An XVCL-based Approach to Software Product Line Development

Hongyu Zhang and Stan Jarzabek
Department of Computer Science, School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 117543
{zhanghy, stan}@comp.nus.edu.sg

Abstract

In recent years, software product line approach has emerged as a promising way to improve software productivity and quality. How to effectively handle variants (including functional variants, variant design decisions, implementation-level variants) has been a major challenge in product line development. We apply XVCL (XML-based Variant Configuration Language), a variability mechanism based on frame technology [1], to product line development. In XVCL, x-frames represent domain knowledge in the forms of product line assets. Specific systems, members of a product line, can be constructed by reusing the x-frames. In this paper, we describe an XVCL-based approach to software product line development, using examples from our product line project on Computer Aided Dispatch domain.

1 Introduction

In recent years, software product line approach, initiated by Parnas back in the 1970s [13], has emerged as a promising way to improve software productivity and quality. A product line (also called product family or system family) arises from situations when we need to develop multiple similar products for different clients, or from a single system over years of evolution.

Members of a product line share many common requirements and characteristics. They may perform similar tasks, exhibit similar behavior, or use similar technologies. While having much in common, members of a product line also differ in certain requirements, design decisions and implementation details. The variability stems from many sources such as customer's specific needs, mutability of the environment, system maintenance and evolution, and so on. In the product line approach, we identify both commonality and variability in a domain, and build generic and adaptable assets such as domain model, product line architecture, generic components, etc. For developing specific products, we reuse the product line assets instead of working from scratch.

Variants (including functional variants, variant design decisions and implementation-level variants) result from the variability in a domain. Product line assets should be generic and flexible enough to accommodate the variants, and to be reusable across members of the product line. However, there could be a large number of variants in a product line. The explosion of possible variant combinations and complicated variant relationships make the manual, ad hoc accommodation and configuration of variants difficult. How to effectively handle variants in a product line is a major challenge faced by both product line researchers and practitioners.

An effective way to deal with the problem of handling variants is to design a variability mechanism [3, 9] that supports automated customization and assembly of product line assets. In Software Engineering Laboratory, we developed XML-based Variant Configuration Language (XVCL) [10, 16], a variability mechanism based on frame technology [1]. Using XVCL, we develop product line assets as a set of x-frames that are capable of accommodating both commonality and variability in a domain. X-frames represent domain knowledge in the forms of product line assets. Specific systems, members of a product line, can be constructed by composing and adapting x-frames.

In this paper, we describe our XVCL-based approach to product line development. In our approach, we perform domain analysis, capture common and variant requirements for a product line in a feature diagram. We then develop a product line architecture and generic components that populate the architecture. We apply XVCL to help us accommodate variants into product line assets and customize them to construct custom systems.

We have applied our approach to many product line projects, including projects on Facility Reservation Systems (FRS), Key-Word-In-Context (KWIC) and Computer Aided Dispatch (CAD) product lines. CAD project is a Singapore-

Ontario joint project, involving both academic and industrial partners. The project started in 2000. One of our industrial partners, Singapore Engineering Software (SES), provided CAD application domain expertise. Other partners, including National University of Singapore and Netron Inc., Toronto, provided expertise in software reuse. In this paper, we illustrate our approach using examples from the CAD project.

2   CAD Domain Analysis

Computer Aided Dispatch (CAD for short) systems are mission-critical systems that are used by police, fire & rescue, health service, and others. Figure 1 depicts a basic operational scenario and roles of a CAD system for Police.
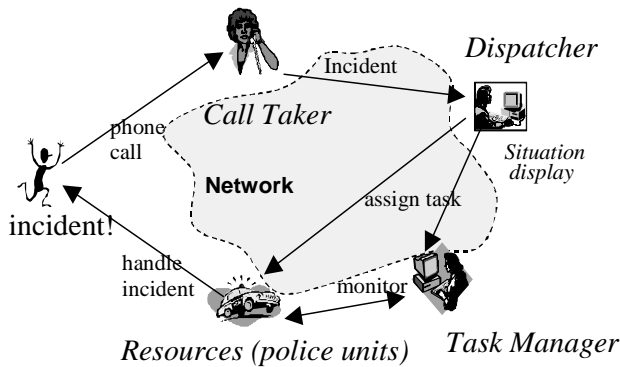

Figure 1. The basic operational scenario in a CAD system for Police

Once a Caller reports an incident, a Call Taker in command and control center captures the details about the incident and the Caller, and creates a task for the incident. The system shows the Dispatcher a list of un-dispatched tasks. The Dispatcher examines the situation, selects suitable Resources (e.g. police units) and dispatches them to execute the task. The Resources carry out the task instructions and report to the Task Manager. The Task Manager actively monitors the situation and at the end - closes the task.

2.1   Variants in CAD domain

At the basic operational level, all CAD systems are similar - basically, they support the dispatch of units to handle incidents. However, there are also differences across CAD systems. The specific context of the operation (such as police or fire & rescue) results in many variations on the basic operational scheme. We classify them into three categories such as functional variants, variant design decisions, and implementation-level variants. All these variants can be treated as variant requirements for developing CAD product line. Some of the variants we have identified are as follows.

Functional variants

1.   *Call Taker and Dispatcher roles* (CT-DISP for short). In some CAD systems, Call Taker and Dispatcher roles are separated (played by two different people), while in other CAD systems the Call Taker and Dispatcher roles are played by the same person. The CT-DISP variant has impact on system functionalities. For example, in the former case, the Call Taker needs to inform Dispatcher of the newly created task, but in the latter case, once the Call Taker creates a task, she/he can straightway dispatch Resources (e.g., Police Units) for this new task.

2.   *Validation* of caller and task information differs across CAD systems. In some CAD systems, a basic validation check (i.e., checking the completeness of the Caller and Task info) is sufficient; in other CAD systems, validation includes duplicate task checking, VIP place checking, etc.; in yet other CAD systems, no validation is required at all.

3.   *Dispatch Algorithm.* There are different ways to dispatch Resources, using shortest distance search algorithm or location code search algorithm.

Variant design decisions

4.   *Database.* The database used in CAD systems could be either centralized database or distributed database.

5.   *Encryption,* which indicates whether or not to encrypt the data sent between clients and servers.

Implementation-level variants

6.   *Package.* CAD components for different systems may belong to different Java packages.

7.   *New attributes/methods.* We include the default class attributes/methods, as well as anticipated variant attributes/methods into the classes. However, there may be additional class attributes/methods needed by a specific system, due to the possible changes in the requirements.

Besides the above variants, we have also identified other variants. In this paper, we shall not describe all the variants in detail.

2.2   Capturing variants in feature diagram

Feature diagrams [7] are often used to model common and variant product line requirements. Feature diagram provides a graphical tree-like notation that shows the hierarchical organization of the features. By traversing the feature trees, we can find out which variants have been anticipated during domain analysis. Features are classified as mandatory, optional and alternative (Czarnecki and Eisenecker also proposed the or-features [4]). Common requirements can be modeled as mandatory features whose ancestors are also mandatory. Variant requirements can be modeled as optional, alternative, and or-features. Figure 2 shows a fragment of feature diagram for CAD product line.
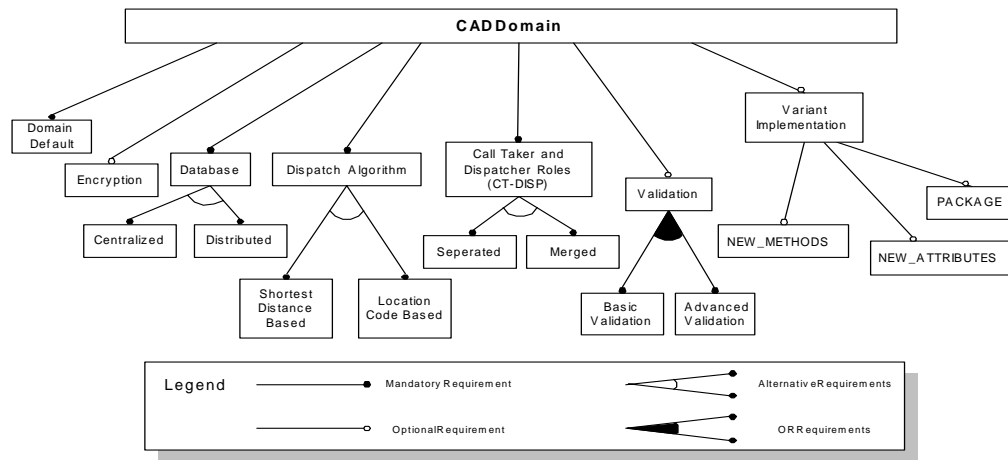
Figure 2. CAD feature diagram (partial)

3 Towards a Product Line Architecture

3.1 Developing CAD default system and runtime software architecture

In order to better understand the CAD domain, we develop a CAD default system. The default system is a domain archetype - it covers all the common requirements and also implements typical variant requirements that most product line members support. The default system is presented as a typical system in the domain.

We adopt a use-case driven, architecture centric development approach to develop the default CAD system. The problem and solution space are decomposed along the classes, which are identified from the use cases analysis and design. The CAD default system is designed as a component-based system. It is modeled in UML, implemented in Java, and deployed on a CORBA-compliant component platform.

The software architecture is the major artifact produced from the design of the default system. The software architecture of the default system is a typical software architecture for a product line. It reflects the software architecture that most product line members exhibit, and can be reused for developing future members in the product line. Here we shall highlight that the software architecture of the default system is a runtime architecture – we focus on its runtime behavior (the interacting components that are working at system runtime). Figure 3 shows the component diagram of the CAD runtime architecture.

3.2 The impact of variants on CAD runtime software architecture

After developing the runtime software architecture, we shall analyze the impact of variants on it, focusing on the following aspects:

The level of the impact

- Architecture-level impact. At architecture level, to address certain variants, we may need to include new components into the system, remove existing components from the system, modify components' interfaces, or change the allocations of the functions to components. Thus for some product line members, their runtime software architecture could be different from the runtime architecture of the default system.

For example, for the CT-DISP variant, if the Dispatcher and Call Taker roles are played by one person, then the CallTaker UI and Dispatcher UI components in Figure 3 can be merged into a single user interface component.
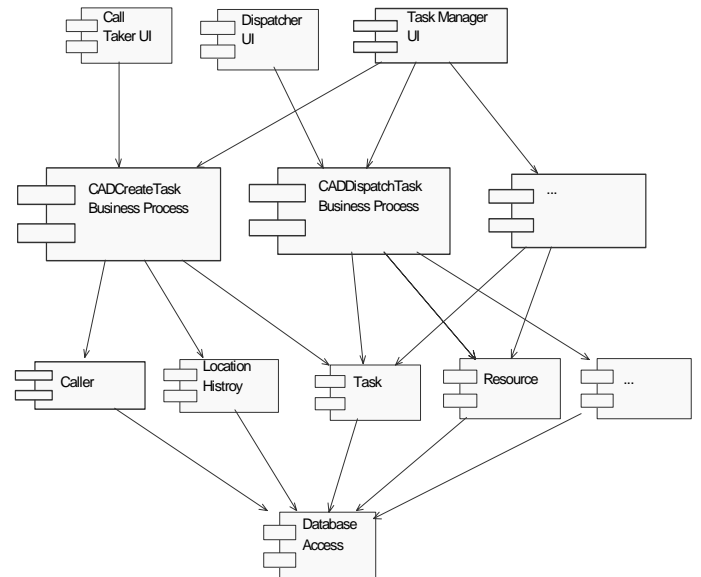

Figure 3. The CAD runtime software architecture (component diagram)

- Component-level impact. In most cases, addressing a variant also requires us to do certain changes to components' internal implementation. Table 1 shows the impact of variants on CAD components (partial).

Variant Component	Validation	Dispatch Algorithm	CT-DISP	Encryption	…
Call Taker UI			X	X	
Dispatcher UI		X	X	X	
CADCreateTask	X		X	X	
CADDispatchTask		X		X	
Database Access					
…					

Table 1: The impact of variants on CAD components (partial)

The degree of the impact

- Narrow impact. Some variants may have limited impact on one or a few components. To accommodate these variants, only one or a few components need to be changed.

- Wide impact. Some variants may have wide impact on many components. For example, From Table 1, we can see that one component could be affected by many variants, and one variant may also affect many components. To accommodate these variants, we may have to trace the impact of variants across the components and make pervasive changes.

The predictability of the impact

- Anticipated. We can anticipate the impact of some variants and the actual changes required for incorporating these variants. To cater for specific requirements, we can select one of the anticipated implementations of the variants.

- Unexpected. For some variants, we understand that they have impact on the runtime architecture. However, we do not know the exact changes required to accommodate them as different systems may demand different implementations. Addressing unexpected variants is essential in poorly understood and evolving domains where requirements are always changing.

We shall be able to effectively handle the identified variants during product line development, so that our product line architecture is capable of incorporating both commonality and variability in a domain. For this, a good variability mechanism is needed.

4 XVCL – A Variability Mechanism

4.1 An overview of XVCL

XVCL (XML-based Variant Configuration Language) is a general-purpose mark-up language for configuring variants in programs and other types of documents.

Using XVCL, we design generic, adaptable components (or asset fragments) called x-frames. X-frames represent domain knowledge in the form of reusable assets, accommodating both domain defaults and variants. X-frame body is written in the base language, which could be a programming language such as Java, or a natural language such as English. XVCL commands allow the composition of the x-frames (via <adapt> command), and also make x-frames customizable by allowing one to select pre-defined options based on certain conditions (via <select> command), by marking breakpoints where additional changes can be inserted (via <break> and <insert> commands), and by providing variables as a parameterization mechanism (via <set> and <value-of> commands).

X-frames are organized into a layered hierarchy called an x-framework. X-frames at lower-levels are building blocks of higher-level x-frames. The hierarchal x-framework enables us to handle variants at all the granularity levels.

XVCL supports automated configuration of variants in product line assets. Given specifications recorded in a special x-frame (specification x-frame or SPC for short), the *XVCL processor* traverses an x-framework, performs composition and adaptation by executing XVCL commands embedded in x-frames, and constructs components of a specific system, a member of a product line.

For more details about XVCL and its commands, we refer the readers to our website at http://fxvcl.sourceforge.net.

4.2 A brief history of XVCL

The concept of "frame" was first introduced by Marvin Minsky in 1975. In his paper entitled "A Framework for Knowledge Representation", Minsky described a frame as follows:

Here is the essence of the frame theory: When one encounters a new situation (or makes a substantial change in one's view of a problem) one selects from memory a structure called a "frame". This is a remembered framework to be adapted to fit reality by changing details as necessary [12].

According to Minsky, a frame may be viewed as a static data structure used to represent well-understood stereotyped situations. We adjust to new situations by calling up past experiences captured as frames. We then specially revise the details of these past experiences to represent the individual differences for the new situation.

Frame has been widely used as a structured knowledge representation scheme. Many knowledge representation languages have been developed based on the frame concept. Some examples include KRL [2], FRL [14] and KEE [5].

In 1979, Paul Bassett applied the frame concept in the context of software engineering. Bassett's frames are analogous to the Minsky's frames. Bassett's frames are reusable pieces of code that can be adapted to meet specific needs. Bassett's use of frame hierarchies and default

BREAKs was inspired by the "frames" and "slots" concepts proposed by Minsky [1].

Bassett's frame technology has been extensively applied by Netron Inc. to manage variants and evolve multi-million-line, COBOL-based information systems. While designing a frame architecture is not trivial, subsequent complexity reductions and productivity gains are substantial. An independent analysis showed that frame technology has reduced large software project costs by over 84% and their times-to-market by 70%, when compared to industry norms [1]. These gains are due to the flexibility of the resulting architectures and their evolvability over time. The excellent record of frame technology in large-scale software applications was the main reason that led us to implement XVCL.

In 2000, our research team at the Software Engineering Laboratory of National University of Singapore developed XVCL (XML-based Variant Configuration Language), a new form of frame language that is free of COBOL heritage. In XVCL, Bassett's frame commands are designed as XML tags. Frames in XVCL are called x-frames, which are used to represent domain knowledge in the forms of product line assets (such as domain model or generic components). We also developed two versions of XVCL processors. In September 2002, we made XVCL available at an Open Source forum <u>fxvcl.sourceforge.net</u>, from which the latest XVCL language specification, processor and source code can be downloaded.

5 Applying XVCL to Construct CAD Product Line Architecture

Product line architectures are designed for software product lines instead of one-of-a-kind products. It is more general than the runtime architecture for a single system because it is engineered to be reusable, extendable, and configurable. For product line architecture, we are more concerned about the construction-time architecture – the generic architecture that can generate product line members having runtime architectures.

We develop the CAD product line architecture based on the CAD runtime software architecture (Figure 3). Using XVCL, we design generic, adaptable components as x-frames that incorporate both domain defaults and variants. The resulting x-frames are meta-components, from which concrete components are constructed during application engineering (the process of producing specific system using the reusable assets). We realize a product line architecture as a set of layered x-frames (x-framework). Being built of x-frames, the CAD product architecture has the ability to accommodate both commonality and variability in CAD domain. Specific CAD systems, members of CAD product line, can be constructed from the product line architecture.

5.1 Realizing CAD meta-components as x-frames

In this section, we show how we apply XVCL to handle variants in CAD components. Figure 4 shows the x-frame for constructing the CADCreateTask component in Figure 3, which contains a control class for creating a task. To accommodate variants, we instrument the CADCreateTask component with XVCL commands (highlighted as bold lines in Figure 4). The reader may find the variants described in this section in the feature diagram of Figure 2.

```
<x-frame name="x_CreateTask" language="java">

<set var="PACKAGE" value="BusinessLogic"/>
<break name="CREATETASK_NEW_PARAMETERS"/>

package <value-of expr="?@PACKAGE?"/>
      …
import java.util.*;
<break name="CREATETASK_NEW_IMPORTS"/>

public class CADCreateTask {
    private Caller    aCaller;
    private Task      aTask;

<break name="CREATETASK_NEW_ATTRIBUTES"/>

    public Caller GetCallerInfo() {
        …              // Code about capturing Caller's info
      return aCaller;
    }
     …
    public int SaveTask() {
        …              // Code about saving a task
<break name ="Validation"/>
      int nTaskID = aTask.Save();
      return nTaskID;
    }

<select option="CT-DISP">
   <option value="SEPARATED">
      <adapt x-frame = "InformDispatcher"/>
   </option>
</select>

<break name="CREATETASK_NEW_METHODS"/>
}
</x-frame>
```

Figure 4. The *x_CreateTask* x-frame

For the "Call Taker and Dispatcher roles" (CT-DISP) variant, we accommodate it by using a `<select>` command (`<select option="CT-DISP">`), which indicates the variation point where anticipated customization will occur. When the XVCL processor encounters a `<select>` command, it will select appropriate contents based on the value of "CT-DISP".

The `<break name="Validation">` command indicates the variation point brought up by the optional variant requirement Validation. Code that is related to

Validation variant may be `<insert>`ed into this variation point during x-frame adaptation.

XVCL variables, such as "PACKAGE", provide another means of handling variants. We may want to put components of different CAD systems into different Java packages. We define a meta-variable "PACKAGE" to represent the package name, with default value of "BusinessLogic". In Figure 4, we use an XVCL command `<value-of expr="?@PACKAGE?"/>` to indicate a place holder at which the actual value of PACKAGE is filled during x-frame adaptation.

Unexpected changes in requirements in the future may require us to add new attributes/methods to classes in CAD components. To accommodate these unexpected changes into the CADCreateTask component, we introduce two breakpoints `CREATETASK_NEW_ATTRIBUTES` and `CREATETASK_NEW_METHODS`. Specific code can be inserted into breakpoints through `<insert>` commands defined in the higher-level x-frames (e.g., in SPC). As we may also need more Java packages, we introduce a breakpoint `CREATETASK_NEW_IMPORTS` to accommodate the NEW_IMPORT variant. In addition, we introduce a breakpoint NEW_PARAMETERS to indicate a slot at which more XVCL parameters can be inserted.

5.2 Realizing CAD product line architecture as an x-framework

We realize a product line architecture as an x-framework - a hierarchy of x-frames. The x-framework for CAD product line is shown in Figure 5.

We design x-frames according to the principle of separation of concerns. For example, we design the Business Logic x-frames to encapsulate the business logic concerns and the UI x-frames to encapsulate the user interface concerns. In addition to the separation of concerns that we can achieve using conventional design and programming techniques, using XVCL we can achieve more "advanced" separation of concerns. Using XVCL, we can have arbitrary

decomposition of solution space, without necessarily following the class or function dimension. An x-frame could encapsulate a class, a function, or a code fragment at any granularity level that separates certain concerns. For example, the *x_CreateTask* x-frame separates the concern of creating a task (containing a class); the *Button* x-frame encapsulates the concern of creating Java buttons (containing code fragments for button definition, initialization and action). Separating different concerns into x-frames facilitates understanding, reuse and maintenance of the x-frames. To accommodate changes related to certain concerns, we need only examine and modify x-frames that encapsulate these concerns without having to do pervasive modifications to runtime components.

XVCL provides composition mechanisms (via `<adapt>`s and `<insert>`s) that compose separate x-frames together, at program construction time. Specifications for building a specific CAD system are defined in the root x-frame called specification x-frame (SPC). The SPC specifies the composition and adaptation of lower-level x-frames, and records the configuration of required variants. Given SPC, the XVCL processor traverses the x-framework, performs composition and adaptation, and constructs a custom CAD system meeting required variants. The constructed CAD systems shall have runtime software architectures such as the one in Figure 3 (other runtime software architectures are also possible).

During CAD x-framework development, we also identify common abstractions and design x-frames to represent them. For example, we find that CAD UI components, such as CallTaker UI and Dispatcher UI, have much in common: basically, they contain groups of elementary UI components and perform actions when these elementary components are activated. The differences are in the specific properties (such as number, name, layout) of the elementary components and in the way they respond to the events (i.e., specific implementations of the actions). The commonalities among UI components inspire us to design a generic *CADUI* x-frame for constructing these components.
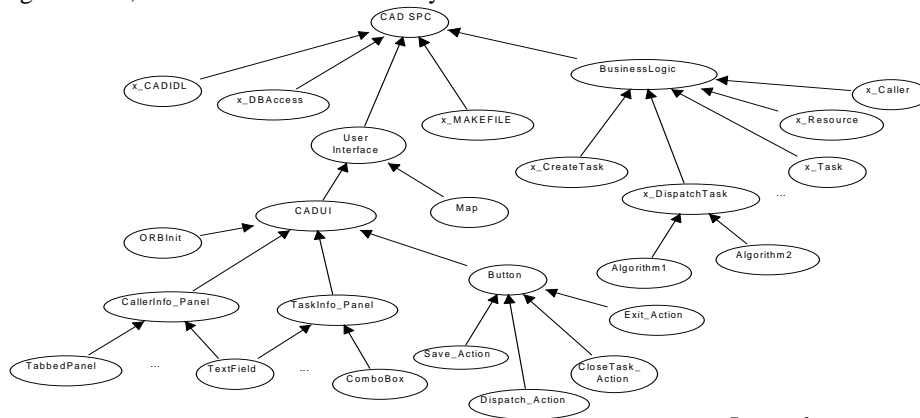

Figure 5. The CAD x-framework

5.3 Customizing CAD x-framework

To develop a specific system from product line architecture, we shall first examine the feature diagram (Figure 2) to select required variants. Having selected variants for a specific system, we record the variant configuration in an SPC, as well as possible additional implementations needed by certain variants. Figure 6 shows a partial SPC for customizing the *x_CreateTask* x-frame. In Figure 6, the value of CT-DISP variable is set to "SEPARATED", which means that the Call Taker and Dispatcher roles are separated. The <insert> command inserts specific code into the breakpoint Validation, to satisfy the requirement of "Basic Validation".

```
<x-frame name="CAD_SPC">
  <set var="Encryption" value="NO"/>
  <set var="CT-DISP" value="SEPARATED"/>
      …
  <adapt x-frame="x_CreateTask"
          outfile="CADCreateTask.java">
    <insert  break="Validation">
      if (!aTask.BasicValidation()) // Code about Basic Validation
        return -1;
      if (!aCaller.BasicValidation())
        return -1;
    </insert>
  </adapt>
      …
</x-frame>
```

Figure 6. A partial SPC for adapting *x_CreateTask* x-frame

The SPC enables automated customization of an x-framework. Given SPC in Figure 6, the XVCL processor adapts the *x_CreateTask* x-frame, customize it according to the instructions given as XVCL commands, and generates a specific CADCreateTask component that meets the specific requirements ("Basic Validation" and "Separated Call Taker & Dispatcher Roles") of a specific CAD system. Figure 7 illustrates the process of constructing custom CAD components from x-framework. Figure 8 shows the generated code of the CADCreateTask component. The constructed CAD system has the runtime architecture as shown in Figure 3.
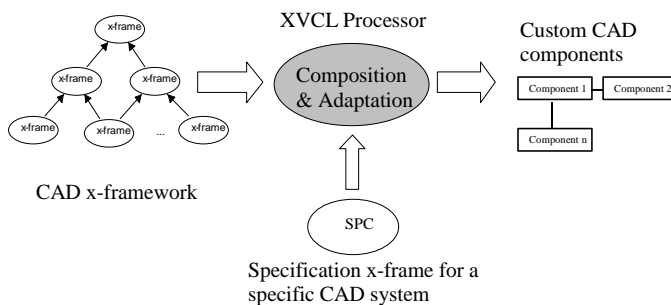

Figure 7. The construction of CAD components from the CAD x-framework

```
package BusinessLogic;
    …
import java.util.*;

public class CADCreateTask {
  private Caller    aCaller;
  private Task      aTask;

  public Caller GetCallerInfo() {
      …              // Code about capturing Caller's info
    return aCaller;
  }
      …
  public int SaveTask() {
      …              // Code about saving a task
    if (!aTask.BasicValidation())   // Code about Basic Validation
      return -1;
    if (!aCaller.BasicValidation())
      return -1;

    int nTaskID = aTask.Save();
    return nTaskID;
  }
  public int InformDispatcher (Task aTask) {
      // Code about informing dispatcher of a newly created task
      ...
    return 0;
  }
}
```

Figure 8. **The generated *CADCreateTask* component**

The SPC such as the one given in Figure 6 only contains configuration knowledge for one specific system. To develop other CAD systems that meet different requirements, we design different SPCs. For example, if in Figure 6 we change the value of CT-DISP to MERGED, the constructed CAD system will have a runtime architecture with CallTaker UI and Dispatcher UI components merged.

Currently, we write SPCs manually. However, it is possible to develop a GUI-based "wizard" tool to help us generate SPCs from feature diagram.

6   Discussions

Rather than addressing variants at construction-time, we could implement variants directly into the component code, and provide a runtime mechanism (such as runtime variables, inheritance and dynamic binding) to select required variants. The choice between construction-time and runtime techniques for handling variants is an important decision in supporting product lines, having profound impact on system properties such as complexity, performance, ease of maintenance, etc.

Currently, component technology largely builds on existing Object-Oriented approaches. Very often, components are implemented as classes.   Existing Object-Oriented approaches focus on developing single systems rather than product lines. Many OO programming languages, such as Java, do not support generics directly. To accommodate variants, one may use OO reuse mechanisms such as

inheritance and polymorphism. For example, one may design a CADCretateTask class hierarchy as shown in Figure 9.

Though this solution only accommodates two variants, we need to define seven classes. If the number of variants increases, the class hierarchy will expand quickly, causing maintenance and scaling problems, and the overhead of method overriding.
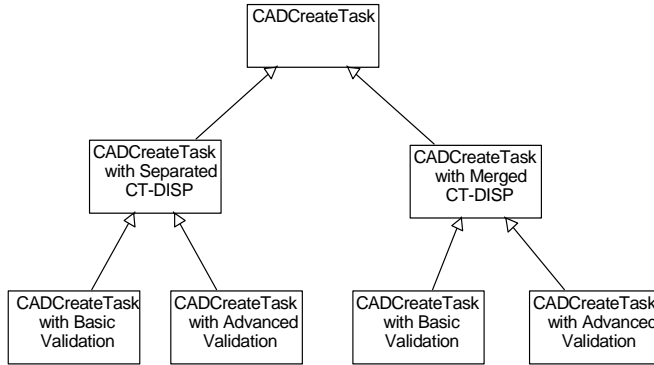

Figure 9. Implementation of *CADCreateTask* using inheritance

XVCL technique complements OO or component-based methods by providing generic solutions at program construction time. From x-frames, concrete components containing only necessary code are generated. Thus, we avoid uncontrolled growth of concrete components in size and number, and ease the maintenance and evolution of components.

XVCL shows its strength when addressing variants that have global impact on a system. The impact of such a variant (such as the CT-DISP variant) cannot be nicely localized into a single component identified through conventional modularization approaches such as OO analysis and design. We accommodate these variants by instrumenting components with XVCL commands, and by designing lower-level x-frames that separate the impact of variants. Although these variants have impact on many x-frames, the x-frame processor automates the composition and adaptation process so that the transition from the x-frames to concrete components is transparent to programmers, releasing programmers from handling variants manually.

A limitation of XVCL is that, being generic, x-frames could be difficult to understand. The verbose XML syntax also has negative impact on understanding x-frames. These problems can be mitigated by documenting the design rationale, by carefully designing x-frameworks according to XVCL design rules, and by re-factoring the design as an x-framework evolves. We are also developing XVCL Workbench to facilitate x-framework development. The XVCL Workbench includes tools such as a smart x-frame editor that hides XML syntax and displays graphical views of x-frames, and a static analysis tool that helps us understand an x-framework.

7   Related Work

The concept of software product lines is derived from what Parnas referred to as program families [13]. A product line is a collection of systems sharing a managed set of features constructed from a common set of core software assets [3]. These assets include a product line architecture and a set of components that populate it.

The simplest way to handle variants in a product line could be "copy-and-modify". When programmers are called upon to write a new program, naturally they look for a similar one and copy it. They then modify the code to implement variants. Although a step in the right direction, the "copy-and-modify" method is rather ad hoc and difficult to scale up.

Object-Oriented frameworks [11] use information hiding along with inheritance, dynamic binding and design patterns [6] to handle variants in product lines. An application generator approach [15] is a powerful and cost-effective solution to product lines in well-understood and stable domains. Generators transform problem specifications written in domain-specific languages into concrete programs. Template meta-programming [4] also addresses certain types of variants at the program construction time. Templates are tightly coupled with specific programming languages such as C++.

Recent work focuses on advanced separation of concerns. Concerns in multiple dimensions may spread throughout the whole program and cannot be nicely confined to a small number of components. A number of approaches have been proposed to address crosscutting concerns and concern compositions. In aspect-oriented programming [8], each computational aspect is programmed separately and rules are defined for waving aspects with the base code (typically object-oriented classes).

XVCL is a frame-based technique. Although x-frames in our examples contain Java code, XVCL is programming language independent. In fact, it is a uniform mechanism that can be used to handle variants in a variety of inter-related product line assets such as domain models, components, documentation, etc. Unlike OO and template techniques, XVCL can handle variants at any granularity level. Unlike AoP, in XVCL we explicitly mark the points affected by variants and specify required adaptations. Unlike other methods, XVCL can accommodate unexpected changes (via <break>s). Additional code that caters for unexpected changes can be <insert>ed into the breakpoints (slots) defined in an x-frame during customization.

8   Conclusion

In this paper, we described an XVCL-based approach to software product line development. XVCL is a variability mechanism we developed for handling variants in a product line. XVCL is based on the concepts of frame technology [1]. In XVCL, x-frames represent domain knowledge in the

forms of product line assets. Using XVCL, we realize generic, adaptable components as x-frames and product line architecture as a set of layered x-frames (x-framework). Specific systems, members of a product line, can be constructed by reusing the x-frames. We illustrated our approach using examples from our CAD product line project.

Note that there are also economical, managerial and organizational issues in the development of a product line, such as market analysis, strategic planning, organization structure, etc. These issues are all very important for product line approach to succeed in industrial practices. However, in this paper, we only discuss the technical aspects of the product line development, focusing on handling variants in a product line.

In the future, we plan to extend the scope of the experimentation, apply our approach to larger-scale industrial projects in a variety of application domains. We shall also continue exploring XVCL-based solution to advanced separation of concerns.

Acknowledgements

References

[1] Bassett, P., *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997.

[2] Bobrow, D. and Winograd, T., An Overview of KRL: a Knowledge Representation Language, *Cognitive Science 1*, 1 (1977) 3-46, 1977.

[3] Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Reading, Addison-Wesley, 2001.

[4] Czarnecki, K. and Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[5] Fikes, R. and Kehler, T., The Role of Frame-Based Representations in Reasoning, *Communications of ACM*, Vol.28, 9, 1985, pp.904-920.

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] Kang, K., Cohen, S.G., Hess, J.A., Novak, W.E. and Peterson, A.S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, SEI, CMU, Nov. 1990.

[8] Kiczales, G. et al., Aspect-Oriented Programming, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, 1997.

[9] Jacobson, I., M. Griss and P. Jonsson, *Software Reuse Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[10] Jarzabek, S. and Zhang, H., XML-based Method and Tool for Handling Variant Requirements in Domain Models, *Proc. Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada, August 2001.

[11] Johnson, R. and Foote, B., Designing Reusable Classes, *Journal of Component-Oriented Programming*, June 1988, Vol.1, No.2, 1988, pp. 22-35.

[12] Minsky, M., A framework for representing knowledge, *The Psychology of Computer Vision*, P. Winston (ed.), New York: McGraw-Hill, 1975.

[13] Parnas, D., On the Design and Development of Program Families, *IEEE Trans. on Software Eng.*, vol. 2, 16, 1976, pp. 1-9.

[14] Roberts, B. and Goldstein I., The FRL Primer, MIT AI Laboratory Memo 408, July 1977.

[15] Smaragdakis, Y. and Batory, D., Application Generators, *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*, J. Webster (ed.), John Wiley and Sons, 2000.

[16] Soe, M.S., Zhang, H. and Jarzabek, S., XVCL: A Tutorial, *Proc. of 14th Int. Conf. on Software Engineering and Knowledge Engineering*, SEKE'02, Italy, ACM Press, July 2002, pp. 341-349.