolean InstanceBind IsSafe;
	}

	Structure
	{
		choice(Parameters.enforcingInterfacePresence == true)
		{
			HasInterface(true);
		}

		Belief SubjectType("Subject being passed in method Update
is of type tSubject")
		{

			HasParameterOfType(Observer().Update(),1,tSubject);
		}
		choice(Parameters.PushVariant == true)
		{
			Belief 2ndParameterUpdate("Method Update() has a
second parameter")
			{
				HasParameterOfType(Observer().Update(),2,T);
			}
		}
	}
```

Figure 57: Generative Observer - Parameters & Structure

The "Structure" Block contains multiple static checks. The static check that verifies the type of the first parameter of the "update" responsibility occurs in all of the variations and is therefore not located inside a "choice" block. However, the other two static checks are only done in specific variations. The determination of whether to include the check is done through the use of the appropriate parameter variable and a "choice"

block. The static check for whether the Observer class implements an interface is done only when the user states the intention of wanting to look for that variation. The static check for the second parameter of the "update" responsibility is only done when checking for the push variation.

The Observer contract deals with the observabilities, invariant, and the two special responsibilities "new" and "finalize" that were seen in the base Observer contract of the Refine approach. They are discussed in section 3.1.4.1.1 and shown in Figures 44 and 45.

The two responsibilities "Register" and "DeRegister" can be seen in Figure 58. The "Register" responsibility is exactly the same as what was seen in the base contract of the Observer in the Refine ACL approach. The "DeRegister" responsibility has been altered to follow a Generative Programming approach, in that there is a part of the responsibility in which the decision on whether or not it is included is done at runtime. The difference between this responsibility and the one that occurs in the Refine ACL approach is the presence of a "choice" statement that contains code to be included at runtime if it is the safe variant being checked for in the contract. In this case, the content of the "choice" block is a belief and pre-conditions that checks to see if the Subject to be removed is not null and that the Observer contains the Subject being removed. The rest of the responsibility is the same as what was seen in the base contract of the Observer in the ACL Refine approach to the Observer pattern.

```
	Responsibility Register(tSubject s)
	{
		Pre(s not= null);
		Pre(Subject().contains(s) == false);

		Execute();

		Belief SubjectAdded("Subject has been added to the
Observer")
		{
			Post(Subject().contains(s) == true);
		}
	}

	Responsibility DeRegister(tSubject s)
	{
		choice(Parameters.IsSafe) == true
		{
			Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
			{
				Pre(s not= null);
				Pre(Subject().contains(s) == true);
			}
		}

		Execute();

		Belief removed("Subject has been removed from the
observer")
		{
			Post(Subject().contains(s) == false);
		}
	}
```

Figure 58: Generative Observer - Register and DeRegister

The "Update" responsibility of the Observer contract can be seen in Figure 59. This responsibility contains a variety of differences from those that can be seen in the base Observer contract of the Refine ACL approach to the Observer pattern. The parameters and the code contained in "choice" blocks are the parts of the responsibility that differ from the base Observer contract in the Refine approach. This responsibility contains two parameters, "updatedSubject" and "newState." The "newState" parameter is preceded by the "Optional" keyword. This keyword is part of aACL and means that not all occurrences of this responsibility need to have a second parameter. In this case, the

parameter is used for the push variant of the Observer pattern. The first "choice" block that occurs in this responsibility is for a section of code that is to be included in the push variant of the pattern. In this section of code, it verifies that the value handed in via the second parameter is not the same as the Subject's state that is currently known to the Observer. The second "choice" block is only used for the push variant. It contains a belief that, in combination with a post-condition, verifies that the new state has been pushed by the Subject to the Observer.

```
	Responsibility Update(tSubject updatedSubject, Optional S
newState)
	{
		once Scalar S oldState;
		oldState = SubjectState();

		Pre(updatedSubject not= null);
		Pre(Subject().contains(updatedSubject));

		choice(Parameters.PushVariant) == true
		{
			Pre(newState not= oldState);
		}

		Belief Notifiable("Observer can be notified of a change by
all subjects");

		Execute();

		choice(Parameters.PushVariant) == true
		{
			Belief PushedBySubject("In Push Variation Observer is
given new state by Subject without requesting it")
			{
				Post(SubjectState() == newState);
			}
		}
		Belief ObserverUpdated("Observer has been Updated from
oldState")
		{
			Post(oldState not= SubjectState());
		}
	}
```

Figure 59: Generative Observer - Update Responsibility

```
Contract Subject
{
	Parameters
	{
		Scalar Boolean InstanceBind enforcingInterfacePresence;
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind IsSafe;
	}
	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}
	}
```

Figure 60: Generative Subject – Parameters & Structure

The Subject contract contains the same parameter variables that can be found in the Observer contract, as seen in Figure 60. However, its "Structure" block contains only one static check. This check is done when the contract is set to verify the existence of an interface being applied to the Subject class. The observabilities, as well as the "new" and "finalize" responsibilities included in the Subject contract, are the same as the ones that were described in the base Subject contract written for the Refine ACL approach.

```
Responsibility attach(tObserver o)
{
	Pre(o not= null);
	Pre(observer().contains(o) == false);
	Execute();
	Belief ObserverRegistered("Observer has been Registered")
	{
		Post(observer().contains(o) == true);
	}
}
Responsibility detach(tObserver o)
{
	Pre(o not= null);
	Pre(observer().contains(o) == true);
	Execute();
	Belief ObserverDetached("Observer has been Detached")
	{
		Post(observer().contains(o) == false);
	}
}
Responsibility notify()
{
	Belief ObserversToUpdate("Subject has Observer(s) to Update")
	{
		Pre(observer().count() not= 0);
	}
	choice(Parameters.IsSafe) == true
	{
		Belief observerSafe("If Safe Variant, Checks that no
attached observer is null")
		{
			loop(0 to observer().length())
			{
				Pre(observer().At(counter) not= null);
			}
		}
	}
	Belief AllObservers("Subject must be able to notify all observers
of a change even if its through a mediator");
	Execute();
}
```

Figure 61: Generative Subject - attach, detach, and notify responsibilities

The responsibilities “attach,” “detach,” and “notify” can be seen in Figure 61. These responsibilities are similar to the ones found in the base Subject contract of the Refine ACL approach with the only difference occurring in the “notify” responsibility. The difference in the “notify” responsibility is the “choice” block which states that its contents are to be included only in the safe variant. The contents of the “choice” block verify that all the Observers that the Subject has listed as being attached to it are not null.

The “getState” responsibility is shown in Figure 62. The only difference in this responsibility from when it occurred in the Refine ACL approach is that the responsibility is preceded by a statement that checks to see if this responsibility is to be included; this check is done at runtime. The “getState” responsibility is included for the pull variation, which is why the statement is checking to see if the value of the parameter variable “PushVariant” is false. The contract for the Subject also contains a responsibility for “setState” but it is the same as the one seen in the Refine approach.

```
	[Parameters.PushVariant == false] Responsibility T getState()
	{
		Belief PullInformation("In Pull information subject has a
state to send to the observer when asked")
		{
			Pre(SubjectState() not= null);
		}
		Execute();
	}
```

Figure 62: Generative Subject – getState responsibility

The Subject contract contains a scenario that shows the expected order of responsibility calls that will occur during its lifetime, and can be seen in Figure 63. This scenario starts when the Subject is created and then expects a Subject to be created. The scenario also expects that when the “setState” responsibility is executed that it is followed by the “notify” responsibility. After these two responsibilities are executed, if it is looking for the safe variant then it knows it can expect the possibility of a “detach” responsibility being executed. Then, if it is the pull variation being checked for, it knows to expect the “getState” responsibility to be executed. Finally, the “detach” responsibility can occur. These are set to be possibly done several times, and the “attach” or “detach” responsibilities can be done outside of the beginning or the end and can be done at

different times during the execution. This scenario is included to show the behaviour that happens inside of the class in the IUT.

```
Scenario SubjectLifetime()
{
      Trigger(new()),
            (attach()*;
            (setState(), notify()+;)*;

            choice(Parameters.IsSafe) == true
            {
                  detach()*;
            }

            choice(Parameters.PushVariant) == false
            {
                  getState()*;
            }
            detach()*;
            )*;
      Terminate(finalize());
}
```

Figure 63: Generative Subject – Scenario

These scenario blocks in contracts show a more detailed representation of the expected series of events than are possible in Jézéquel's or Elaasar's approach. In Elaasar's approach, it is only possible to show a basic series of events, a method being called through the use of a parameter. Jézéquel's approach does not contain a way of verifying and representing an expected sequence of events. These scenario blocks enable ACL to show the expected behaviour of the system in order to better verify the existence of a pattern in the IUT.

The interaction class in the Generative version of the Observer pattern is identical to the one from the Refine ACL approach shown earlier in this thesis. It is not discussed here but is reviewed in section 3.1.4.1.1 and can be seen in Figure 56. The contracts for this approach can be also seen in Appendix A.2 and in the file "ACL – Observer –

Generative" on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) in their entirety.

3.1.4.2.2 Handling of Variability

The Generative approach for the specification of the multiple variants of a design pattern contains advantages unique to this approach as well as advantages found in the other approaches, including the ACL Refine approach. In this Generative approach, the contracts can be written in a manner that eliminates repetition while also covering multiple variations. One of the benefits of the ACL approaches in comparison to one done in Eiffel (e.g. Jézéquel's) is that since the contracts are bound to an implementation through the use of a tool and are not tied to a specific implementation, this allows for the reuse of contracts on other implementations. The contracts can capture some unexpected variations because with the tool the responsibilities, as well as the variable type and return type for methods, can be set via the binding tool. In doing this, the contracts can be more general and capture unexpected structural variants by identifying the patterns by their behaviour instead of their structure.

3.1.4.2.3 Handling of Scenarios

For the handling of scenarios, the Generative approach does not add anything to what was seen in the Refine approach.

3.1.4.2.4 Checking Compliance of Generative Approach

Checking the compliance of the Generative Approach was done by hand. This is due to the fact that the Generative Approach makes use of syntax that is going to be implemented in the fall of 2014. To verify by hand I made use of the IUT that was used

in the Refine Approach to check compliance and went through the implementation verifying that the dynamic and static checks of the contracts would work correctly and also verifying that the scenarios and interactions would run as expected.

3.2 Mediator, Factory, and Memento ACL Overview

3.2.1 Overview

While working on this thesis I also developed ACL contracts for the Mediator, Factory, and Memento design patterns. In this subsection, an overview of the benefits seen in the contracts for the three patterns will be provided.

3.2.2 Mediator

The Mediator pattern was modelled in ACL using both the Refine approach and the Generative approach. There were several variants of the pattern modelled. The first two variants are similar to ones seen in the Observer, the pull and push variants. The next two variant were distinct to the Mediator pattern; they were called the static and dynamic variants. In the dynamic variant, a Colleague was able to be added to a Mediator at any time during runtime while in the static variant Colleagues could only be added at specific times. There were two other variants that dealt with whether or not the Colleague class had an abstract version of the class, and this variant was checked by a static check in the Colleague contract.

For the application of the Refine approach, there needed to be an abstract contract for the Mediator and Colleague, as well as subcontracts for each of them. There were subcontracts for a dynamic Mediator with the pull variant, dynamic Mediator with the push variant, static Mediator with the pull variant, static Mediator with the push

variant, Colleague with the pull variant, and a Colleague with the push variant. Each of these contracts showed what was special about the variant. However with the pull and push variants of the Mediator there was significant redundancy between the contracts for the dynamic and static Mediator.

The Generative ACL contracts that were created for the Mediator were able to solve the redundancy problem. The Generative approach accomplished this by combining the multiple Mediator contracts into a single contract where the differences between the variations are either hidden behind "choice" statements or behind a statement that determines whether a responsibility or observability is to be included in the contract at runtime. This enabled the variants of a class to be represented in a single contract.

The contracts for the Mediator pattern can be found in Appendix B and on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) in the file "ACL - Mediator."

3.2.3 Factory Pattern

For the Factory design pattern, the contracts were made using the Generative approach. The Factory contract was also applied to an example of a Pizza Factory. The example was based on one from the Head First Design Patterns book (Freeman, Freeman, Bates, & Sierra, 2004). The example can be seen in the file "ACL- Pizza Factory" on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/).

The principle difference in the approach taken for the ACL modelling of the Factory pattern was that, for this pattern, I was not trying to model specific variants but was seeking to model a space of variants. In this instance, the components of the contract are written in a way that they can be combined to look for a variant without being

designed to look specifically for that variant. This is possible through the use of Generative Programming in ACL because the components of the contracts can be combined in any manner at runtime. The ability to combine components at runtime allows for the user to dynamically create contracts to look for a specific variant when running the specification and IUT.

While it is possible in the Refine approach to model the space of variants, it is more feasible in the Generative approach. Whereas in the Refine approach there would be the need to write contracts for each possible variant and combination of variants, the Generative approach allows for the components of a variant to be combined at runtime. This allows for the contracts to be concise in their content and for more flexibility in the declaration of variants to be looked for in the specification.

The contracts for the Factory pattern can be seen in Appendix D and on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) in the file "ACL - Factory."

3.2.4 Memento Pattern

The Memento pattern was modelled using the Generative ACL approach. The contracts for this pattern covered a couple of variants. The first variant is a Caretaker that can hold multiple Mementos. The second variant is a Caretaker that can only hold one Memento at a time. The use of the Generative approach enables the writing of the contracts in a concise manner while avoiding redundancy in the contracts.

The contracts for the Memento pattern can be seen in Appendix C and on the website (http://people.scs.carleton.ca/~jeanpier/PhilipEaganModels/) in the file "ACL – Memento."

4 Chapter: Conclusion

Variability is an inherent feature of the Gang of Four design patterns. While there have been many approaches proposed and implemented that deal with the detection of the design patterns, they have largely ignored the ability of a pattern to have variations. This has caused implemented patterns to be missed by the proposed techniques. The major contributions in this thesis to the research area are two new approaches. Both use ACL for the specification of patterns. The Generative approach makes use of Generative Programming techniques to show the variants in a single contract while the Refine approach uses subcontracts to handle variability. Each uses the Validation Framework to show the feasibility of an approach that captures variability in a design pattern as well as verifying the realization of a pattern using the ACL specification.

In this thesis, the approaches of Elaasar and Jézéquel were examined for the purpose of comparing them to the Refine and Generative ACL approaches. Elaasar's approach was to model the patterns and their variants in VPML diagrams that were then converted to a QVTR representation for the purpose of comparing it with a system. Jézéquel used the Eiffel programming language to provide contracts for implementations of design patterns. The approaches of Elaasar and Jézéquel were found to lack the ability to represent the interaction between classes in any meaningful way. An additional problem with Jézéquel's approach is that to deal with any variation the contracts need to be rewritten to look for that variation. For Elaasar's approach, the focus is on the structure of the system which causes the technique to ignore the behaviour of the system and for the approach to miss unexpected variations in a pattern.

The two approaches in ACL that have been developed in this thesis are the Refine approach and the Generative approach. Both are able to largely overcome the problems faced by Jézéquel's and Elaasar's approaches. They address the scenario handling problem through ACL's ability to show the interaction between classes and the behaviour of a system. This is done through the use of Interaction Classes and scenario blocks in contracts.

The Refine approach can show the variants in a pattern through the use of abstract contracts and subcontracts. The abstract contracts contain all the information that is shared by all the variants while the subcontracts contain the information that shows what is different about each variation. For every new variant in the Refine approach, all that needs to be done is to write a subcontract for the classes that show what is unique about that variant. However, this approach can result in redundancy in some contracts. This is usually caused by information being in some variants but not all.

The Generative approach enables the elimination of redundancy by combining all the subcontracts and abstract contracts. This is done by using a Generative Programming approach to write the contracts in a manner in which the sections dealing with variants are separated into components that are determined to be included at runtime.

The proposed techniques of this thesis make use of the ACL programming language. While the Refine approach can be handled by the original ACL system the Generative approach currently cannot be run in ACL. This is due to the fact that the original ACL system is .NET3.5 specific, which does not support most of the Generative syntax used in this thesis. However, as part of Corriveau's research group, several

students are currently working at supporting a new version of ACL that works on top of AspectJ and will offer in fall 2014 the syntax used in this thesis.

In the future, there is the need to apply these ACL approaches to all of the Go4 patterns. This will help to further show the benefits of these approaches and help promote the reusability of the Go4 patterns. This would be beneficial since reusability is the primary reason that the patterns were created in the first place.

Another potential area of future research is to explore how these approaches could be applied to other types of patterns to promote their reusability. This could include the Gang of Five architectural patterns and/or antipatterns related to a systems structure or behaviour. This would demonstrate the broader applicability of these two approaches beyond the Go4 patterns.

Bibliography

ACM Digital Library. (2014, 07 12). *Design patterns: elements of reusable object-oriented software*. Retrieved from ACM Digital Library: http://dl.acm.org/citation.cfm?id=186897

Alexander, C. (1979). *The Timeless Way of Building.* Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language.* New York: Oxford University Press.

Arnold, D. (2009). *An Open Framework for the Specification and Execution of a Testable Requirements Model.* PhD Thesis, Carleton University, Computer Science, Ottawa, ON.

Arnold, D., Corriveau, J.-P., & Shi, W. (2010a). Scenario Based-Validation: Beyond the User Requirements Notation. *Proceedings of the 21st Australian Software Engineering Conference*, (pp. 75-84). Auckland, New Zealand.

Arnold, D., Corriveau, J.-P., & Shi, W. (2010b). Modeling and Validating Requirements using Executable Contracts and Scenarios. *Proceedings of Software Engineering Research, Management & Applications*, (pp. 311-320). Montreal, Canada.

Arnold, D., Corriveau, J.-P., & Shi, W. (2010c). Reconciling Offshore Outsourcing with Model-Based Testing. *Software Engineering Approaches for Offshore and Outscourced Development*, (pp. 6-22). Saint Petersburg, Russia.

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley Professional.

Birkner, M. (2007). *OBJECT-ORIENTED DESIGN PATTERN DETECTION USING STATIC AND DYNAMIC ANALYSIS IN JAVA SOFTWARE.* Masters Thesis, University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin.

Bishop, J. (2007). *C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems.* O'Reilly Media.

Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture.* Wiley.

Czarnecki, K., & Eisenecker, U. W. (2000). *Generative Programming Methods, Tools, and Applications.* Addison-Wesley.

Czarnecki, K., Eisenecker, U. W., Gluck, R., Vandevoorde, D., & Veldhuizen, T. L. (1998). Generative Programming and Active Libraries. *Selected Papers from the International Seminar on Generic Programming* (pp. 25-39). London, UK: Springer-Verlag.

Czarnecki, K., Helsen, S., & Eisenecker, U. (2004). Staged Configuration Using Feature Models. *Software Product Line Conference.*

ECMA International. (2006, June). *Standard ECMA-367 —Eiffel: Analysis, Design and Programming Language 2nd edition.* Retrieved from ECMA International: www.ecma-international.org/publications/standards/Ecma-367.htm

Eisenecker, U. (1997). Generative Programming (GP) with C++. *Proceedings of Modular Programming Languages* (pp. 351-365). Linz, Austria: Springer-Verlag.

Elaasar, M. (2012). *An Approach to Design Pattern and Anti-Pattern Detection in MOF-Based Modeling Languages.* PhD Thesis, Carleton University, Ottawa.

Elaasar, M., Briand, L. C., & Labiche, Y. (2006). A Metamodeling Approach to Pattern Specification. *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems* (pp. 484-498). Berlin: Springer-Verlag.

France, R. B., Kim, D.-K., Ghosh, S., & Song, E. (2004). A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 196-206.

France, R. B., Kim, D.-K., Ghosh, S., & Song, E. (2004). A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 193-206.

Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns.* O'Reilly.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Gupta, M., Pande, A., & Tripathi, A. K. (2011). Design patterns detection using SOP expressions for graphs. *ACM SIGSOFT Software Engineering Notes*, 1-5.

Heuzeroth, D., Holl, T., Hogstrom, G., & Lowe, W. (2003). Automatic Design Pattern Detection. *Proceedings of the 11th IEEE International Workshop on Program Comprehension* (pp. 94-). Washington, D.C.: IEEE Computer Society.

Huang, H., Zhang, S., Cao, J., & Duan, Y. (2005). A practical pattern recovery approached based on both structural and behavioral analysis. *The Journal of Systems and Software*, 69-87.

Jézéquel, J.-M., Train, M., & Mingins, C. (1999). *Design Patterns and Contracts.* Addison-Wesley.

Kim, D.-K., & Shen, W. (2008). Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies. *Software Quality Journal*, 329-359.

Kramer, C., & Prechelt, L. (1996). Design recovery by automated search for strucural design patterns in object-oriented software. *Working Conference on Reverse Engineering*, (pp. 208-).

Le Guennec, A., Sunyé, G., & Jézéquel, J.-M. (2000). Precise modeling of design patterns. *3rd international conference on the unified modeling language: advancing the standard* (pp. 482-496). Berlin: Springer-Verlag.

Mak, J. K., Choy, C. S., & Lun, D. P. (2004). Precise Modeling of Design Patterns in UML. *26th International Conference on Software Engineering.* IEEE.

Petterson, N., & Löwe, W. (2006). Efficient and Accurate Software Pattern Detection. *XIII Asia Pacific Software Engineering Conference.*

Pettersson, N., & Löwe, W. (2007). A Non-conservative Approach to Software Pattern Detection. *15th IEEE International Conference on Program Comprehension.* IEEE.

Radonjic, V., Bashardoust, S., Corriveau, J.-P., & Arnold, D. (2011). Design Patterns: A Modelling Challenge. *International Conference on Software Engineering Research and Practice*, *2*, pp. 191-197. Las Vegas, USA.

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects.* Wiley.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. (2006). Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 896-909.

Appendix

Appendix A Observer Pattern

A.1 Refine Approach

Abstract Observer

```
abstract Contract Observer<Type T>
{
	Parameters
	{
		Scalar Boolean InstanceBind enforcingInterfacePresence;
	}

	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}

		Belief SubjectType("Subject being passed in method Update
is of type tSubject")
		{
			HasParameterOfType(Observer().Update(),1,tSubject);
		}
	}

	Observability List<T,S> S SubjectState();

	Observability List T subject();

	Invariant ObserverHasSubject
	{
		Check(subject().length() > 0);
	}

	Responsibility new()
	{
		Belief ObserverCreated("Must be created with at least one
subject")
		{
			Post(Subject().length() > 0);
		}
	}
```

```
	Responsibility finalize()
	{
		Pre(subject() not= null);
		Belief SubjectRemoved("When Observer destroyed Subjects
have been removed")
		{
			Post(Subject() == null);
		}
	}

	Responsibility Register(T sub)
	{
		Pre(subject().contains(sub) == false);
		Execute();
		Post(subject().contains(sub) == true);
	}

	Responsibility Deregister(T sub)
	{
		Pre(subject().contains(sub) == true);
		Execute();
		Post(subject().contains(sub) == false);
	}

	Responsibility Update(T updatedSubject)
	{
		once Scalar S oldState;
		oldState = SubjectState[updatedSubject]();

		Pre(subject().contains(updatedSubject));

		Belief Notifiable("Observer can be notified of a change by
all subjects");

		Execute();

		Belief ObserverUpdated("Observer has been Updated from
oldState")
		{
			Post(oldState not= SubjectState[updatedSubject]());
		}
	}

	abstract Scenario updateObserver(T updatedSubject);

	Exports
	{
		Type tObserver conforms Observer
		 {Subject::tObserver;}
		Type S;
	}
}
```

Safe Observer with Pull Variant

```
Contract SafeObserverPullVariant extends Observer<tSubject>
{
	Parameters
	{
		Scalar Boolean CheckMembers = true;
	}

	Structure
	{
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Subject being
Observed")
			{
				HasMemberOfType(tSubject);
			}
		}
	}

	Responsibility Deregister(tSubject sub) extends Deregister(sub)
	{
		Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
		{
			Pre(sub not= null);
		}
		Execute();
	}

	Responsibility Update(T updatedSubject) extends
Update(updatedSubject)
	{
		Pre(updatedSubject not= null);

		loop(0 to subject().length())
		{
			Pre(subject().At(counter) not= null);
		}

		Execute();
	}


	refine Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject)),
		Terminate( );
	}
```

```
	Scenario ObserverLifetime()
	{
		Trigger(new()),
		Update()*,
		Terminate(finalize());
	}

	Exports
	{
		Type tSubject conforms Subject
		 {Subject::tSubject;}
	}
}
```

Safe Observer with Push Variant

```
Contract SafeObserverPushVariant extends Observer<tSubject>
{
	Parameters
	{
		Scalar Boolean CheckMembers = true;
	}

	Structure
	{
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Subject being
Observed")
			{
				HasMemberOfType(tSubject);
			}
		}
		Belief 2ndParameterUpdate("Method Update() has a second
parameter")
		{
			HasParameterOfType(Observer().Update(),2,T);
		}
	}

	Responsibility Deregister(tSubject sub) extends Deregister(sub)
	{
		Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
		{
			Pre(sub not= null);
		}
		Execute();
	}
```

```
	Responsibility Update(tSubject sub, S state) extends Update(sub)
	{
		Pre(SubjectState[sub]() not= state);
		Pre(sub not= null);

		loop(0 to subject().length())
		{
			Pre(subject().At(counter) not= null);
		}

		Execute();

		Belief PushedBySubject("In Push Variation Observer is given
new state by Subject without requesting it")
		{
			Post(SubjectState[sub]() == state);
		}
	}

	refine Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject, dontcare)),
		Terminate( );
	}


	Scenario ObserverLifetime()
	{
		Trigger(new()),
		Register()*;
		Update()*;
		DeRegister()*;
		Terminate(finalize());
	}

	Exports
	{
		Type tSubject conforms Subject
		 {Subject::tSubject;}
		Type S;
	}
}
```

Unsafe Observer With Pull Variant

```
Contract UnsafeObserverPullVariation extends Observer<tSubject>
{
	Parameters
	{
		Scalar Boolean CheckMembers = true;
	}

	Structure
	{
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Subject being
Observed")
			{
				HasMemberOfType(tSubject);
			}
		}
	}

	refine Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject)),
		Terminate( );
	}

	Scenario ObserverLifetime()
	{
		Trigger(new()),
		Update()*,
		Terminate(finalize());
	}

	Exports
	{
		Type tSubject conforms Subject
		 {Subject::tSubject;}
	}
}
```

Unsafe Observer With Push Variant

```
Contract UnsafeObserverPushVariation extends Observer<tSubject>
{
	Parameters
	{
		Scalar Boolean CheckMembers = true;
	}
```

```
	Structure
	{
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Subject being
Observed")
			{
				HasMemberOfType(tSubject);
			}
		}
		Belief 2ndParameterUpdate("Method Update() has a second
parameter")
		{
			HasParameterOfType(Observer().Update(),2,T);
		}
	}

	Responsibility Update(tSubject sub, S state) extends Update(sub)
	{
		Pre(SubjectState[sub]() not= state);

		Execute();

		Belief PushedBySubject("In Push Variation Observer is given
new state by Subject without requesting it")
		{
			Post(SubjectState[sub]() == state);
		}
	}

	refine Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject)),
		Terminate( );
	}


	Scenario ObserverLifetime()
	{
		Trigger(new()),
		Update()*,
		Terminate(finalize());
	}

	Exports
	{
		Type tSubject conforms Subject
		 {Subject::tSubject;}
		Type S;
	}
}
```

Abstract Subject

```
abstract Contract Subject(Type S)
{
	Parameters
	{
		Scalar Boolean InstanceBind enforcingInterfacePresence;
	}


	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}
	}

	Observability Scalar T SubjectState();

	Observability List S observer();

	Responsibility new()
	{
		Post(observer() not= null);
		Belief SubjectCreated("Must be created with at least one
observer")
		{
			Post(observer().length() > 0);
		}
	}

	Responsibility finalize()
	{
		Pre(observer() not= null);
		Pre(observer().length() > 0);
		Belief ObserverRemoved("When Subject destroyed Observers
have been removed")
		{
			Post(observer() == null);
		}
	}

	Responsibility attach(S o)
	{
		Pre(observer().contains(o) == false);
		Execute();
		Belief ObserverRegistered("Observer has been Registered")
		{
			Post(observer().contains(o) == true);
		}
	}
```

```
	Responsibility detach(S o)
	{
		Pre(observer().contains(o) == true);
		Execute();
		Belief ObserverDeRegistered("Observer has been
DeRegistered")
		{
			Post(observer().contains(o) == false);
		}
	}

	Responsibility notify()
	{
		Belief ObserversToUpdate("Subject has Observer(s) to
Update")
		{
			Pre(observer() not= null);
		}

		Belief AllObservers("Subject must be able to notify all
observers of a change even if its through a mediator");

		Execute();
	}

	Responsibility setState(Optional S o, T newState)
	{
		Pre(SubjectState() not= newState);
		Execute();
		Belief StateUpdated("Subjects State has been updated,
Observers should be notified")
		{
			Post(SubjectState() == newState);
		}
	}

	abstract Scenario ChangeOccurs();
	abstract Scenario SubjectLifetime();

	Exports
	{
		Type tSubject conforms Subject
		 {Observer::tSubject;}

		Type T;
	}
}
```

Safe Subject With Pull Variant

```
Contract SafeSubjectPullVariant extends Subject<tObserver>
{
	Responsibility notify() extends notify()
	{
		Belief ObserversToUpdate("Subject has Observer(s) to
Update")
		{
			Pre(observer() not= null);
		}

		Belief observerSafe("If Safe Variant, Checks that no
attached observer is null")
		{
			loop(0 to observer().length())
			{
				Pre(observer().At(counter) not= null);
			}
		}

		Execute();
	}

	Responsibility T getState()
	{
		Belief PullInformation("In Pull variant subject has a state
to send to the observer when asked")
		{
			Pre(SubjectState() not= null);
		}
		Execute();
	}

	refine Scenario ChangeOccurs()
	{
		Trigger(setState()),

		loop(0 to observer().length)
		{
			choice(observer().at(counter)) == null
			{
				detach()+;
			}
		}

		notify()+, getState()+;

		Terminate( );
	}
```

```
	refine Scenario SubjectLifetime()
	{
		Trigger(new()),
			(attach()*;
			(setState(), notify()+;)*;
			getState()*;
			detach()*;
			)*;
		Terminate(finalize());
	}

	Exports
	{
		Type tObserver conforms Observer
		{ Observer::tObserver }
	}
}
```

Safe Subject With Push Variant

```
Contract SafeSubjectPushVariant extends Subject<tObserver>
{
	Responsibility notify() extends notify()
	{
		Belief ObserversToUpdate("Subject has Observer(s) to
Update")
		{
			Pre(observer() not= null);
		}

		Belief observerSafe("If Safe Variant, Checks that no
attached observer is null")
		{
			loop(0 to observer().length())
			{
				Pre(observer().At(counter) not= null);
			}
		}

		Execute();
	}

	refine Scenario ChangeOccurs()
	{
		Trigger(setState()),

		loop(0 to observer().length)
		{
			choice(observer().at(counter)) == null
			{
				detach()+;
			}
		}
		notify()+;
		Terminate( );
	}
```

```
	refine Scenario SubjectLifetime()
	{
		Trigger(new()),
			(attach()*;
			(setState(), notify()+;)*;
			detach()*;
			)*;
		Terminate(finalize());
	}

	Exports
	{
		Type tObserver conforms Observer
		{ Observer::tObserver }
	}
}
```

Unsafe Subject With Pull Variant

```
Contract UnsafeSubjectPullVariant extends Subject<tObserver>
{
	Responsibility notify() extends notify()
	{
		Execute();
	}

	Responsibility T getState()
	{
		Belief PullInformation("In Pull variant subject has a state
to send to the observer when asked")
		{
			Pre(SubjectState() not= null);
		}
		Execute();
	}

	refine Scenario ChangeOccurs()
	{
		Trigger(setState()),
		notify()+,
		getState()+;
		Terminate( );
	}

	refine Scenario SubjectLifetime()
	{
		Trigger(new()),
			(attach()*;
			(setState(), notify()+;)*;
			getState()*;
			detach()*;
			)*;
		Terminate(finalize());
	}
```

```
	Exports
	{
		Type tObserver conforms Observer
		{ Observer::tObserver }
	}
}
```

Unsafe Subject With Push Variant

```
Contract UnsafeSubjectPushVariant extends Subject<tObserver>
{
	Responsibility notify() extends notify()
	{
		Execute();
	}

	refine Scenario ChangeOccurs()
	{
		Trigger(setState()),

		notify()+;

		Terminate( );
	}

	refine Scenario SubjectLifetime()
	{
		Trigger(new()),
			(attach()*;
			(setState(), notify()+;)*;
			detach()*;
			)*;
		Terminate(finalize());
	}

	Exports
	{
		Type tObserver conforms Observer
		{ Observer::tObserver }
	}
}
```

Observer interaction

```
Interaction ObserverSubjectInteraction
{
	Instance Subject sub;

	Instance List Observer obs;

	Instance List Mediator med;
```

```
	Relation updateSubjectState
	{
		sub.notify()+, obs.updateObserver(sub)+;
	}

	Relation ObserverCreationtoRegistration
	{
		Instance s;
		Instance o;

		s = sub.new()+; (o = obs.new(), sub.attach(o))+;

		//Adding instances to class variable
		Sub = s;
		obs.Add(o);
	}

	Relation ObserverDeRegistrationtoDestruction
	{
		Contract Observer obs2;

		Pre(obs.contains(obs2))
		obs2.finalize()+, sub.detach(obs2)+;
	}
}
```

A.2 Generative Approach

Observer

```
Contract Observer
{
	Parameters
	{
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind enforcingInterfacePresence;
		Scalar Boolean InstanceBind IsSafe;
	}

	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}
		Belief SubjectType("Subject being passed in method Update
is of type tSubject")
		{
			HasParameterOfType(Observer().Update(),1,tSubject);
		}
```

```
            choice(Parameters.PushVariant == true)
            {
                  Belief 2ndParameterUpdate("Method Update() has a
second parameter")
                  {
                        HasParameterOfType(Observer().Update(),2,T);
                  }
            }
      }

      Observability S SubjectState();

      Observability List tSubject Subject();

      Invariant ObserverHasSubject
      {
            Check(subject().length() > 0);
      }

      Responsibility new()
      {
            Belief ObserverCreated("Must be created with at least one
subject")
            {
                  Post(Subject().length() > 0);
            }
      }

      Responsibility finalize()
      {
            Belief SubjectRemoved("When Observer destroyed Subjects
have been removed")
            {
                  Post(Subject() == null);
            }
      }

      Responsibility Register(tSubject s)
      {
            Pre(s not= null);
            Pre(Subject().contains(s) == false);

            Execute();

            Belief SubjectAdded("Subject has been added to the
Observer")
            {
                  Post(Subject().contains(s) == true);
            }
      }
```

```
	Responsibility DeRegister(tSubject s)
	{
		choice(Parameters.IsSafe) == true
		{
			Belief ObsDeRegSafe("In Safe Variation it checks that
subject being removed is not null")
			{
				Pre(s not= null);
				Pre(Subject().contains(s) == true);
			}
		}

		Execute();

		Belief removed("Subject has been removed from the
observer")
		{
			Post(Subject().contains(s) == false);
		}
	}

	Responsibility Update(tSubject updatedSubject, Optional S
newState)
	{
		once Scalar S oldState;
		oldState = SubjectState();

		Pre(updatedSubject not= null);
		Pre(Subject().contains(updatedSubject));

		choice(Parameters.PushVariant) == true
		{
			Pre(newState not= oldState);
		}

		Belief Notifiable("Observer can be notified of a change by
all subjects");

		Execute();

		choice(Parameters.PushVariant) == true
		{
			Belief PushedBySubject("In Push Variation Observer is
given new state by Subject without requesting it")
			{
				Post(SubjectState() == newState);
			}
		}

		Belief ObserverUpdated("Observer has been Updated from
oldState")
		{
			Post(oldState not= SubjectState());
		}
	}
```

```
	Scenario updateObserver(tSubject updatedSubject)
	{
		Trigger(Update(updatedSubject)),
		Terminate( );
	}

	Exports
	{
		Type tObserver conforms Observer
		 {Subject::tObserver;}
		Type tSubject conforms Subject
		 {Subject::tSubject;}
		Type S;
	}
}
```

Subject

```
Contract Subject
{
	Parameters
	{
		Scalar Boolean InstanceBind enforcingInterfacePresence;
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind IsSafe;
	}

	Structure
	{
		choice(Parameters.enforcingInterfacePresence) == true
		{
			HasInterface(true);
		}
	}

	Observability T SubjectState();
	Observability List tObserver observer();

	Responsibility new()
	{
		Belief SubjectCreated("Must be created with at least one
observer")
		{
			Post(observer().length() > 0);
		}
	}
	Responsibility finalize()
	{
		Pre(observer().length() > 0);

		Belief ObserverRemoved("When Subject destroyed Observers
have been removed")
		{
			Post(observer() == null);
		}
	}
```

```
	Responsibility attach(tObserver o)
	{
		Pre(o not= null);
		Pre(observer().contains(o) == false);
		Execute();
		Belief ObserverRegistered("Observer has been Registered")
		{
			Post(observer().contains(o) == true);
		}
	}

	Responsibility detach(tObserver o)
	{
		Pre(o not= null);
		Pre(observer().contains(o) == true);
		Execute();
		Belief ObserverDetached("Observer has been Detached")
		{
			Post(observer().contains(o) == false);
		}
	}

	Responsibility notify()
	{
		Belief ObserversToUpdate("Subject has Observer(s) to
Update")
		{
			Pre(observer().count() not= 0);
		}

		choice(Parameters.IsSafe) == true
		{
			Belief observerSafe("If Safe Variant, Checks that no
attached observer is null")
			{
				loop(0 to observer().length())
				{
					Pre(observer().At(counter) not= null);
				}
			}
		}

		Belief AllObservers("Subject must be able to notify all
observers of a change even if its through a mediator");

		Execute();
	}
```

```
	[Parameters.PushVariant == false] Responsibility T getState()
	{
		Belief PullInformation("In Pull information subject has a
state to send to the observer when asked")
		{
			Pre(SubjectState() not= null);
		}
		Execute();
	}

	Responsibility setState(T newState)
	{
		Pre(SubjectState() not= newState);
		Execute();
		Belief StateUpdated("Subjects State has been updated,
Observers should be notified")
		{
			Post(SubjectState() == newState);
		}
	}

	Scenario ChangeOccurs()
	{
		Trigger(setState()),
		choice(Parameters.IsSafe) == true
		{
			loop(0 to observer().length)
			{
				choice(observer().at(counter)) == null
				{
					detach()+;
				}
			}
		}

		choice(Parameters.PushVariant) == false
		{
			notify()+, getState()+;
		}
		alternative
		{
			notify()+;
		}

		Terminate( );
	}

	Scenario SubjectLifetime()
	{
		Trigger(new()),
			(attach()*;
			(setState(), notify()+;)*;

			choice(Parameters.IsSafe) == true
			{
				detach()*;
			}
```

```
                choice(Parameters.PushVariant) == false
                {
                      getState()*;
                }
                detach()*;
                )*;
          Terminate(finalize());
    }

    Exports
    {
          Type tObserver conforms Observer
           {Observer::tObserver;}
          Type tSubject conforms Subject
           {Observer::tSubject;}

          Type T;
    }
}
```

Observer Interaction

```
Interaction ObserverSubjectInteraction
{
    Instance Subject sub;

    Instance List Observer obs;

    Relation updateSubjectState
    {
          sub.notify()+, obs.updatedSubject(sub,dontcare)+;
    }

    Relation ObserverCreationtoRegistration
    {
          Instance s;
          Instance o;

          s = sub.new()+; (o = obs.new(), sub.attach(o))+;

          //Adding instances to class variable
          Sub = s;
          obs.Add(o);
    }
    Relation ObserverDeRegistrationtoDestruction
    {
          Contract Observer obs2;

          Pre(obs.contains(obs2));
          obs2.finalize()+, sub.detach(obs2)+;
    }
}
```

Appendix B Mediator Contracts

B.1 Refine Approach

Abstract Mediator Contract

```
abstract Contract Mediator<Type T>
{
	Parameters
	{
		Scalar Boolean CheckMembers = true;
	}

	Structure
	{
		choice(Parameters.CheckMembers) == true
		{
			Belief CheckMember("There should be a Colleague of
the Mediator")
			{
					HasMemberOfType(T);
			}
		}
		Belief ColleagueType("Colleague being passed in method
ColleagueChanged() is of type tColleague")
		{

	HasParameterOfType(Mediator().ColleagueChanged(),1,T);
		}
	}

	Observability List T Colleagues();

	Invariant NumColleaguesCheck
	{
		Check(Colleagues().count() > 1);
	}

	Responsibility new()
	{
		Post(Colleagues() not= null);
	}

	Responsibility ColleagueChanged(T col)
	{
		Pre(col not= null);
		Pre(Colleagues.count() not= 0);
		Execute();
		Belief notfiedColleagues("Mediator has notified all the
relevant colleagues with the changed information");
	}

	abstract Scenario MediatorLifeTime();
```

```
	Exports
	{
		Type tMediator conforms Mediator
		{Colleague::tMediator;}
	}
}
```

Dynamic Mediator Pull Variant

```
Contract DynamicMediatorPull extends Mediator<tColleague>
{
	Responsibility Register(tColleague newColleague)
	{
		Pre(newColleague not= null);
		Execute();
		Belief ColleagueRegistered("New Colleague has been added to
the mediator")
		{
			Post(Colleagues.contains(newColleague) == true);
		}
	}

	Responsibility DeRegister(tColleague c)
	{
		Pre(c not= null);
		Pre(Colleagues.contains(newColleague) == true);
		Execute();
		Belief ColleagueDeRegistered("Existing Colleague has been
removed from the mediator")
		{
			Post(Colleagues.contains(newColleague) == false);
		}
	}

	Responsibility ColleagueChanged(T from) extends
ColleagueChanged(from)
	{
		Belief AllColleagues("Mediator must Deal with every
Colleague")
		{
			Execute();
		}
	}

	refine Scenario MediatorLifetime()
	{
		Trigger(new()),
		(
			Register();*
			ColleagueChanged();*
			DeRegister()*;
		)*;
		Terminate(finalize());
	}
```

```
	Exports
	{
		Type tColleague conforms Colleague
		{Colleague::tColleague;}
	}
}
```

Dynamic Mediator Push Variant

```
Contract DynamicMediatorPush extends Mediator<tColleague>
{
	Structure
	{
		Belief 2ndParameterColleagueChanged("Method
ColleagueChanged() has a second parameter")
		{
		HasParameterOfType(Mediator().ColleagueChanged(),2,T);
		}
	}

	Responsibility Register(tColleague newColleague)
	{
		Pre(newColleague not= null);
		Execute();
		Belief ColleagueRegistered("New Colleague has been added to
the mediator")
		{
			Post(Colleagues.contains(newColleague) == true);
		}
	}

	Responsibility DeRegister(tColleague c)
	{
		Pre(c not= null);
		Pre(Colleagues.contains(newColleague) == true);
		Execute();
		Belief ColleagueDeRegistered("Existing Colleague has been
removed from the mediator")
		{
			Post(Colleagues.contains(newColleague) == false);
		}
	}

	Responsibility ColleagueChanged(tColleague from, T status)
extends ColleagueChanged(from)
	{
		Belief receivedStatus("Colleague pushed data to the
mediator")
		{
			Pre(status not= null);
		}
		Execute();
	}
```

```
	refine Scenario MediatorLifetime()
	{
		Trigger(new()),
		(
			Register();*
			ColleagueChanged();*
			DeRegister()*;
		)*;
		Terminate(finalize());
	}

	Exports
	{
		Type tColleague conforms Colleague
		{Colleague::tColleague;}
	}
}
```

Static Mediator Pull Variant

```
Contract StaticMediatorPull extends Mediator<tColleague>
{
	Responsibility new() extends new()
	{
		Belief("All Colleagues must be created here")
		{
			Post(Colleagues().length() > 1)
		}
	}

	Responsibility ColleagueChanged(T from) extends
ColleagueChanged(from)
	{
		Belief AllColleagues("Mediator must Deal with every
Colleague")
		{
			Execute();
		}
	}

	refine Scenario MediatorLifetime()
	{
		Trigger(new()));
		atomic
		{
			ColleagueChanged()*;
		}*;
		Terminate(finalize());
	}

	Exports
	{
		Type tColleague conforms Colleague
		{Colleague::tColleague;}
	}
}
```

Static Mediator Push Variant

```
Contract StaticMediatorPush extends Mediator<tColleague>
{
	Structure
	{
		Belief 2ndParameterColleagueChanged("Method
ColleagueChanged() has a second parameter")
		{
		HasParameterOfType(Mediator().ColleagueChanged(),2,T);
		}
	}

	Responsibility new() extends new()
	{
		Belief("All Colleagues must be created here")
		{
			Post(Colleagues().length() > 1)
		}
	}

	Responsibility ColleagueChanged(tColleague from, T status)
extends ColleagueChanged(from)
	{
		Belief receivedStatus("Colleague pushed information to the
mediator")
		{
			Pre(status not= null);
		}
		Execute();
	}

	refine Scenario MediatorLifetime()
	{
		Trigger(new()),
		(
			ColleagueChanged();*
		)*;
		Terminate(finalize());
	}

	Exports
	{
		Type tColleague conforms Colleague
		{Colleague::tColleague;}
	}
}
```

Abstract Colleague Contract

```
abstract Contract Colleague
{
	Parameters
	{
		Scalar Boolean InstanceBind HasAbstractParent;
	}
	Structure
	{
		choice(Parameters.HasAbstractParent) == true
		{
			Belief AbstractParent("Colleague has an abstract
parent class")
			{
				HasAbstractParent(true);
			}
		}
	}

	//The Colleagues Mediator
	Observability Scalar tMediator Mediator();

	Invariant HasMediatiorCheck
	{
		Check(Mediator() not= null);
	}

	Responsibility new()
	{
		Belief MediatorCreated("Mediator must be created at
construction")
		{
			Post(Mediator() not= null);
		}
	}

	Responsibility finalize()
	{
		Post(Mediator() == null);
	}

	Responsibility Changed()
	{
		Execute();
	}

	Responsibility action(T input)
	{
		Belief ColleagueAction("Colleague Responds to data from
Mediator appropriately")
		{
			Pre(input not= null);
			Execute();
		}
	}
```

```
	abstract Scenario ColleagueLifeTime();

	Exports
	{
		Type tMediator conforms Mediator
	 {Mediator::tMediator;}
		Type tColleague conforms Colleague
		{Mediator::tColleague;}
		Type T;
	}
}
```

Colleague Pull Variant

```
Contract ColleaguePullVariant extends Colleague
{
	Responsibility Changed() extends Changed()
	{
		Execute();
	}

	Scenario ColleagueLifeTime()
	{
		Trigger(new()),
		atomic{
			Changed()*,  Belief MediatorQueries("Mediator
requests information from the Changed Colleague")+;
			Action()*;
		}*;
		Terminate(finalize());
	}
}
```

Colleague Push Variant

```
Contract ColleaguePushVariant extends Colleague
{
	Responsibility Changed() extends Changed()
	{
		Belief PushedInformation("Complete state of changed
colleague is sent to the mediator");
		Execute();
	}

	refine Scenario ColleagueLifeTime()
	{
		Trigger(new()),
		atomic{
			Changed()*;
			Action()*;
		}*;
		Terminate(finalize());
	}
}
```

Dynamic Mediator Pull Variant Interaction Class

```
Interaction DynamicMediatorPullVariantInteraction
{
	Instance Mediator med;

	Instance List Colleague col;

	Relation MediatorCreation
	{
		Contract Mediator newMediator;
		//Mediator created is added to class variable
		med = newMediator.new();
	}

	Relation ColleagueCreation
	{
		Contract Colleague newColleague;

		Instance c;

		c = newColleague.new(),
		//Add colleague to instance variable
		col.Add(c)+;
		// mediator registers colleague unless it is added when
mediator is created
		med.Register()*;
	}

	Relation MediatorChangeOccurs
	{
		col.Changed()+, med.ColleagueChanged(col)+,
		Belief collectInformation("Mediator Collects Changed
information from Colleague")+;
		Belief notifiesColleagues("Mediator notifies all colleagues
of the change except for the one that made the change")
		{
			col.action(dontcare)[(col.length() - 2)];
		}+;
	}
}
```

Dynamic Mediator Push Variant Interaction Class

```
Interaction DynamicMediatorPushVariantInteraction
{
	Instance Mediator med;

	Instance List Colleague col;
```

```
	Relation MediatorCreation
	{
		Contract Mediator newMediator;
		//Mediator created is added to class variable
		med = newMediator.new();
	}

	Relation ColleagueCreation
	{
		Contract Colleague newColleague;

		Instance c;

		c = newColleague.new(),

		//Add colleague to instance variable
		col.Add(c)+,
		// mediator registers colleague unless it is added when
mediator is created
		med.Register()*;
	}

	Relation MediatorChangeOccurs
	{
		col.Changed()+, med.ColleagueChanged(col,dontcare)+,
		Belief notifiesColleagues("Mediator notifies all colleagues
of the change except for the one that made the change")
		{
			col.action(dontcare)[(col.length() - 2)];
		}+;
	}
}
```

Static Mediator Pull Variant Interaction Class

```
Interaction StaticMediatorPullVariantInteraction
{
	Instance Mediator med;

	Instance List Colleague col;

	Relation MediatorCreation
	{
		Contract Mediator newMediator;
		//Mediator created is added to class variable
		med = newMediator.new();
	}
```

```
	Relation ColleagueCreation
	{
		Contract Colleague newColleague;

		Instance c;

		c = newColleague.new(),
		//Add colleague to instance variable
		col.Add(c)+;
	}

	Relation MediatorChangeOccurs
	{
		col.Changed()+, med.ColleagueChanged(col)+,
		Belief collectInformation("Mediator Collects Changed
information from Colleague")+;
		Belief notifiesColleagues("Mediator notifies all colleagues
of the change except for the one that made the change")
		{
			col.action(dontcare)[(col.length() - 2)];
		}+;
	}
}
```

Static Mediator Push Variant Interaction Class

```
Interaction StaticMediatorPushVariantInteraction
{
	Instance Mediator med;

	Instance List Colleague col;

	Relation MediatorCreation
	{
		Contract Mediator newMediator;
		//Mediator created is added to class variable
		med = newMediator.new();
	}

	Relation ColleagueCreation
	{
		Contract Colleague newColleague;

		Instance c;

		c = newColleague.new(),
		//Add colleague to instance variable
		col.Add(c)+;
	}
```

```
	Relation MediatorChangeOccurs
	{
		col.Changed()+, med.ColleagueChanged(col,dontcare)+,
		Belief notifiesColleagues("Mediator notifies all colleagues
of the change except for the one that made the change")
		{
			col.action(dontcare)[(col.length() - 2)];
		}+
	}
}
```

B.2 Generative Approach

Mediator Contract

```
Contract Mediator
{
	Parameters
	{
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind IsDynamic;
	}

	Structure
	{
		Belief ColleagueType("Colleague being passed in method
ColleagueChanged() is of type tColleague")
		{

	HasParameterOfType(Mediator().ColleagueChanged(),1,tColleague);
		}

		choice(Parameters.PushVariant == true)
		{
			Belief 2ndParameterColleagueChanged("Method
ColleagueChanged() has a second parameter")
			{
		HasParameterOfType(Mediator().ColleagueChanged(),2,T);
			}
		}
	}

	Observability List tColleague Colleagues();

	Invariant NumColleaguesCheck
	{
		Check(Colleagues().length() > 1);
	}

	Responsibility new()
	{
		Post(Colleagues() not= null);
	}
```

```
	Responsibility finalize()
	{
		Belief AllColleaguesDeRegistered("All Colleagues have been
deallocated from the Mediator")
		{
			Post(Colleagues() == null);
		}
	}

	[Parameters.IsDynamic == true] Responsibility Register(tColleague
newColleague)
	{
		Pre(newColleague not= null);
		Execute();
		Belief ColleagueAdded("Colleague has been added to the
mediator, can be called at any time in Dynamic Mediator")
		{
			Post(Colleagues.contains(newColleague) == true);
		}
	}

	[Parameters.IsDynamic == true] Responsibility
DeRegister(tColleague c)
	{
		Pre(Colleagues.contains(c) == true);
		Execute();
		Belief ColleagueRemoved("Colleague has been removed from
the mediator, can be called at any time in Dynamic Mediator")
		{
			Post(Colleagues.contains(c) == false);
		}
	}

	Responsibility ColleagueChanged(tColleague col, Optional T
change)
	{
		Pre(col not= null);
		Pre(Colleagues().contains(col) == true);

		choice(Parameters.PushVariant) == true
		{
			Belief PushData("In Push Variation Colleague has
given mediator needed data")
			{
				Pre(change not= null);
			}
		}
		alternative
		{
			Belief AllColleagues("Mediator must ask changed
colleague for its state and send relevant information to those
colleagues that require it");
		}

		Execute();
```

```
            Belief notfiedColleagues("Mediator has notified all the
relevant colleagues with the changed information");
      }

      Scenario MediatorLifeTime()
      {
            Trigger(new()),
            (
                  choice(Parameters.IsDynamic) == true;
                  {
                        Register();*
                  }
                  ColleagueChanged();*
                  choice(Parameters.IsDynamic) == true;
                  {
                        DeRegister();*
                  }
            )*;
            Terminate(finalize());
      }

      Exports
      {
            Type tMediator conforms Mediator
            {Colleague::tMediator;}
            Type tColleague conforms Colleague
            {Colleague::tColleague;}
            Type T;
      }
}
```

Colleague Contract

```
Contract Colleague
{
      Parameters
      {
            Scalar Boolean InstanceBind HasAbstractParent;
            Scalar Boolean InstanceBind PushVariant;
      }

      Structure
      {
            choice(Parameters.HasAbstractParent) == true
            {
                  HasAbstractParent(true);
            }
      }

      Observability Scalar tMediator Mediator();

      Invariant HasMediatiorCheck
      {
            Check(Mediator() not= null);
      }
```

```
	Responsibility new()
	{}


	Responsibility finalize()
	{}

	Responsibility Changed()
	{
		Execute();
	}

	Responsibility action(T input)
	{
		Pre(input not= null);
		Execute();
	}

	Scenario ColleagueLifeTime()
	{
		Trigger(new()),
		(
			Changed()*,
			choice(Parameters.PushVariant) == false
			{
				Belief MediatorQueries("Mediator requests
information from Changed Colleague")+;
			}
			action()*;
		)*;
		Terminate(finalize());
	}

	Exports
	{
		 Type tMediator conforms Mediator
		 {Mediator::tMediator;}
		 Type tColleague conforms Colleague
		 {Mediator::tColleague;}
		 Type T;
	}
}
```

Mediator Interaction Class

```
Interaction MediatorInteraction
{
	Parameters
	{
		Scalar Boolean InstanceBind PushVariant;
		Scalar Boolean InstanceBind IsDynamic;
	}

	Instance Mediator med;

	Instance List Colleague col;
```

```
	Relation MediatorCreation
	{
		Contract Mediator newMediator;

		med = newMediator.new();
	}

	Relation ColleagueCreation
	{
		Contract Colleague newColleague;

		Instance c;

		c = newColleague.new(),

		col.Add(c)+;

		choice(Parameters.IsDynamic) == true
		{
			med.Register(c)+;
		}
	}

	Relation MediatorCommunication
	{
		Contract Colleague col;
		Contract Mediator med;

		col.Changed()+, med.ColleagueChanged(col,dontcare)+,
		choice(Parameters.PushVariant) == false
		{
			Belief collectInformation("Mediator Collects Changed
information from Colleague")+;
		}
		Belief notifiesColleagues("Mediator notifies all colleagues
of the change except for the one that made the change")
		{
			col.action(dontcare)[(col.length() - 2)];
		}+;
	}
}
```

Appendix C Memento Contracts

C.1 Generative Approach

Memento Contract

```
Contract Memento
    {
      Parameters
      {
            Scalar Boolean InstanceBind OriginatorAccessOnly;
      }

      [Parameters.OriginatorAccessOnly == true] Observability
tOriginator originator();

      Observability S state();

      Responsibility new()
      {
            Post(state() not= null);
      }

      Responsibility finalize()
      {
            Pre(state() not= null);
      }

      Responsibility S getState()
      {
            choice(Parameters.OriginatorAccessOnly) == true
            {
                  Pre(originator() not= null);
                  Belief("Only Originator who created the memento can
access this");
            }
            Pre(state() not= null);
            Execute();
      }
```

```
	Responsibility setState(S newState)
	{
		choice(Parameters.OriginatorAccessOnly) == true
		{
			Pre(originator() not= null);
			Belief("Only Originator who created the memento can
access this");
		}

		Pre(state() not= null);
		Pre(newState not= null);
		Execute();
		Belief MementoStateChanged("Memento state has been changed
to newstate")
		{
			Post(newState == state());
		}
	}

	Scenario MementoLifetime()
	{
		Trigger(new()),
		(getState()*, setState()*)*,
		Terminate(finalize());
	}

	Exports
	{
		Type S;
		Type tOriginator conforms Originator
		{Originator::tOriginator;}
		Type tMemento conforms Memento
		{ Originator::tMemento; }
		Type tMemento conforms Memento
		{ Caretaker::tMemento; }

	}
}
```

Originator Contract

```
Contract Originator
{
	Observability S state();

	Responsibility new()
	{
		Post(state() not= null);
	}

	Responsibility finalize()
	{
		Pre(state() not= null);
	}
```

```
	Responsibility setState(S newState)
	{
		Pre(state() not= null);
		Pre(newState not= null);
		Execute();
		Belief OriginatorStateChanged("Originator state has been
changed to new state")
		{
			Post(newState == state());
		}
	}

	Responsibility tMemento createMemento()
	{
		Pre(state() not= null);
		Execute();

		Belief MementoReturned("Memento has been returned with the
current state")
		{
			Post(value.state() == state());
		}
	}

	Responsibility setMemento(tMemento m) ;
	{
		Pre(state() not= null);
		Pre(m not= null);
		Execute();
		Belief OriginatorStateChanged("Originator state has been
changed to Memento state")
		{
			Post(state() == m.state());
		}
	}

	Scenario OriginatorLifetime()
	{
		Trigger(new()),
		((setState()*, createMemento()*)*, setMemento()*)*,
		Terminate(finalize());
	}

	Exports
	{
		Type tMemento conforms Memento
		{ Originator::tMemento; }

		Type tOriginator conforms Originator
		{Memento::tOriginator;}
		Type tOriginator conforms Originator
		{ Caretaker::tOriginator; }

		Type S;
	}
}
```

CareTaker Contract

```
MainContract CareTaker
{
	Parameters
	{
		Scalar Boolean InstanceBind SingleMemento;
	}

	Structure
	{
		Belief hasMemento("Caretaker has Memento to take care of")
		{
			hasMemberOfType(tMemento);
		}
	}

	Observability tOriginator o;

	[Parameters.SingleMemento == false] Observability List tMemento
memento();

	[Parameters.SingleMemento == true] Observability Scalar tMemento
memento();

	Responsibility requestMementoBeMade()
	{
		once Scalar tMemento oldMemento;
		once Scalar Integer numMemento;

		choice(Parameters.SingleMemento == false)
		{
			numMemento = memento().count();
			Pre(memento() not= null);
		}
		alternative
		{
			oldMemento = Memento();
		}

		Execute();

		Belief MementoMade("A new memento has been created")
		{
			choice(Parameters.SingleMemento == false)
			{
				Post(memento().count() > numMemento);
			}
			alternative
			{
				Post(Memento() not= oldMemento);
			}
		}
	}
```

```
	Responsibility returnMemento(Optional tMemento m)
	{
		Belief HasMementoToRemove("There exists a memento to
remove")
		{
			choice(Parameters.SingleMemento == false)
			{
				Pre(memento().contains(m) == true);
			}
			alternative
			{
				Pre(memento() not= null);
			}
		}
		Execute();
		Belief MementoRemoved("Memento has been removed")
		{
			choice(Parameters.SingleMemento == false)
			{
				Post(memento().contains(m) == false);
			}

			alternative
			{
				Post(memento() == null);
			}
		}
	}
	Scenario CaretakerLifetime()
	{
		Trigger(new()),
		requestMementoBeMade()*,
		returnMemento()*;
		Terminate(finalize());
	}

	Exports
	{
		Type tMemento conforms Memento
		{ Memento::tMemento; }
		Type tOriginator conforms Originator
		{ Originator::tOriginator; }
		Type S;
	}
}
```

Memento Interaction Class

```
Interaction MementoInteraction
{
	Relation SaveState()
	{
		Contract CareTaker caretaker;
		Contract Originator originator;
		Contract Memento memento;

		(
			caretaker.requestMementoBeMade(),
			oiginator.createMemento(),
			memento.new(),
			memento.setState();
		)*

	}

	Relation RestoreState()
	{
		Contract CareTaker caretaker;
		Contract Originator originator;
		Contract Memento memento;

		(
			caretaker.returnMemento(),
			originator.setMemento(),
			memento.getState();
		)
	}
}
```

Appendix D Factory Contracts

D.1 Generative Approach

Creator Contract

```
Contract Creator
{
	Parameters
	{
		Scalar Boolean InstanceBind StandardFactory;
		Scalar Boolean InstanceBind CreatorMapped;
		Scalar Boolean InstanceBind ReconfigurableCreator;
		Scalar Boolean InstanceBind HasPrototype;
		Scalar Boolean InstanceBind HasNoInterface;
		Scalar Boolean InstanceBind OnlyOneTypeOfProduct;
		Scalar Boolean InstanceBind ProductSetInOtherClass;
		Scalar Boolean InstanceBind HasRepository;
	}

	Structure
	{
		choice(Parameters.HasNoInterface == true)
		{
			Belief NoInterface("Has noInterface")
			{
				HasInterface(false);
			}
		}
		alternative
		{
			Belief HasInterface("Has an Interface")
			{
				HasInterface(true);
			}
		}

		choice(Parameters.ProductSetInOtherClass == true)
		{
			Belief hasConfigFile("Solution that this creator is
part of contains a Config Client Class")
			{
				ProjectHasClassOfType(tConfigClient);
			}
		}
	}


	[Parameters.CreatorMapped == true] Observability tProductType
isMappedTo(tProductType productID);
```

```
	[(Parameters.ReconfigurableCreator == true) |
(Parameters.HasPrototype == true) | (Parameters.HasNoInterface == true)
| (Parameters.OnlyOneTypeOfProduct == true)] Observability tProductType
productType();

	[(Parameters.ProductSetInOtherClass == true) |
(Parameters.HasRepository == true)] Observability tProductType
getProductCreationType();

	[Parameters.HasRepository == true] Observability
tRepositoryPointer getRepositoryPointer();

	Responsibility new()
	{
		choice((Parameters.ReconfigurableCreator == true) |
(Parameters.HasPrototype == true) | (Parameters.HasNoInterface == true)
| (Parameters.OnlyOneTypeOfProduct == true))
		{
			Belief CreationConfigured("Product creation has been
configured to type " + productType())
			{
				Post(productType() not= null);
			}
		}
		alternative(Parameters.HasRepository) == true
		{
			Belief RepositoryLocated("A repository location has
been indicated.")
			{
				Post(getRepositoryPointer() not= null);
			}
		}
	}

	Responsibility finalize() {}

	Responsibility tProduct create(Optional tProductType productID)
	{
		Execute();
		Post(value not= null);

		Belief ProductCreated("A product has been created")
		{
			choice(Parameters.StandardFactory == true)
			{
				Post(value.isType(tProduct) == true);
			}
			alternative(Parameters.CreatorMapped == true)
			{
				Post(value.getType() == isMappedTo(productID));
			}
			alternative(Parameters.ReconfigurableCreator == true)
			{
				Post(value.getType() == getProductType());
			}
```

```
                  alternative(Parameters.HasPrototype == true)
                  {
                        Post(value.getType() ==
getPrototype().getType());
                  }
                  alternative((Parameters.HasNoInterface == true) |
(Parameters.OnlyOneTypeOfProduct == true))
                  {
                        Post(value.getType() == productType());
                  }
                  alternative((Parameters.ProductSetInOtherClass ==
true) | (Parameters.HasRepository == true))
                  {
                        Post(value.getType() ==
getProductCreationType());
                  }
            }
      }

      [Parameters.ReconfigurableCreator == true] Responsibility
tProductType getProductType()
      {
            Execute();
            Post(value not= null);
      }

      [(Parameters.ReconfigurableCreator == true) |
(Parameters.HasPrototype == true) | (Parameters.HasNoInterface ==
true)] Responsibility configureClient(tProductType productID)
      {
            Execute();
            Belief CreationConfigured("Product creation has been
configured to type " + productID)
            {
                  Post(productType() == productID);
            }
      }

      [Parameters.HasPrototype == true] Responsibility tProduct
getPrototype()
      {
            Execute();
            Post(value not= null);
            Belief PrototypeCreated("A prototype has been created of
type " + getProductType())
            {
                  Post(value.getType() == productType());
            }
      }
```

```
        Scenario ConcreteCreatorLifetime()
        {
              choice(Parameters.ReconfigurableCreator == true)
              {
                    Trigger(new()),
                    create()*,
                    (configureClient() | getProductType())*,
                    Terminate(finalize());
              }
              alternative(Parameters.HasPrototype == true)
              {
                    Trigger(new()),
                    create()*,
                    (configureClient() | getPrototype())*,
                    Terminate(finalize());
              }
              alternative(Parameters.HasNoInterface == true)
              {
                    Trigger(new()),
                    (create()* | configureClient())*,
                    Terminate(finalize());
              }
              alternative
              {
                    Trigger(new()),
                    create()*,
                    Terminate(finalize());
              }
        }

        Exports
        {
              Type tProduct conforms Product
              { Product::tProduct; }

              Type tProductType;
              Type tRepositoryPointer;

              Type tCreator conforms Creator
              { Product:: tCreator; }

              Type tConfigClient conforms ConfigClient
              { ConfigClient:: tConfigClient; }
        }
}
```

Product Contract

```
Contract Product
{
	Parameters
	{
		Scalar Boolean InstanceBind EnforcingAbstractClass;
		Scalar Boolean InstanceBind EnforcingInterfacePresence;
	}

	Structure
	{
		choice(Parameters.EnforcingAbstractClass == true)
		{
			Belief ImplementsAbstractClass("Class implements an
abstract class")
			{
				HasAbstractParent(true);
			}
		}

		choice(Parameters.EnforcingInterfacePresence == true)
		{
			Belief ImplementsInterface("Class implements an
interface")
			{
				HasInterface(true);
			}
		}

		Belief hasACreator("Product has been created by a creator
class")
		{
			HasMemberOfType(tCreator);
		}
	}

	Observability Boolean isType(tProduct product);
	Observability tProductType getType();

	Responsibility new() {}
	Responsibility finalize() {}

	Exports
	{
		Type tProductType;
		Type tCreator;

		Type tCreator conforms Creator
		{ Creator:: tCreator; }

		Type tProduct conforms Product
		{ Creator::tProduct; }

	}
}
```

ConfigClient Contract

```
Contract ConfigClient
{
	Responsibility new() {}

	Responsibility finalize() {}

	Responsibility Creator<tProductType> instanciateCreator()
	{
		Execute();
		Post(value not= null);
		Belief CreatorInstanciated("A creator has been created
which can create products of type " + tProductType)
		{
			Post(value.getProductType() == tProductType);
		}
	}

	Scenario ConfigClientLifetime
	{
		Trigger(new()),
		instanciateCreator()*,
		Terminate(finalize());
	}

	Exports
	{
		Type tProductType;
	}
}
```