Guidelines for designing an x-framework for a product line

1. The overall process

The following are the main activities in the development of an x-framework for a product line:

1. Domain analysis: Analyze and model common and variant requirements for the product line.

2. Scope the product line: decide which common and variant requirements we will implement within the x-framework

3. Design a runtime architecture in terms of components and connectors for software systems, product line members.

4. Design a default system, a product line member, and develop a prototype for a (subset of the) default system. The default system should contain common and some of the variant requirements that occur in most of the product line members. The default system is presented as a typical system in the domain.

5. Develop an x-framework for the product line incrementally: An x-framework is an XVCL implementation of the concept of product line architecture. In the x-framework, we shall have "generic component/connectors" from which components/connectors of the specific system, product line member, are generated. The specific components/connectors are organized according to the runtime architecture developed in point 3. The development of an x-framework is an incremental process. In the first iteration, we frame the default system to cater for some variants. In the following iterations, we extend the x-framework developed so far by implementing more functionality into it (that is extending the default system) and/or by addressing some new variants.

2. Designing a software runtime architecture

A software runtime architecture is a blueprint for runtime structure of product line members. Runtime architecture is described in terms of components, connectors (that is component interactions) and constraints imposed on connectors. Most often, product line members will share the same runtime architecture. However, occasionally, certain variants may affect some details of the runtime architecture. For example, a component may or may not be included in the product line member X, depending on whether or not X should satisfy a specific variant. Yet other variants may affect component interfaces.

2.1. Architectural styles

Shaw and Garlan identified a number of styles for runtime architecture (Shaw and Garlan, 1996). An architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined. Some common architectural styles are:

- Pipes and filters
- Main Program/Subroutine with Shared Data

- Abstract Data Types (Object-Oriented)
- Implicit Invocation
- Client/Server
- Three-tiered
- Distributed component architecture

In the design of runtime software architecture, we can adopt one or more architectural styles. Different architectural style have different characteristics (in terms of performance, simplicity, reusability, modifiability, etc.). For example, as Parnas points out from the KWIC experimentation (Parnas, 1972), the shared-data style is particularly weak in its support for changes in data representations. On the other hand it can achieve relatively good performance. The abstract data type (Object-Oriented) style is good for accommodating changes in data representation, but may have negative impact on system performance. Thus we often make tradeoffs in selecting architectural style.

2.2. The impact of variants on runtime software architecture

After developing the runtime architecture of the default system, we analyze the impact of variants on the runtime architecture, focusing on the following aspects:

The level of impact

- Architecture-level impact. To address certain variants, we may need to include new components into the system, remove existing component 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.
- Component-level impact. In most cases, addressing a variant also requires us to do certain changes to components' internal implementation.

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. To accommodate these variants, we shall be able 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 variants.
- Unexpected. For some variants, we understand that they have impact on the runtime architecture. However, we do not know the exact changes required in the future as different systems may demand different implementations. Addressing unexpected

variants is essential in poorly understood and evolving domains where requirements are always changing.

3. Developing a default system

Before developing an x-framework for a product line, we develop a default system. The default system represents a typical product line member. The default system covers all the common requirements and also implements typical variant requirements that most of product line members are likely to have.

During the design of the default system, we keep in mind that we are not developing a one-of-a-kind system; instead we are laying out the foundation for developing many systems that a product line will comprise. In the next step, we shall turn a default system into the first-cut x-framework for a product line. Therefore, we should design the default system to make the transition to the x-framework as easy as possible. We apply usual software engineering principles, such as modularization, information hiding and separation of concerns, in the design of a default system.

The information hiding and separation of concerns principles (Parnas, 1972) allow the design decisions and functions to be implemented within separate components and hidden from outside. A component offers its users abstract interfaces by which they can use its services. The abstract interface separates the concern of which services the component offers from the concern of how those services are implemented within the component. Information hiding makes it possible to change the internal implementation of one component without knowledge of the implementation of other components and without affecting the behavior of other components. Applying information hiding and separation of concerns principles facilitates the accommodation of changes into components.

Typical classes identified in OO analysis encapsulate data representation and facilitate changes of data structures. Overall system control and workflows tend to be implicit in object interactions. Then, changes that affect system control and workflows are difficult to implement as they may involve many interacting classes. In such cases, it is worthwhile to introduce control classes/components (or workflow classes/components) so that control/workflow elements are localized. By introducing such control components, the interactions among classes are greatly simplified and localized, therefore, amenable to future changes.

4. Developing an x-framework

The x-frames are meta-components – that is components that generate components. During x-framework processing, the XVCL processor executes XVCL commands embedded in the x-frames, and constructs concrete components for a specific system, a member of the product line.

X-frames should be *generic* enough to be reusable across members of the product line. X-frames should also be *flexible* in respect to both anticipated variants and new, unexpected requirements. Flexible x-frames (and flexible x-framework) can be (1) easily customized to accommodate anticipated variants (that is variants identified in domain analysis) into a

specific system we wish to build, and (2) easily extended to address new, unexpected requirements that we need in a specific system we wish to build.

An x-framework implements common product line requirements and typical variants. The x-frame body is written in the base language, which could be a base language, usually a programming language such as Java or C++. XVCL commands mark the variation points at which pre-defined variant behaviour can be selected, extra behaviour can be added or default behaviour can be modified.

4.1. Framing components

We start x-framework development by framing components defined by the runtime architecture of the default system. Framing turns runtime components into generic components. The design of x-frames is driven by variants identified in domain analysis. There are many types of variants such as functional variants, variant design decisions and implementation variants. We address all these variants during framing.

The development of x-frames is an incremental process. We incorporate variants into the default system incrementally until all variants have been addressed. During framing, we:

- introduce XVCL variables to represent generic parameters
- use <select> command to select one from many alternatives
- use <break> and <insert> commands to address variants whose implementation are uncertain at the time of designing x-frames
- use <while> command to generate code that have similar patterns
- split a large x-frames into several smaller x-frames, and compose them by using <adapt> commands
- modify the design and implementation of the defaults (e.g., change data structure, algorithm, component interface, etc).

As the number of variants increases, x-frames become more and more useful and reusable.

4.2. A sample x-frame for Java components

An x-frame of Figure 1 shows an x-frame for a typical Java component. The x-frame contains a Java class implementing a component. XVCL commands mark variation points. In this simple x-frame, we address only five variants, namely:

- PACKAGE. The component may be placed in different packages.
- CLASS_NAME. The class implementing the component may have different names in different applications.
- NEW_METHODS. The component may require new class attributes to cater for unexpected changes in requirements.
- NEW_ATTRIBUTES. The component may require new class methods to cater for unexpected changes in requirements.

- NEW_IMPORTS. The component may require additional Java import packages that are needed by newly added methods/attributes.

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

<set var="PACKAGE", value="package"/>        //  XVCL variable definitions here
<set var="CLASS_NAME", value="aClass"/>
        …
<break name="FRAME_NEW_PARAMETERS"/>

package <value-of expr="?@PACKAGE?"/>  // Java package

import java.util.*;                        //  Java imports here
        …
<break name="FRAME_NEW_IMPORTS"/>

public class <value-of expr="?@CLASS_NAME?"/> {
    private int      nAttribute1;        // class attributes
    private String      sAttribute2;
      …
<break name="FRAME_NEW_ATTRIBUTES"/>

     public <value-of expr="?@CLASS_NAME?"/> () {  // constructor method
        …
     }
     public void  method1 {            // class method
        …
<break name ="FRAME_BREAKPOINT1"/> // possible breakpoints in the method

        return;
     }

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

Figure 1. An example x-frame for Java components

To accommodate the PACKAGE variant, we define an XVCL variable PACKAGE to represent the package name, with default value of “package”. The default value can be specialized during program customization. In Figure 1, 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 processing. Similarly, we define an XVCL variable CLASS_NAME to accommodate the CLASS_NAME variant.

To accommodate the NEW_ATTRIBUTES and NEW_ATTRIBUTES variants, we introduce two breakpoints FRAME_NEW_ATTRIBUTES and FRAME_NEW_METHODS, to indicate slots at which possible new attributes or methods can be inserted. Similarly, we introduce a breakpoint FRAME_NEW_IMPORTS to indicate the slot at which more import packages can be inserted. In addition, we introduce a breakpoint FRAME_NEW_PARAMETERS to indicate a slot at which more possible XVCL parameters can be inserted. Slots indicated by the breakpoints can be filled in with specific code during x-frame processing.

5. Realizing product line architecture as an x-framework

An x-framework is an implementation of the product line architecture concept. From an x-framework, product line members can be constructed. The runtime architecture is implicitly defined in the x-framework and all the product line members constructed from the x-framework comply to that runtime architecture.

Each component of the runtime architecture (or sometimes a group of components), usually has a corresponding x-frame from which this component (or a group of components) is constructed.

X-framework has a layered hierarchy. Higher-level x-frames adapt the lower-level x-frames during x-frame processing. X-frames in the x-framework differ in their level of context sensitivity. X-frames near the bottom of the hierarchy are relatively context free. Examples of these low-level x-frames are I/O access x-frames, sorting algorithm x-frames, elementary UI components, etc. X-frames near the top of the hierarchy adapting their child x-frames, they are relatively context sensitive.

To develop x-framework, we apply the following principles:

5.1. Separation of concerns

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, the UI x-frames to encapsulate the user interface concerns, the error handling x-frames to encapsulate the error handling concerns. With XVCL, we can achieve "advanced" separation of concerns, that extends separation of concerns that we can achieve using conventional design and programming techniques.

Conventional design and programming techniques allow us to decompose a program into modules along a single, "dominant" dimension of concern (Tarr et al., 1999). For example, in object-oriented methods the dominant dimension is class (object) dimension, and a program is decomposed into a set of classes. With procedural programming languages, a program is decomposed into a set of procedures (or functions). Code pertaining to other concerns (e.g., synchronization) are not be nicely encapsulated into separate modules, instead they scatter across many modules and tangle with one another, making reuse and maintenance difficult.

XVCL allows us to separate such concerns. Using XVCL, we can have arbitrary decomposition of a program along a number of dimensions, without necessarily following the class or function dimension. An x-frame may encapsulate a class, a function, or a code fragment that is related to any concern and at any granularity level. XVCL provides a mechanisms (via <adapt>s and <insert>s) to compose x-frames containing code related to different concerns (Figure 2).

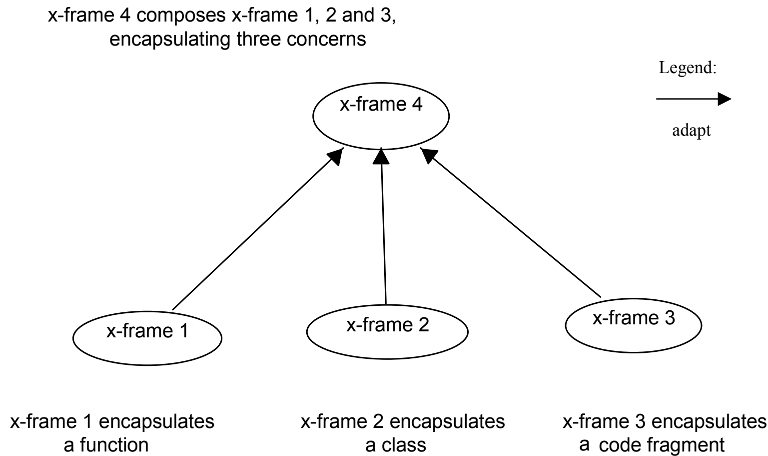


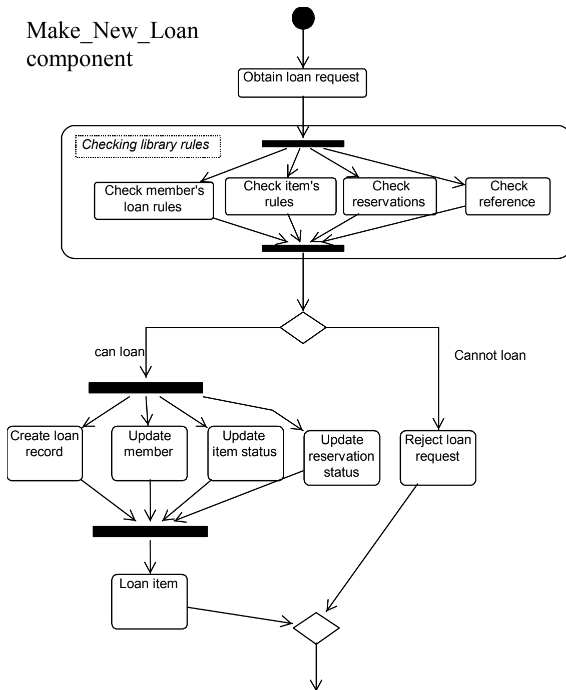
Figure 2. The composition of x-frames that separate concerns

In Figure 2, x-frame 1 encapsulates a function, x-frame 2 encapsulates a class, x-frame 3 encapsulates an arbitrary code fragment. These x-frames all related to different concerns. They can be composed together, forming x-frame 4 from which specific runtime components can be constructed. Although the resulting runtime components may contain scattered and tangled code, we achieve separation of concerns at program construction time (at the x-frame level).

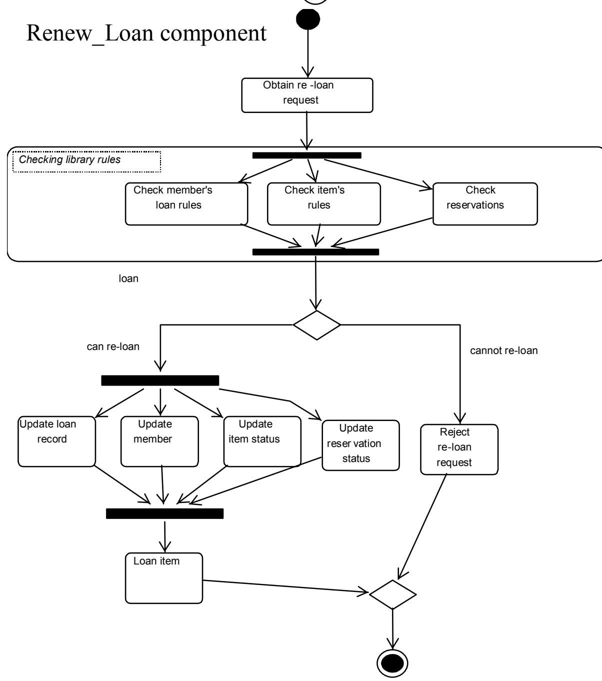
Separating different concerns into x-frames facilitates reuse and maintenance of the x-frames. To accommodate changes related to certain concerns, we need only examine and modify x-frames that encapsulate these concerns without having to do pervasive modifications to runtime components.

5.2. Designing common abstractions as x-frames

If we find that some components have much in common, we design x-frames to capture the commonalities. Then, we instrument the x-frames with XVCL commands to handle component parts that are different among components. We build x-frames from which similar components can be constructed.



Make_New_Loan component
Obtain loan request
Checking library rules
Check member's loan rules
Check item's rules
Check reservations
Check reference
can loan
Cannot loan
Create loan record
Update member
Update item status
Update reservation status
Reject loan request
Loan item



Renew_Loan component
Obtain re -loan request
Checking library rules
Check member's loan rules
Check item's rules
Check reservations
loan
can re-loan
cannot re-loan
Update loan record
Update member
Update item status
Update reser vation status
Reject re-loan request
Loan item

Figure 3 shows UML activity diagrams of two control components in a typical library system for universities, namely Make_New_Loan (for making a new loan from library) and Renew_Loan (for renewing a loan). We can see that these two components have much in common in terms of control flow pattern and major activities. These commonalities inspire us to design an x-frame to represent the common abstraction.

By capturing commonalities and variations among components, we reduce the redundant code within x-framework, making the x-framework more compact and easy to maintain.

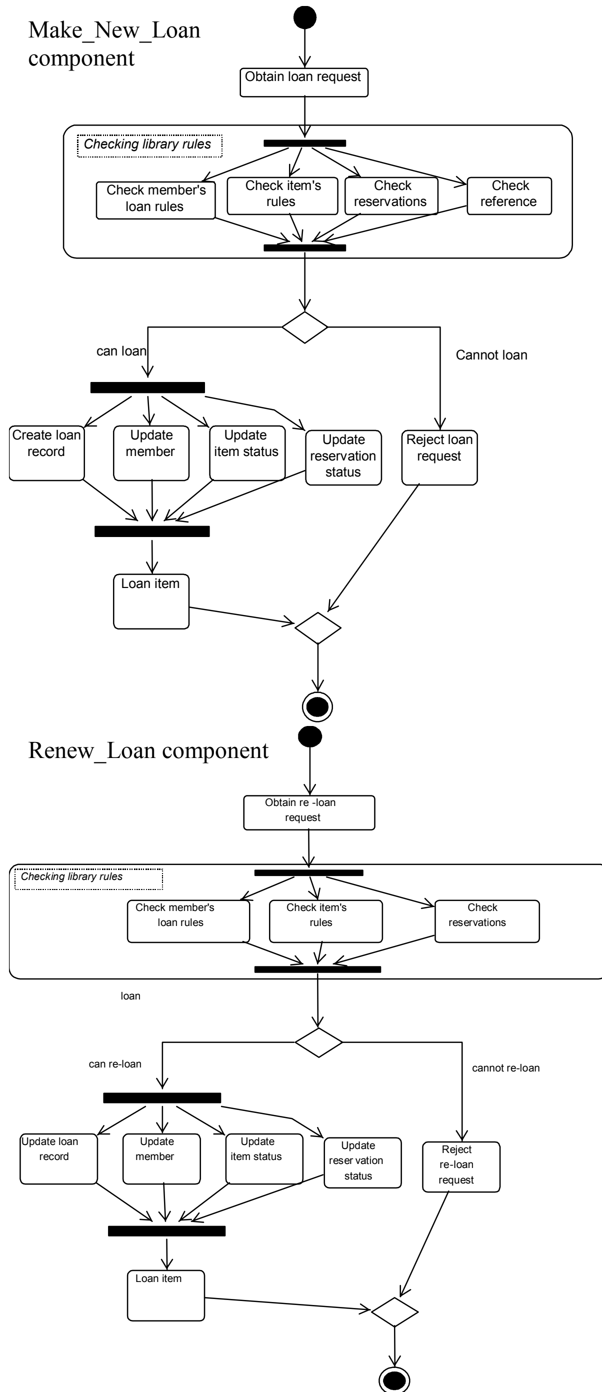


Figure 3. Two control flow components in a library system

5.3. Handling variants that affect the runtime architecture

Product line members may differ in details of runtime architecture. Some variants affect not only component implementation but also the runtime architecture – that is components that form the system and/or component interfaces. In x-framework design, we must also address such variants. An x-framework of Figure 4, facilitates construction of systems that differ in runtime software architectures. For example, runtime architecture 1 includes component5, while architecture 2 does not include component5..

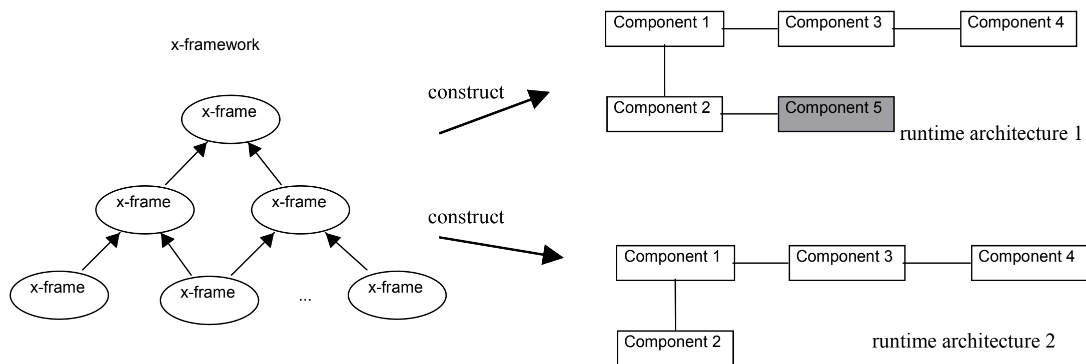


Figure 4. Constructing different runtime architectures from x-framework

Architectural variants can be also accommodated into an x-framework by using XVCL. For example, the inclusion/exclusion of the components can be achieved by the <select> commands, as illustrated by the x-frame fragment in Figure 5. An XVCL variable VARIANT decides whether to construct a specific component.

```
…
<select option="VARIANT">
  <option  value="ARCH1">
    <adapt x-frame="x-frameN.xvcl" outfile="Component5"> // constructing component 5
        …
    </adapt>
  </option>
  <option  value="ARCH2">
        …                    // not constructing component 5
  </option>
</select>
…
```

Figure 5. The selection of x-frames for accommodating architectural variants

6. Customizing an x-framework

We customize an x-framework to construct specific systems, members of a product line. First, we select the required variants that the customer wants. Having selected variants, we design an SPC for customizing the x-framework. Given an SPC, the XVCL processor traverses the x-framework, performs composition and adaptation according to the instructions given as XVCL commands, and constructs a custom system meeting required

variants. Figure 6 illustrates the process of generating product line members from x-framework.

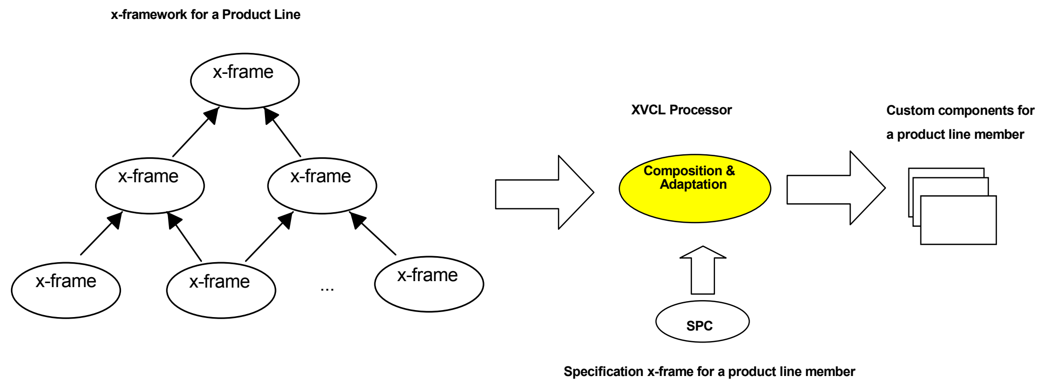


Figure 6. The generation of product line members from x-framework

It is possible that not all x-frames in an x-framework are used for constructing a specific system. We may only need a set of x-frames to do so. It is also possible that we need to assembly new x-frames into the x-framework for constructing specific systems.

7. Criteria for a good x-framework

An x-framework is designed to make programming easier and more productive. A good x-framework will facilitate customization and evolution improve software quality by:

- reducing the amount of code which must be written and maintained
- eliminating redundancies
- minimizing the impact of changes.

8. X-frame documentation

X-frames are more difficult to understand than specific programs. The verbose XML syntax also has negative impact on understanding x-frames. A large number of variants would increase the complexity of the x-frames. Special cares (such as using comments, documenting design rationales and following x-frame writing conventions) must be taken to increase the understandability of x-frames.

8.1. Naming conventions

The names of XVCL variables and breakpoints should be standardized, e.g.:

"FRAME_VAR" and "FRAME_BREAKS".

We use upper-case to distinguish XVCL commands from the base code. For example, in an x-frame called NOTEPAD, the variable BGCOLOR and break NEWMETHODS could be defined as:

"NOTEPAD_BGCOLOR" and "NOTEPAD_NEW_METHODS"

The reason for using x-frame's name as prefix is to avoid potential name conflict in a large x-framework. Default values of the XVCL variable

XVCL variables can be set in higher-level x-frames and referred in low-level x-frames. In lower-level x-frames, we usually set default values of the XVCL variables that will be used in case there are no overriding set commands at higher-level x-frames.

8.2. Documenting anticipated changes

During x-frame writing, we document the anticipated changes. For example, in the <break> section, we give the default implementation, and comment on other possible implementations. This info will help x-frame users understand the intention of the breakpoints and see possible ways to customize the default implementation given at breakpoints.

Another example is related to a single-valued variable. We only assign one value to a single-valued variable. However, we should also comment on other possible values.

```
<set var="NOTEPAD_VAR1" value="VAL1"/>   <!-- Could be VAL1, VAL2 or VAL3-->
```

Similarly, for the <select> command, we should address all possible options of the variables, including the option when the variable is not defined (option-undefined).

9. The maintenance of the x-frames

X-frame developers should be proficient programmers. Their task is to create generic programs. Designing x-frames requires more effort than just writing a single program.

Once developed and tested, x-frames are marked as read-only and stored in the x-frame repository. Application engineers can only reuse the x-frames. Direct modification of x-frames by application engineers is not allowed. The x-frame repository should be able to categorize x-frames, and provide service for browsing and retrieving x-frames.

If an x-frame (or a group of x-frames) is not fully suitable for a given project, application engineers may design "wrapper x-frames" to cater for project-specific needs. The wrapper x-frames adapt the original x-frames rather than change them directly. The complexity of original x-frames is hidden in the wrapper frames.

X-Frames evolve over time. Frame engineers need to upgrade x-frames to cater for new requirements and to keep up with new technologies. When new versions of x-frames are developed, they must be tested and reviewed to pass proper certification, before they can be checked back to the x-frame repository.

References

Booch G., Rumbaugh, J. and Jacobson, I., 1999. *The Unified Modeling Language User Guide*, Addison Wesley, 1998.

Kruchten, P., 1995. The 4+1 View Model of Architecture, *IEEE Software* 12(6), pp. 42-50.

Jacobson, I., Booch, G., and Rumbaugh, J., 1999. *The Unified Software Development Process*, Addison-Wesley.

Jarzabek, S. and Zhang, H., 2001a. Enhancing Component Reuse with Control Flow Abstraction Analysis, *Proc. of 13th International Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina, 2001, Knowledge System Institute, pp. 171-178.

Parnas, D.L., 1972. On the Criteria To Be Used in Decomposing Software into Modules, *Communications of the ACM,* Vol. 15, No. 12, December, 1972, pp.1053-1058.

Shaw, M. and Garlan, D., 1996. *Software architecture: perspectives on an emerging discipline*, Prentice-Hall, 1996.

Tarr, P., Ossher, H., Harrison, W. and Sutton, S., 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns, *Proc. International Conference on Software Engineering*, ICSE'99, Los Angeles, 1999, pp. 107-119.