Design Patterns – A Modeling Challenge

Vojislav D. Radonjic, Soheila Bashardoust, Jean-Pierre Corriveau, and Dave Arnold
School of Computer Science, Carleton University
Ottawa, Canada
radonjic@acm.org

Abstract – *Design patterns and their catalogs are an important phenomenon in software development, and they present a challenge to modeling techniques and supporting tools. In this paper we identify and illustrate the problem in terms of the creational family and one of its members, the Factory Method.*

Keywords: *design patterns; variability modeling; variant selection*

1. Introduction: Design Patterns

Designers are bridge builders from the problem world of requirements to the solution world of various candidate systems. Design patterns have emerged as important tools for documenting and sharing the 'bridge building' experience of spanning from the side of requirements (scenarios and features) to the side of implementations (C++, Java, various run-times). Like craftsmen in other disciplines, software designers place great importance on their tools, and judging by the large number of copies of the GoF catalog [3] sold, and by the ubiquity of these patterns in newly developed systems, these craftsmen have chosen design patterns as one of their essential tools.

Design patterns, however, remain a challenge for the modeling community. The core pattern conceptualization – the often repeated 'a design solution to a problem in context' – is considerably more complex, as we will see in Section 2. In Section 3 we summarize the modeling challenges.

2. Design Patterns: a Challenge to Modeling

Design Patterns are complex abstractions that span from requirements through design to implementation, with variability at each phase. In the remainder of this section we illustrate the challenge of modeling the creational family, and in particular, the Factory Method from the GoF [3]; our modeling goal is to explicit the concepts, their variations and connections as they appear in various sections of the template format used in the GoF catalog.

2.1 Example: Design Level Requirements For The Creational Family Of Patterns

Let us start at the source, the GoF catalog[3].

"Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. ...

Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class." p81 [3]

The above two paragraphs begin the presentation of creational patterns: they establish the fundamental **goal** of this family of patterns as ***System Creation*** in a *specific context*. The context is what leads to a particular kind of refinement of the goal of system creation that is based on object-orientation; where a system is composed of objects (components), whose structure and behavior are defined in terms of classes (types). It is this context that governs the space of design mechanisms used to achieve the goal.

"There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. [component types] Consequently, the creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when. They let you configure a system with "product" objects that vary widely in structure and functionality. Configuration can be static (that is, specified at compile-time) or dynamic (at run-time)." p81 [3]

The goal of *System Creation* is that of creation and initialization of the system structure out of a set of component types. The previous paragraph mixes the what and the how, the goal and the mechanisms used to achieve it. Our aim is to separately model the goal of system creation from the mechanism used to achieve it.

From a designer's perspective, the goal of system creation is refined in terms of component types and their relationships (family constraints, initialization ordering constraints, incompatibilities between component types, etc.), system object structure, binding place and time. The forces leading to selection of one pattern over another, therefore, are in these terms:

- Sets of component types: static, dynamic, family of component types
- System Structure
- Uniqueness (Singletoness)

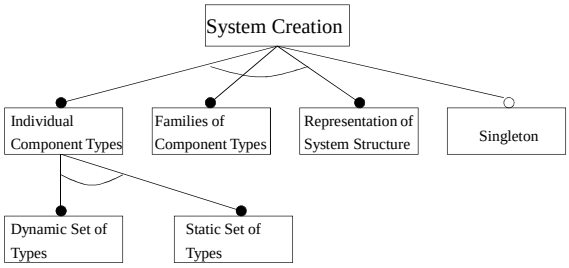

Figure 1: Top-level Feature Model for the Creational Family

Fig. 1 models system creation in terms of a feature model, commonly used in domain engineering [7]. It starts at the root with "System Creation" and refines to a tree of concepts that embodies all the creational patterns.

In the context of design patterns, domain analysis is about building a space of forces that is a starting point for selecting a design variant. The feature model in Fig. 1 models that space of forces. The syntax and general notational semantics are not important — rather, what is important is the internal relations within the feature model and, in particular, its overall relationship to the design space in which the design variants of a pattern live. *Most of the content found in the patterns of GoF is in that relationship of forces to variants.* The problem in the GoF catalogue is that that relationship is not clearly expressed. The aim of the coming description is to explicit the relationship of forces to variants. With that in mind, it is critical to keep the point of view of the designer as the primary customer of this modeling exercise.

The notion of a factory object is not represented in Fig. 1, because factories are the "how" part of the pattern, which we are explicitly leaving out of Fig. 1. Put another way, a factory is a solution abstraction, not a problem abstraction. Factories are a generalization of creation of an object to 1) creation of any object that fulfills a particular interface (i.e. type), 2) creation of a system of objects (abstract factory, builder, ...).

In Fig. 1 the terms *component* and *component type* are used as opposed to objects and classes to indicate independence from OO mechanisms. Specifically, should the following content be part of the feature model?

"Creational patterns become important as systems evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class. " p81 [3]

The above paragraph defines the context of system creation in OO rather than the design mechanisms used in the patterns, and, therefore, its content is eligible for feature modeling. On the other hand, the following paragraph from the concluding discussion about creational patterns is filled with solution mechanics of each pattern, which should not make it into the feature model directly:

"There are two common ways to parameterize a system by the classes of objects it creates. ….

Prototype has the factory object building a product by copying a prototype object. In this case, the factory object and the prototype are the same object, because the prototype is responsible for returning the product."p135 [3]

But the following closing paragraph of the creational patterns would make it into feature modeling because it addresses design-level requirements directly:

"Designs that use Abstract Factory, Prototype, or Builder are even more flexible than those that use Factory Method, but they're also more complex. Often, designs start out using Factory Method and evolve toward the other creational patterns as the designer discovers where more flexibility is needed. Knowing many design patterns gives you more choices when trading off one design criterion against another." p136 [3]

The above paragraph is particularly strong in saying how to navigate and select—being able to easily navigate a catalogue is important to allow the designer to build and refresh a mental map of the design space. This is not simply a surface-level usability issue, although it may exhibit itself as such, but a much deeper semantic one. As the scale of design-spaces grows (witness the scale of options available in J2EE, .NET, etc.), the decision-making support offered by usable cognitive models becomes more than just a deep academic exercise, it becomes a critical practical issue.

It may appear that the only sources for the Fig. 1 Feature Model would be the Intent, Motivation and Applicability sections of the GoF format [3] but, surprisingly, the one that yields the most information is the Implementation section.

2.2 Example: Design Level Forces For The

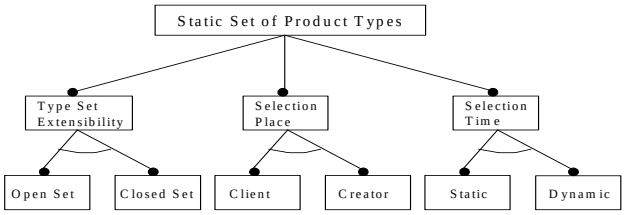

Figure 2: Factory Method Feature Model

Factory Method Pattern

The Factory Method Feature Model (Fig. 2) is the result of domain analysis of the pattern, and fits within Fig 1. (Static Set of Types box). The analysis leading to Fig. 2 focuses on the problem space of a design pattern, and has as its primary customer the designer in a role of a navigator towards a potential solution. Consequently the resulting feature model is a kind of selection/navigation mechanism. There is, however, a deeper aspect to feature modeling — a representation of context within which the solution space of a pattern exists. This context is multi-faceted and dependent

on perspective of the user of the pattern: extensibility, performance, resource costs, etc.

Consider that a designer arrives to this pattern under forces that push towards the overall creational pattern space. The selection of one pattern over another, and one variant over another takes place as a result of differentiation and specialization of the forces: the overall goal of creating a system of components is differentiated into forces described in Fig. 1.

The notion of product type set is central to the Factory Method pattern as well as Abstract Factory and Prototype. [3] The objects that are created (often to serve as delegates to some client) have types from that set of product types. In Factory Method pattern the set is statically bound, i.e. the collection of types is determined before run-time. A designer would think of this feature as a **Static Set of Product Types**.[1]

If a designer needs to have an **Open Set** of types handled by the pattern (as in, for example, vector<T> in STL where user of vector abstraction can specify and add to the existing pool of built-in C++ element types, i.e. int, char, double ...) we will see that there are several variants within the Factory Method design space that can help. The **Closed Set** indicates that the designer only requires a fixed number of product types to be handled (as in, for example, vectorInt, vectorChar, vectorDouble, ... all specialiazing vector type).

A particular client's request for creation of an object may be statically bound to a product type, or that choice may be delayed until run-time. The choice is captured by the notion of **Selection Time** and the **Static** or **Dynamic** alternatives.

The **creator** provides the factory method that returns the newly created product object. The **creator** role concept is not a top-level feature, as it does not concern the initial step of decision making, but it does enter into the picture in the choice between who makes a selection decision about the product type: the **client** that calls the factory method, or the **creator** that provides that factory method. The choice is captured by the notion of **Selection Place** and the **Client** or **Creator** alternatives.

2.3 Example: Design Variants For The Factory Method

In analyzing the Factory Method pattern we identify 8 distinct design variants, most of which are found in the implementation section. In the GoF format [3] the forces that would lead a designer to choose one variant over another are, at best, mentioned where the design space element is discussed, and at the worst, entirely omitted.

The core design idea in the Factory Method pattern is the **factory method** itself—a function that returns a newly created object to its **client**, decoupling its **client** from the specific type of **product**. The **factory method** is a delegate responsible for **creation**, and the **creator** class is a wrapper around that **factory method**. All the variants contain this core design idea in some form—this pattern's invariant is the factory method idea.

Now we explore the variants constituting the design space of the Factory Method pattern [3].

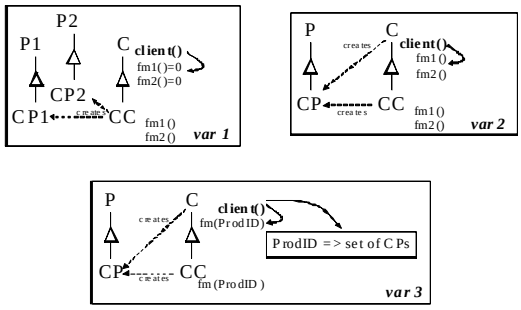

Figure 3. Variants 1-3

Variant-1 in Fig. 3, has the following elements:

P1, P2 — product interface for product category 1 and 2 respectively

CP1, CP2 — concrete product for product category 1 and 2 respectively

C — creator interface

fm1(), fm2() in C — factory method interface only

CC — concrete creator

fm1(), fm2() in CC — factory method implementation

client() — product client which is a template method within creator

Variant-1 elaborates the core idea by separating the factory method interface from its implementation. The different factory method implementations of that interface provide the mappings to the different concrete product types. There is a product hierarchy for each category of product: buttons and menus would be examples of categories and within these there are different types of buttons and menus, simple and fancy ones, for instance. This is modeled with P1, CP1 and P2, CP2 hierarchies corresponding to category-1 and category-2. The factory method interface in C, fm1 and fm2, map to category-1 and category-2 respectively, while the different concrete creators map to the concrete types within these categories. Client is typically a template method in the creator C itself, but the client can also be external to the creator. In subsequent variants we just model a single product category for the sake of not cluttering the diagrams. We do imply multiple categories by the existence of multiple factory methods.

The client is not simply a client of the product individually, but of the pattern as a whole: we can think of a "Factory Method" interface that a client has to follow to participate in the pattern contract. The basis of that contract

[1] Phrases or words in bold are concepts directly modeled in the corresponding feature model (Fig. 2).

is the client's assumption that there is a create-association from the creator hierarchy to the product class hierarchies. This is typical for design patterns: they present a contract-oriented interface to their clients.

Variant-2 in Fig. 3 has the same elements as Variant-1, except for the following difference:

fm1(), fm2() in C — factory method interface and *default* implementation

Variant-2 elaborates the core idea in much the same way as Variant-1, with the minor difference that fm1, fm2 in C provide *default* implementations. Hence a direct create association from creator C to a concrete product CP.

Variant-3 in Fig. 3 has the same elements as Variant-1, except for the following differences:

fm(ProdID) in C and CC — factory method interface and *default* implementation

ProdID=>set of CPs — a map between ProdID values and CPs (concrete product types)

Variant-3 elaborates the core idea by providing a parametrized factory method interface, and a mapping between the parameter values and concrete product types. This parametrized interface allows a single factory method to return multiple types of products, allowing the designer to replace all the different factory methods of variant-1 and variant-2 with a single factory method. In a statically typed language like C++ this means that fm(ProdID) will have a return type of Super-P, which will be a super-type for all the product categories. This may present a problem in that the client may want to invoke category-specific operations (menu specific operations, for example, as opposed to button specific ones). In that case the solution is to recover the type on client side through dynamic casting. In dynamically typed languages this is not an issue.

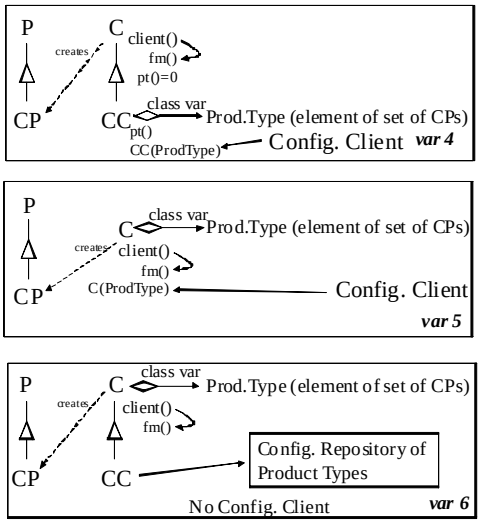

Figure 4. Variants 4-6

Variant-4 elaborates the core idea by having the factory method fm() delegate to CC, the concrete creator, the choice of CP. The factory method fm() in C is fully implemented but dependent on method pt() which returns the product type to be created. The method pt(), implemented in CC, simply returns type of product (not the product itself).

Variant-4 in Fig. 4 has the same elements as Variant-1, except for the following differences:

fm() in C only — factory method implementation, uses pt()

pt() in C and CC — method that returns type of product, with interface in C and implementation in CC

Config. Client, CC(ProdType) — Config. Client conFig.s CC with a particular type CP

ProdType — type variable with values from set of CPs

Variant-4 has a direct implementation in dynamically typed languages such as Smalltalk, but in statically typed languages, where types are typically not directly accessible at run-time, it is more appropriate to shift to a compile-time mechanism where types are visible. C++ templates are one such mechanism, although other meta-level mechanism are conceivable. Generally, when a design requires some kind of a type-computation, a dynamically-typed language would provide a direct implementation, whereas a statically-typed language would require use of a meta-level facility normally available only during compile-time. C++ does provide a run-time RTTI (Run-Time Type Identification) mechanism but that is both awkward and carries a performance penalty relative to a template based approach.

Variant-4 can be implemented using the Prototype pattern[3], where pt() would return a prototype object (rather than product type), on which the factory method fm() would invoke clone() operation.

Variant-5 in Fig. 4 differs from variant-4 in that it replaces method pt() with a class variable Prod. Type. Now, instead of subclassing creator C to provide different implementations of pt(), the creator C simply needs to be configured with a particular value for Prod. Type. The class variable eliminates the need for subclassing of C.

Variant-5 in Fig. 4 has the same elements as Variant-4, except for the following differences:

No CC, no pt() — there is only single creator class C; there is no method pt()

Config. Client, C(ProdType) — Config. Client configures C with a particular type CP

ProdType — type variable with values from set of CPs

Variant-6 in Fig. 4 is similar to Variant-5, but it eliminates the Config. Client by subclassing C and delegating to a concrete creator CC the selection of a value for Prod. Type from set of Product Types. The selection would take place in the constructor of CC.

Variant-6 in Fig. 4 has the same elements as Variant-4, except for the following differences:

No pt() —there is no method pt()

ProdType — type variable with values from set of CPs

Config. Repository of Product Types — repository of CP types

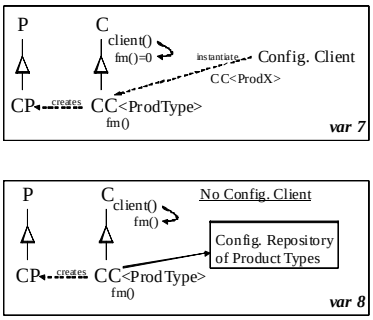

Figure 5. Variants 7-8

Variant-7 elaborates the core idea by type-parametrization of the concrete creator CC with the product type to be returned by the factory method fm(). This reduces the number of concrete creators to only one, regardless of the number of product types that need to be served by the factory method. The configuration client, Config. Client, passes the product type, ProdType, to the concrete creator.

Variant-7 in Fig. 5 has the same elements as Variant-1, except for the following differences:

CC — type-parametrized with the product type.

Config. Client, CC<ProdType> — Config. Client configures CC with a particular type CP

ProdType — type variable with values from set of CPs

Variant-8 in Fig. 5 is similar to Variant-7, but it eliminates the Config. Client by having a configuration repository which provides values for ProdType.

Variant-8 in Fig. 5 has the same elements as Variant-7, except for the following differences:

No Config. Client —there is no configuration client for CC

Config. Repository of Product Types — repository of CP types

With Variant-8 we conclude the description of the design space for the Factory Method pattern. From a designer's perspective the choice of inheritance, with and without templates, or parametrized factory methods, and other mechanisms are all refinements under the influence of forces modeled in Fig. 2. These refinements are aimed to have a direct effect on the **factory method**, and only an indirect one on the **creator**. A purely functional version of this pattern can be envisioned, where there is no creator class, but only a free-standing factory method.

The core idea is a seed out of which the variants emerge to form the design space. This is the basis for the generative process driven by the forces in Fig. 2. All the variants, therefore, contain this design idea in some form. We could envision the core idea of this design pattern as being selected, and then refined and adapted by the designer under the forces modeled in Fig. 2.

2.4 Example: Mapping Design Forces To Design Variants For The Factory Method Pattern

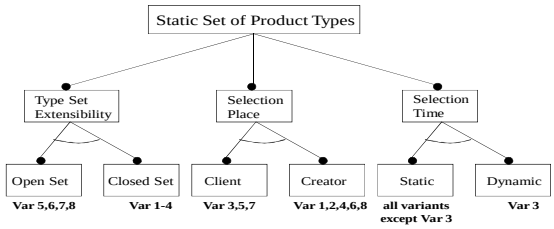

Figure 6. Factory Method Feature Model to Design Space Mapping

Fig. 6 illustrates a mapping from a model of forces influencing a designer to variants provided by the Factory Method pattern. As the feature model indicates, there are a lot more choices left once the pattern itself is chosen: Is the set of product types open or closed? Who makes the selection and when? Is there a dependence between product types that needs to be addressed at creation time? Analysis of these questions and their answers gives rise to the feature model (Fig. 2), variants constituting the design space (Fig. 3-5), and most significantly, a mapping from one to the other (Fig. 6).

An **Open Set** feature indicates that the designer needs to allow for expansion in the set of product types served to the client. The approach used is to externalize, through parametrization, the configuration of creator with the product type. Variants 5,6 make that externalized product type a run-time parameter which makes them more suitable for dynamically typed languages (eg. Smalltalk), while Variants-7,8 make them a compile-time parameter through the mechanism of type-parametrization (generic types). Clearly, Variants-7,8 imply a language like C++ with its template mechanism.

Variant-3 in Fig. 3 is not considered to support the **Open Set** feature because the mapping of ProdID to concrete product types is implemented programmatically inside the factory method fm(ProdID) as illustrated by the following code (p111 [3]) of:

```
class Creator {
        public:
           virtual Product* fm(ProductID);
        };

        Product* Creator::fm (ProductID id) {
           if (id == MINE)  return new MyProduct;
           if (id == YOURS) return new YourProduct;
           // repeat for remaining products...

           return 0;
        }

Product* MyCreator::fm (ProductID id) {
```

```
    if (id == YOURS)  return new MyProduct;
    if (id == MINE)   return new YourProduct;
       // N.B.: switched YOURS and MINE

    if (id == THEIRS) return new TheirProduct;

    return Creator::fm(id); // called if all others fail
}
```

A **Closed Set** feature indicates that set of product types is not extensible, and its choice will lead to Variants-1-4 (fig. 3-5) part of the design space. These variants are closed with respect to set of product types because they hardwire that in the definition of the concrete **factory methods**, as opposed to Variants-5,7 (Fig. 4 and Fig. 5) which provide for a configuration interface.

Who makes the selection and when? Is the selection (mapping of factory method to product type) made by the client or by the creator? Is it made at the time creator is defined, or when it is initialized, or when the client requests a product object?

"Who makes the selection?" is also a question of the interface the pattern presents to the client. Does the client need to make the selection? If the client needs to control the mapping, i.e. **Selection Place->Client**, then designer is led to those variants that externalize the mapping: Variants-3,5,7 (Fig. 3-5). If the client does not need to control that mapping, i.e. **Selection Place->Creator**, then Variants 1,2,4,6,8 (Fig. 3-5) would be designer's choices. In Variant-3 (Fig. 3), as seen in the code shown above, the definition of the mapping is hidden from the client, but the selection based on that mapping is available to the client. Hence, an explicit element in Variant-3 that shows the ProdID=>CP map implied by the fm() implementation. The intent in this variant is to replace N factory methods with a single one: fm1(), fm2() ... fmN() as a mechanism for allowing client choice of product categories 1,2 ... N (not choices of concrete product types for which the inheritance mechanisms is used) is replaced by fm(ProdID). More generally, the map can be envisioned as a way to externalize the relationship of fm(ProdID) to concrete product types and categories, and thereby provide for a reconfigurable way of mapping client creation requests to product types and categories. In this case the factory method fm(ProdID) could simply dispatch on ProdID using the shared ProdID=>CP map.

"When is the selection made?" has an answer that is dependent on the implementation language used. In a statically typed languages like C++ and Java, only Variant-3 (Fig. 3) offers **Selection Time** that is **Dynamic**; the rest of the variants are all **Static**, with a possibility of "simulating" dynamic selection for Variant-5 if an additional dynamic configuration interface is added (eg. changeProdType(prototype)) that complements the existing C(ProdType) interface. The "simulation" is in using a change in prototype object to simulate change in actual product type. This level of language-dependent detail could be modeled as further refinements in the Feature Model with corresponding variants in the design space. The point is that the Feature Model captures the decision making information in layers, with upper layers modeling architectural and design level decisions and lower ones, the detailed-design and implementation level decisions.

Other Creational Patterns

As we have already mentioned the core design idea in the Factory Method pattern is the **factory method** itself—a function that returns a newly created object to its **client**, decoupling its **client** from the specific type of **product**. The **factory method** is a delegate responsible for **creation**. This core idea is shared to some extent with all the creational patterns in [3]. Factory Method puts the creator role inside the client itself, making the client a template method within an existing class (eg. CreateMaze(), a template method, is a member function of the MazeGame class p114[3]). Abstract Factory separates the creator from the client completely, thereby allowing for dynamic selection of creator. Prototype makes things even more flexible: the creator (factory method) can be configured at run time. Builder is about building arbitrary structures rather than individual products and singleton defines a kind of factory method that ensures uniqueness.

3. Conclusion

Variability

The Factory Method analysis illustrates in some detail that the core pattern conceptualization – a design solution to a problem in context – is better described as a family of design solutions to a family of problems in a family of contexts: variability within and across each component of a pattern is fundamental. In [17] we investigate potential modeling solutions to deal with variability.

Even a basic and fundamental pattern like the Factory Method has significant complexity: Fig. 2 models a family of design problems, Fig. 3-5 model a family of design solutions, and Fig. 6 models a mapping from problem family to the solution family.

Navigation and Selection

The variants found in each pattern are points in some design subspace: the 8 variants of the Factory Method pattern define a Factory Method subspace in the larger Creational pattern design space. We can imagine the designer as a navigator in that space, moving towards one pattern subspace over another under the influence of forces, and selecting a particular point, i.e. variant, in that space. In these terms, the GoF format provides a first-level, sparse description of both the space defined by a particular pattern, and the forces acting upon a designer to select a pattern.

Evaluating Choices

How do we evaluate that the chosen variant satisfies the specific goal and that the implementation satisfies the properties of the design?

We are currently developing a technique based on the use of Another Contract Language (ACL) as a pattern modeling language where the corresponding implementations are evaluated using the Validation Framework [14].

Our research goals are to deal with these challenges:

- Traceability [18] from goals and intents to variations and their implementations,
- Variability modeling in analysis, design, and implementation, [17]
- Interaction between traceability and variability, [17] and
- Evaluation of selected variants relative to intents and detailed design properties, and corresponding implementations, [12] and
- Development of tools that would assist designers in evaluating implementations. [12]

4. Acknowledgments

Support from the Natural Science and Engineering Research council of Canada is gratefully acknowledged.

5. References

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

[2] C. Alexander, *The Timeless Way of Building* (New York, NY: Oxford University Press, 1979).

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

[4] F. Buschman, & K. Henney, A Distributed Pattern Language. Proc. Seventh European Conference on Pattern Languages of Programs, Irsee, Germany, 2002.

[5] W. Pree, *Design Patterns for Object-Oriented Software Development* (Reading, MA: Addison-Wesley, 1994).

[6] M. Fontoura, W. Pree, & B. Rumpe, *The UML Profile for Framework Architectures* (Reading, MA: Addison-Wesley, 2002).

[7] K. Czarnecki, & U.W. Eisenecker, *Generative Programming: Methods, Tools, and Application,* (Reading, MA: Addison-Wesley, 2000).

[8] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, & M. Stall, *Pattern-Oriented Software Architecture - A System of Patterns* (Chichester, West Sussex, England: Wiley and Sons, 1996).

[9] D.C. Schmidt, & S.D. Huston, S.D., *C++ Network Programming: Mastering Complexity With ACE and patterns* (Reading, MA: Addison-Wesley, 2002).

[10] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied,* (Reading, MA: Addison-Wesley, 2001).

[11] K. de Volder, *Implementing Design Patterns as Declarative Code Generators*, 2001- http://www.cs.ubc.ca/~kdvolder/

[12] V.D. Radonjic, *An Open Pattern-based Framework for Evaluating Software Systems*, PhD. Thesis (in preparation) Carleton University, School of Computer Science.

[13] K. Beck, & R. Johnson, *Patterns Generate Architectures*. Proc. ECOOP'94, Bologna, Italy,1994.

[14] D. Arnold, *Validation Framework and Another Contract Language*, http://vf.davearnold.ca/

[16] M. Antkiewicz, K. Czarnecki, and M. Stephan, *Engineering of Framework-Specific Modeling Languages*, IEEE Trans. Software Eng., vol. 35, no. 6, pp. 795-824, Nov/Dec 2009.

[17] S. Bashardoust, V.D. Radonjic and J.-P. Corriveau and D. Arnold, *Challenges of Variability in Model-Driven and Transformational Approaches: A Systematic Survey*, Workshop on Variability in Software Architecture, WICSA2011, Boulder, Colorado, USA, 2011.

[18] J.-P. Corriveau, *Traceability Process for Large OO Projects*, IEEE Computer, pp. 63-68, Sep 1996.