Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 53 (2004) 305–331
Science of Computer Programming
www.elsevier.com/locate/scico

Modeling variability in software product lines with the variation point model

Diana L. Webber[a,*], Hassan Gomaa[b]

[a] *Booz Allen Hamilton, 8283 Greensboro Drive, McLean, VA 22102, USA*
[b] *Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA*

Received 12 January 2003; received in revised form 10 April 2003; accepted 24 April 2003
Available online 30 July 2004

Abstract

A major challenge for software reuse is developing components that can be reused in several applications. This paper describes a systematic method for providing components that can be extended through variation points, as initially specified in the software requirements. Allowing the reuser or application engineer to extend components at pre-specified variation points creates a more flexible set of components. The existing variation point methods do not provide enough design detail for the reuser. This paper introduces a method called the Variation Point Model (VPM), which models variation points at the design level, beginning with the common requirements. The product line approach provides a systematic approach for software reuse. A challenge with the product line approach is to model the variability between the core assets and the applications. This paper describes the VPM and how it is used for modeling four different approaches to variability, modeling variability using parameterization, modeling variability using information hiding, modeling variability using inheritance, and modeling variability using variation points. VPM allows a reuser or application engineer to extend components at pre-specified variation points. For this to be possible, a variation point must be modeled such that the reuser has enough knowledge to build a variant.
© 2004 Elsevier B.V. All rights reserved.

1. Introduction

Modeling variation points is accomplished with two major methods in the context of software product lines. The first method is to model the variation points [17] so that the developer of the core assets can manage the variability of the core assets or core components.

* Corresponding author. Tel.: +1-703-902-4062.
E-mail addresses: webber_diana@bah.com (D.L. Webber), hgomaa@gmu.edu (H. Gomaa).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2003.04.004

The second method models the variation points so that the reuser of the core assets can build unique variants from the variation points. Jacobson defines a variation point as follows: "A variation point identifies one or more locations at which the variation will occur" [17]. The Variation Point Model (VPM) adds the mechanism of variability to this definition. The Software Engineering Institute defines a core asset as follows: "Core assets are those assets that form the basis for the software product line. Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirement statements, documentation and specifications, performance models, schedules, budgets, test plans test cases, work plans, and process descriptions" [8].

In the first method, the design of the variation points is only understood and used by the core asset developer. The core asset developer builds and maintains the variants used in the applications. The reuser chooses between a set of variants for each variation point and creates an application. When the reuser needs a variant that is not available in the core assets, the core asset developer adds that variant to the core assets and makes it available to all reusers. Modeling the variation points helps the core assets developer to manage and build new variants.

The objective of the second method is to make the core assets more reusable by allowing the reuser to build variants from variation points. This approach gives the reuser the flexibility of tailoring the core assets to his/her needs. However, a variation point must be modeled such that the reuser has enough knowledge to build a variant. This approach keeps the core asset developer from the task of maintaining every possible variant. This is very useful when a variant is unique to a reuser and not needed by other reusers. In either approach, variation points must be adequately modeled so that they can be maintained and managed. Core assets that include variation points are referred to as the common core in this paper.

There are two main parts of a product line: the core assets and the application. The core assets are the holdings that are reused on several similar applications or target systems. Each target system varies its reuse of the core assets. The variability in a product line is the difference in the distinct applications. Section 2 of this paper provides a summary of variability approaches.

There are four different approaches to model variability: modeling variability using parameterization, modeling variability using information hiding, modeling variability using inheritance, and modeling variability using variation points. Each approach depends on the amount of flexibility needed in the product line. Section 3 of this paper describes and compares the four different approaches to modeling variability. In order to illustrate the differences between the four variability approaches, examples of component variability are taken from the Banking System Case Study (BSCS). The system was designed using the COMET/UML method as described in [13] and then modified to support the variability mechanisms. Section 4 gives a detailed description of the Variation Point Model (VPM), which builds on the work contained in [26].

2. Survey of variability methods

The following survey is a summary of the most relevant pieces of related work. Jacobson described six variability mechanisms for providing variation points in a software

product line: inheritance, uses, extensions, parameterization, configuration, and generation [17]. Bosch describes five variability mechanisms: inheritance, extensions, configuration, template instantiation, and generation [6].

Alcatel/Thomson-CSF Common Research Laboratory (LCAT) are leading an ongoing PRAISE project that focuses on the design and representation of a product-line architecture with UML [11]. They use a UML package to represent a variation point or hot spot with the stereotype <<hot spot>>. They also tag any collaboration with a variant with "variation point". Hot Spots are a method to model variation points [21]. Hot spots show a reuser where the variation point is located and when a reuser must extend a component's functionality.

In addition to PRAISE, LCAT also has a project entitled Software Product Line Integrated Technology (SPLIT) [9]. SPLIT is an experimental method that helps LCAT define and build product lines. SPLIT considers variation points to have attributes and therefore uses the UML classifier, class, to depict a variation point. The attributes SPLIT gives a variation point are attributes used to help provide information needed to choose a variant. A subsequent paper on SPLIT discusses the three variability mechanisms: insert, extend, and parameterize [10].

Looking at some product-line case studies clearly shows the need for VPM. CelciusTech uses parameterization as a variability mechanism [7]. They are in need of a way to manage the parameters and understand the dependencies between them. NASA Goddard Space Flight Center Flight Software Branch (FSB) has come to the same conclusion and created a technique to represent variability [19,20]. FSB created a new symbol to identify the variable elements. However, the reuser must define, for each specialized class, if a tagged operation is kept or not. Software Technology for Adaptable Reliable Systems (STARS) [24,25] and Boeing's avionics software [4,23] both use the same logic as FODA and EDLC by providing the reuser with a list of variants to choose from rather than the ability to create a unique variant of their own.

Product Line Software Engineering (PuLSE) [3], KobrA [1,2], Wheels [9], Bosch's Product-Line Design Process [6], and Family-Oriented Abstraction, Specification, and Translation (FAST) [28] are all processes specifically for building product lines. Although Bosch and FAST discuss variability mechanisms, they do not include a method to model variation points. A more detailed survey is given in reference [27].

In summary, the related work discussed in this section lays the foundation for variation point modeling. VPM builds on the concepts presented in the different methods and processes described in this section. VPM builds on the idea of commonalities and variabilities from domain analysis to concentrate on the point of variation. VPM extends the definition of a variation point to include its variability mechanism in addition to its location. VPM builds on Jacobson's and Bosch's variability mechanisms by creating variation point types that are directly related to their modeling technique. At this time, four variation point types have been modeled: parameterization, information hiding, inheritance, and callback. The survey showed that parameterization followed by inheritance are the two most widely used types today. Callback was modeled due to its appealing flexibility and the availability of a product line that used the callback type. VPM builds on the related work to create a method to model variation points so a reuser can build a unique variant.

3. Four approaches for modeling variability

There are four different approaches to model variability: modeling variability using parameterization, modeling variability using information hiding, modeling variability using inheritance, and modeling variability using variation points. Each approach depends on the amount of flexibility needed in the product line.

3.1. Modeling variability using parameterization

3.1.1. Approach

Modeling variability using parameterization is where the variation is to the value of the parameters defined in the core assets. Parameterization is a variability mechanism that allows a reuser to change the values of the attributes in a core asset component. This can be done by providing the capability through the component's interface to initialize or change the value of parameterized attributes.

Several domain analysis methods use parameterization. The Feature Oriented Domain Analysis (FODA) method developed at the Software Engineering Institute (SEI) uses features to characterize the domain [18]. In FODA, features may be functional features, non-functional features or parameters. In the Domain-Specific Software Architecture program, generic components were parameterized to simplify customization for particular applications [16]. The Evolutionary Domain Life Cycle (EDLC) Model for software product lines is used to develop domain models, domain architectures, and reusable component types [15]. During target system configuration, the component types to be included in a given member of the product line are selected from the domain reuse library. Given the target system configuration parameters, an instance of the target system is then composed by instantiating the target system components, interconnecting them and mapping them onto a hardware configuration. The FAST product line method [28], which emphasizes analyzing commonality and variability in a product line, uses parameterization extensively as one of the approaches to achieve variability.

A well-documented case study from the Software Engineering Institute (SEI) was done at CelsiusTech [7]. The Ship System 2000 (SS2000) is a product line for command and control of naval systems with nine reusers. The SS2000 contained components that were written with parameters and the reusers supplied the values for these parameters. The SS2000 features 3000–5000 parameters that must be individually tuned for each customer system built in the product line. CelciusTech has little or no methodological approach to ensure that no parameters are in conflict with each other. The fact that there are so many parameters in fact undermines some of the benefits gained.

3.1.2. Example from banking system case study

This paper uses examples from an Automated Teller Machine (ATM). The design used is from the Banking System Case Study (BSCS) [13]. The BSCS was modified to include six variation points to create a product line that can be used on several different banking solutions. The BSCS product line solves a problem for a bank that has several ATMs. Each ATM is geographically distributed and connected via a wide area network to a central server. Each ATM has a card reader, a cash dispenser, a keyboard display, and a receipt printer. The six variation points concentrate on the startup and validate PIN use cases.

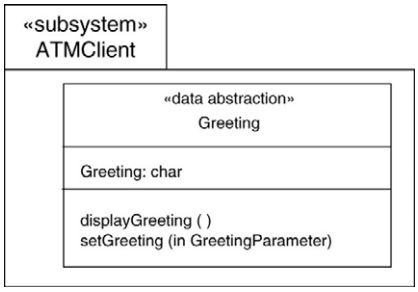

Fig. 1. Greeting—parameterization.

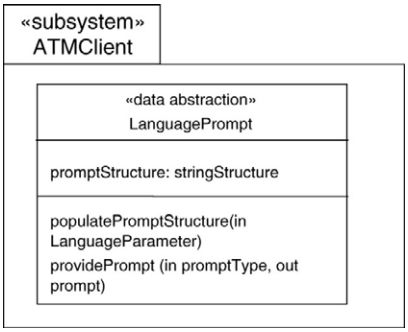

Fig. 2. Language—parameterization.

The following three examples of variability will be used:

- The ability to vary a greeting for display.
- The ability to vary the language of choice for display.
- The ability to vary the action if the card has expired.

Figs. 1 and 2 show the static modeling view of the variability using parameterization. The figures use the Unified Modeling Language (UML) notation [5,22], with the COMET class structuring criterion depicted as a UML stereotype, e.g., <<data abstraction>> [13].

With parameterization, a method is specified in the class interface to supply the value of the parameters needed. For example, Fig. 1 shows that the class Greeting has a method, setGreeting (in GreetingParameter), which is used to set the Greeting to be displayed. All greetings are predefined and the value chosen for the parameter determines the appropriate greeting to display.

Fig. 2 shows that the class LanguagePrompt has a method populatePromptStructure (in LanguageParameter), which is used to populate the structure with all the necessary prompts in the language desired. One solution is to have all language versions implemented in the LanguagePrompt class. The LanguageParameter is used to select the appropriate language.

The expired card variation is much more complicated to support using parameterization. The Banking System Case Study contains a class, ATMControl, which contains the state

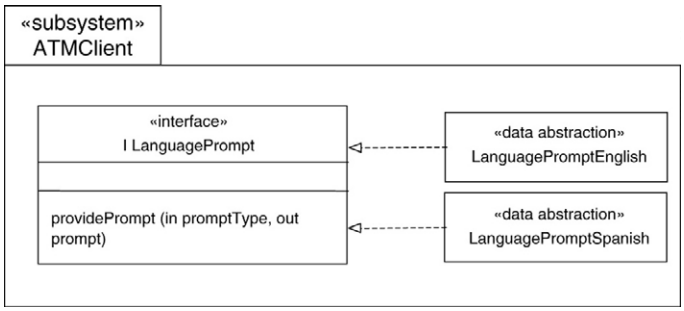

Fig. 3. Language—information hiding.

machine for the ATM machine. The state machine determines the response to an expired card, e.g., to eject or confiscate the card. Every possible reaction to an expired card must be coded in advance so that it can be selected by a parameter. This is liable to result in a much more complicated and error-prone implementation, particularly as further changes are made during maintenance. This illustrates the limitations of modeling variability using parameterization.

3.2. Modeling variability using information hiding

3.2.1. Approach

Modeling variability using information hiding is where several versions of the same component are built with the same interface. The variability is hidden inside each version of the component. In this case, the variants are the different versions of the same component. However all variants of a component must adhere to the same interface. This approach works well as long as changes can be limited to individual components and no changes are required to the interface.

From a reuse perspective, the reuser selects a component from a limited set of choices and inserts it into the application. The reuser does not need to be concerned about developing new variants, instead concentrating on selecting which version of the component to include in a given application.

Every domain analysis method studied thus far uses information hiding techniques. In addition to parameterization, FODA, DSSA, FAST, and EDLC all use information hiding. The Defense Advanced Research Projects Agency (DARPA) Software Technology for Adaptable, Reliable Systems (STARS) Program also used information hiding [24].

3.2.2. Example from banking system case study

Figs. 3 and 4 show how information hiding is used to vary the language and expired card. Since the greeting and language examples are similar, only the language example is shown. For each variation, a class is designed that has the same interface but hides the variation in the implementation. Using the UML notation, Fig. 4 shows the interface specified, and the different class realizations for the interface. Both classes, LanguagePromptEnglish and LanguagePromptSpanish realize the ILanguagePrompt interface. Since the class interface is common, these classes can be

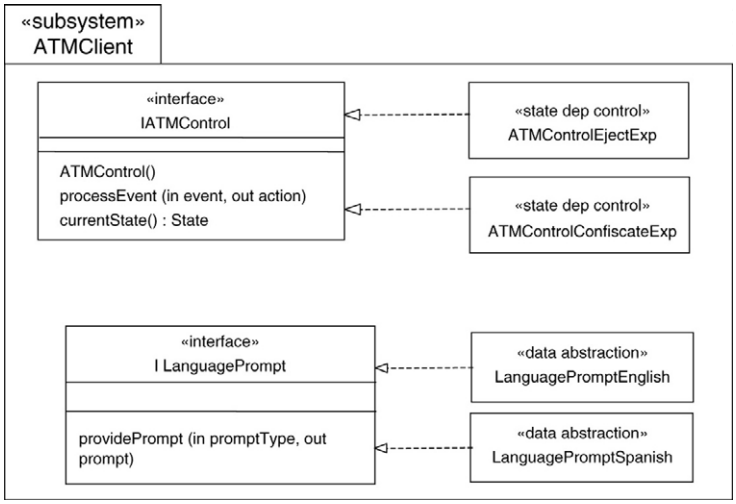

Fig. 4. ExpiredCard—information hiding.

interchanged with each other. LanguagePromptEnglish will respond with the desired prompt in English. LanguagePromptSpanish will respond with the desired prompt in Spanish. The application engineer chooses from a limited set of class implementations.

Fig. 4 shows the expired card example. This variation impacts two classes, ATMControl and LanguagePrompt. The LanguagePrompt variation is similar to the language variation in Fig. 3. The ATMControl variation contains the reaction to an expired card. Both classes, ATMControlConfiscate and ATMControlEjectExp realize the IATMControl interface. Since the class interface is common, these classes can be interchanged with each other. ATMControlConfiscateExp will respond to an expired card by confiscating the card and displaying a prompt saying the card was confiscated. ATMControlEjectExp will respond to an expired card by ejecting the card and displaying a prompt saying the card has expired. The application engineer chooses from a limited set of class implementations.

3.3. Modeling variability using inheritance

3.3.1. Approach

Modeling variability using inheritance is where variants are provided that do not have to adhere to the same interface. Variants can be specializations of other components where a subclass can extend the interface provided by a superclass by adding new operations or over-riding existing operations.

As with modeling variability using information hiding, with this approach the variants of the core asset components are usually limited and the reuser must select a variant from a predefined set of choices. The reuser or application engineer does not need to be concerned about developing new variants or components.

As product line processes were developed, variability using inheritance became more common. The KobrA Approach is an object-oriented customization of the Product Line

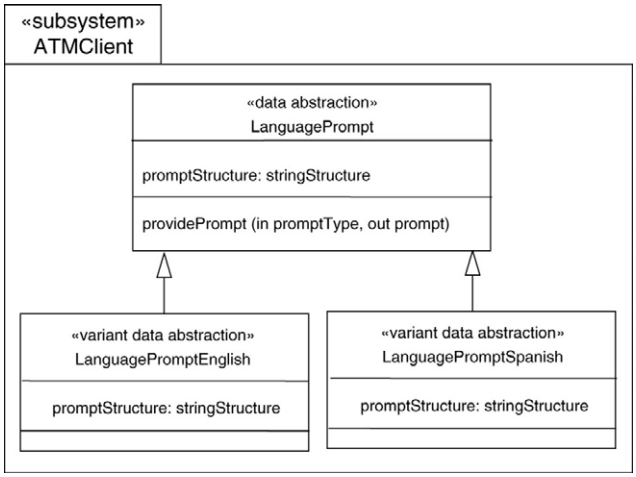

Fig. 5. Language—inheritance.

Software Engineering (PuLSE) [3] method [1,2]. The approach contains a framework engineering activity, which creates, and later maintains, a generic framework that embodies all product variants that make up the family, including information about their common and disjoint features. The application engineering activity instantiates the framework and chooses the particular variants in the product family, each tailored to meet the specific needs of different customers. In the UML based product line method [14], inheritance is used to model different variant subclasses of an abstract class, which are used by different members of the product line. Depending on the needs of the product line, the different variants might be mutually exclusive; alternatively the different variants might coexist in the same product line member.

3.3.2. Example from banking system case study

Figs. 5 and 6 show the same two variations with the inheritance approach. The language and greeting variations are handled in the same way, so only the language variation is shown. For the Language variation, a standard interface is specified in an abstract superclass. For each language, there would be a subclass that would inherit the operations specified in the interface, and provide an alternative implementation for the appropriate language. The specific language to be used would determine which alternative subclass is selected. For the expired card variation, in addition to the language subclass, the state machine is also inherited. The variant subclass contains a different state machine that provides a different reaction to the stolen card event. The method processEvent(), which executes the state machine, is adapted as needed. This allows the core assets developer to provide a large number of variants based on the parent class.

3.4. Comparison of approaches

Thus far, we have discussed 3 of the 4 approaches. Modeling variability using parameterization allows the reuser to populate attributes. The parameters that can be parameterized

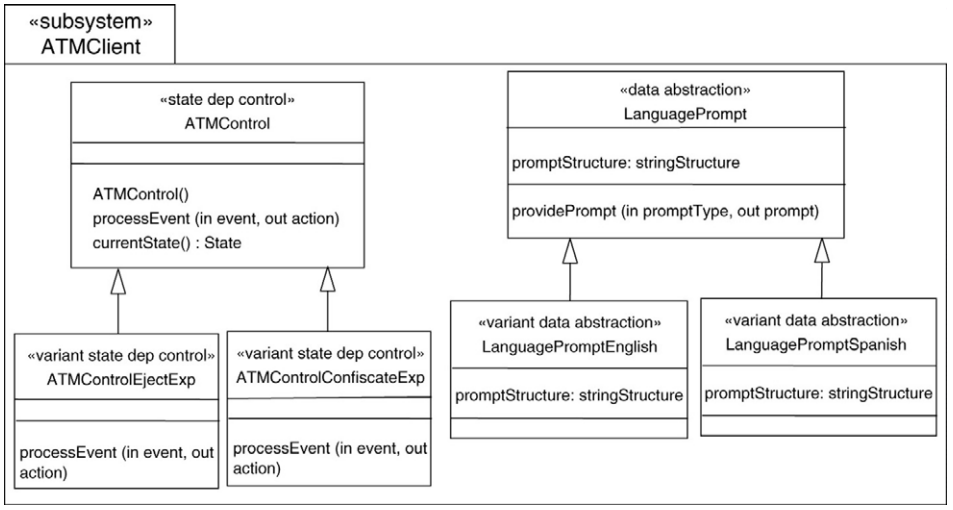

Fig. 6. ExpiredCard—inheritance.

must be communicated to the reuser. All of the core asset components adhere to an architecture and only the parameters need to be supplied. This can result in a very short time to market for a new application. The variability is limited in that no functionality can change. Using parameterization to select among alternative functionality is complicated and error prone, and is not recommended. Instead, one of the other approaches should be used.

Modeling variability using information hiding allows the reuser to choose variants from a limited set of choices in which the interface is common but the implementations are variable. This also results in a relatively short time to market, however the choices are limited. There is a higher variability than parameterization because functionality and parameters can be varied.

Modeling variability using inheritance allows the reuser to choose variants from a limited set of choices. In addition to the information hiding approach, it allows the core asset developer to manage the variation points. Because of the nature of inheritance, this approach also allows the core asset developer a wider range of variants by extending the superclass interface in variant subclasses. It also provides an approach to manage change in the core asset. If the change is in a parent class, the change must only be made once.

3.5. Modeling variability using variation points

Modeling variability using variation points is where the core asset components consist of variation points and the reuser may build target system components using unique variants built from the variation points. This approach provides the reuser with the most flexibility as it allows them to create their unique variants and maintain them.

This is in contrast to the other approaches that supply all the possible variants. In these cases, the application engineer would choose which variant they would like to use. The disadvantages of these approaches are the additional resources to develop the variants as

part of the core assets before deployment and the lack of flexibility for the reuser to create his or her own variant. If a new variant were desired, it would have to be added to the core assets. This adds cost to the maintenance and management of the core assets.

Therefore the advantages with modeling variability using variation points are less resources to develop the core assets or common core and to allow the reuser to create a unique variant not supplied with the common core or considered by the common core developer. This decreases the development time for the common core, which does not include the variants, but increases the time to market for the individual applications, as the variants must be created rather than chosen.

Discussing this approach is done by explaining the Variation Point Model (VPM). The following sections will show how variation points can be used to model the three different approaches to variability described in the previous sections, namely parameterization, information hiding, and inheritance. Each type is modeled differently as described in Section 4. VPM also models the callback variation point type.

4. Introduction to VPM

The Variation Point Model (VPM) is designed to model variation points, which are contained in reusable software components. Jacobson defines a variation point as follows: "A variation point identifies one or more locations at which the variation will occur" [17]. VPM adds the mechanism of variability to this definition. VPM can be used with any reusable software development process that contains variation points. VPM is most compatible with the Product Line Architecture or a family of systems. VPM has the following qualities:

1. VPM visualizes the variation points.
2. VPM adds the variation point view to the requirements.
3. VPM adds the variation points to the reference architecture.
4. VPM uses the well-known and accepted Unified Modeling Language (UML) [5,13,22]. Using UML to model software product lines is also described in [1,2,11,14,17,20].
5. VPM categorizes the variation points into four types of variation points, Parameterization, Information Hiding, Inheritance, and Callback.
6. VPM gives the reuser an understanding of how to build a variant from a variation point.

The structure of a variation point includes looking at the variation point from at least four different views. The structure of a variation point includes describing the variation point function in the requirements. The structure of a variation point includes showing all the components that are created to realize the variation point. Lastly, the structure of a variation point includes showing all its static and dynamic aspects. Each type of variation point has a different static and dynamic structure.

The following four views are necessary to adequately communicate the variation points to the reuser.

1. Requirements View
2. Component Variation Point View
3. Static Variation Point View
4. Dynamic Variation Point View

The *requirements variation point view* shows the variation point in terms of the requirements of the common core. This view is created during the analysis phase of developing the common core. This allows the variation points to be captured at the same time the requirements are captured.

The *component variation point view* is a composite view that shows all the variation points that belong to a particular component. This view also shows if a variation point belongs to more than one component and if several variation points belong to one component.

The *static variation point view* shows the static parts of the variation point and their relationships. This view shows which classes and methods are involved in the variation point. Since it is necessary to understand the variation point type to create this view, this is the stage at which the variation point type is determined. When a variant is created for a variation point, the component needs to know this variant now exists and the *dynamic variation point view* shows how to make the variant known to the rest of the components. For example, if a variation point is to create a new class that contains a new method, this new class and method must now be registered with the component so the component knows to use this new method.

The static variation point view and the dynamic variation point view are created during the design phase of the common core development. This is where the design of the component needs to be understood so the common core developer can also design the variation point. Together the static and dynamic variation point views give the reuser an understanding of how to build a variant from a variation point.

The Common Core Process (CCP) is a software process to create a product line using variation points. In summary, CCP has four phases. The first phase includes the domain analysis effort to create the requirements and the variation points. This requirements variation point view is created during this phase. This phase determines the different reuser needs for variations. However, since variation points are used, this phase does not need to determine all the different variants. Phase 2 included the architecture work to determine the different components of the common core. The component variation point view is created during this phase. The variation points in the requirements variation point view are traced to a component in the component variation point view. Phase 2 also determines if a variation point involves one or more components. Phase 3 includes the design of the common core. The static and dynamic variation point views are created in phase 3. The developer of the common core designs the variation point in this phase. Phase 4 includes the reuser building variants from the common core created in the first three phases. Reference [27] contains a comprehensive discussion of CCP.

This paper introduces the VPM model with an example of an Automated Teller Machine (ATM). The design used is from the Banking System Case Study (BSCS) [24]. There are six variation points that concentrate on the startup and validate PIN use cases. These use cases along with the six variation points were implemented in Java.

4.1. Requirements variation point view

The requirements variation point view shows the variation point in terms of the requirements of the core assets. This view is created during the analysis phase of

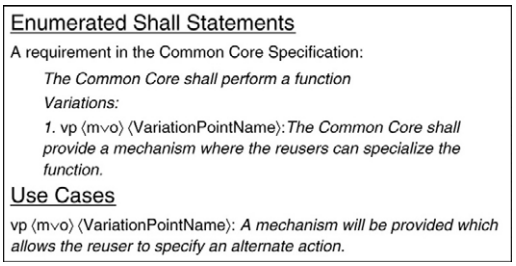

Fig. 7. Requirements variation point view.

developing the common core. Fig. 7 shows a generic depiction of the requirements variation point view in both the enumerated shall statement form and the use case form.

A variation point is labeled with the first two characters "vp" followed by an "m" for mandatory or an "o" for optional. A mandatory variation point does not supply a default variant. The reuser must supply one, and it is essential for the reuser to understand where the mandatory variation points exist.

An example for a stolen ATM card is given below.

vpoStolenCard. A mechanism will be provided which allows the reuser to add any action as a result of a stolen card. The default action is to confiscate the card.

The purpose of the vpoStolenCard variation point is to allow the bank to take a unique action to a stolen card based the banks unique business rules. A reuser may wish to call the police or call the local bank if a stolen card is presented to the ATM machine.

In summary, VPM includes a requirements variation point view. This view should be developed during the domain analysis phase of building a product line, and should have the following attributes:

1. The common core requirement should be clear and testable.
2. If the common core requirement allows variability for the reuser, it should be reflected in a variation point.
3. If the common core requirement is selected for use in a target system, a trace should be maintained.
4. The common core requirement should trace to a component category and a component.
5. A common core requirement may trace to more than one component. If so, it should be clear where the variation point traces.

4.2. Component variation point view

The component variation point view is a composite view that shows all the variation points that belong to a particular component. Fig. 8 shows a generic model of a component variation point view. This view shows the relationship from a component to its variation points. It shows that the variation point is allocated to the component. VPM models a component using the UML package. Since VPM looks at a component as a collection of classes, the UML package symbol is chosen to represent the collection. The UML stereotype <<Component>> is used to designate the package as a component. The variation

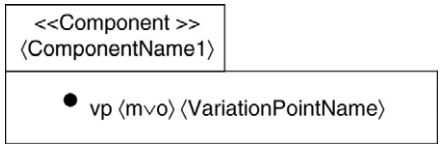

Fig. 8. Component variation point view.

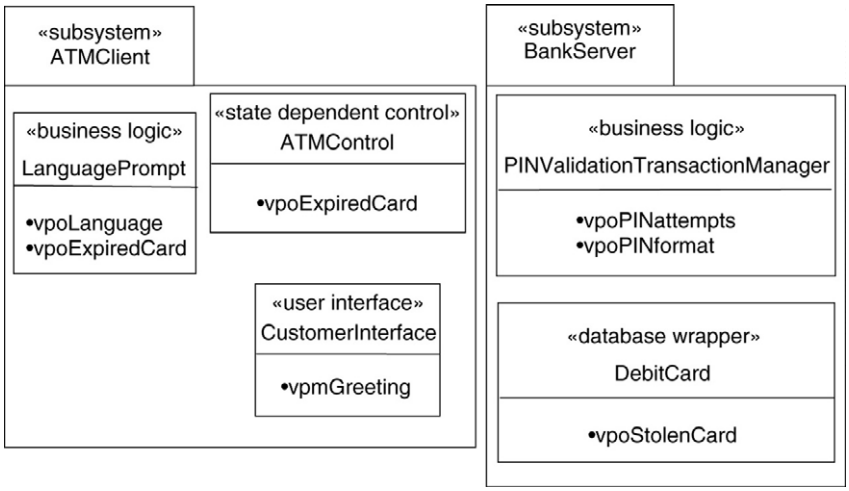

Fig. 9. Component variation point view example.

points are designated by a • symbol. Although not included in the UML User's Guide [5], Jacobson introduced the • symbol for variation points [17].

The component variation point view also shows if a variation point belongs to more than one component and if several variation points belong to one component. This view is created during the phase that also creates the architecture of the core assets. As requirements are allocated to components, so are the variation points. Fig. 9 shows an example of a component variation point view for the Banking System Case Study. The system was designed using the COMET/UML method as described in [13] and then modified to support VPM. The classes are structured using the COMET class structuring criteria, which are depicted using stereotypes such as <<user interface>>.

This example shows six variation points and how they are allocated to the common core. This figure gives a high-level view of where the variation points are and what components or classes are affected. If the reuser wants to exercise a variation point, they now know which class to extend. Notice that the class PINValidationTransactionManager contains two variation points. Also notice that the vpoExpiredCard variation point belongs to two classes, ATMControl and LanguagePrompt. This means that both classes must be examined to build a variant from this variation point.

The component variation point view is extremely useful for the reuser because it indicates not only where the variation points are, but also when a reuser must extend

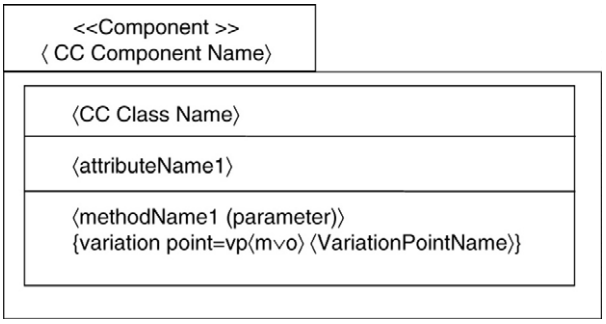

Fig. 10. Generic static variation point view for the parameterization type.

a component's functionality. The requirements variation point view and the component variation point view become part of the reference architecture and are completed in the analysis phase of a development. The static and dynamic variation point views are created in the design phase but are also part of the reference architecture provided to the reuser. They will differ a bit in that they will show some of the inside workings of the common core components.

4.3. Static variation point view

The static variation point view shows which classes, methods or attributes in the component make up the framework of the variation point. This view only shows what the reuser needs to know. There may be other classes affected by the variation point view, but only the classes the reuser needs to be aware of are shown in this view.

The static variation point view uses the UML class diagram. The class diagram shows the set of classes, interfaces, and collaborations and their relationships. The static variation point view includes the classes that are involved in the variation point.

The static variation point view uses two UML extensibility mechanisms: tagged values and constraints. The most critical part of the static variation point view is tagging the variation point. In other words, tagging the methods or attributes of the class that are involved in the variation point. A UML tagged value {variation point = vp ⟨m∨o⟩ ⟨VariationPointName⟩} is placed after the element it effects. Not only does this show the reuser the parts of a variation point, but it also shows which classes the variation point impacts that the reuser must understand in order to build a variant. The UML extension mechanism, constraint, is used to show that the relationships in the static variation point view are for reuse purposes only. In other words, these are the relationships the reuser uses to build a variant.

This research has concentrated on four types of variation points: parameterization, information hiding, inheritance, and callback. The static view is used to depict the variation point type; examples of all the variation point types are discussed in Section 4.5.

To introduce the static variation point view, the generic parameterization static variation point view is shown in Fig. 10. Anything within the symbols ⟨⟩ is for the builder of the variation point model to supply. The variation point tag is contained within brackets { }

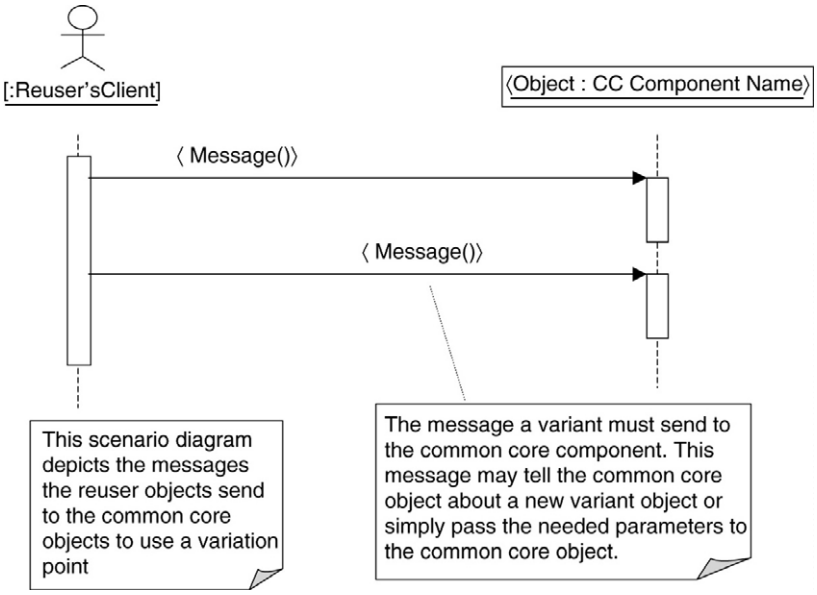

Fig. 11. Dynamic variation point view.

and is placed after any method or attribute that is to be overwritten in the common core class. This is the actual location of the variation point in the common core component. The symbols [ ] are used to designate information supplied by the reuser. The method used to supply these parameters is tagged with the variation point name in the common core component. This shows where the variation within the common core component is taking place without changing the common core component itself.

4.4. Dynamic variation point view

The dynamic variation point view uses the UML sequence diagram. The purpose of this view is to show the interaction needed for the common core component to use the new variant. UML has two interaction diagrams, the sequence diagram and the collaboration diagram. Either could be used; however, the sequence diagram emphasizes the ordering of messages, which is crucial in using the new variant. One dynamic variation point view is created for each variation point.

Once the variant is created there is dynamic behavior which must be understood to initialize or instantiate the variant. For example, if the variant is a derived class, how does the common core component know that this new derived class exists and how to use it? As for parameterization, some parameters must be set in a certain order; the dynamic variation point view shows the order of events. Fig. 11 shows a generic dynamic variation point view. Only the classes needed to create the variation point are shown. The common core component may take the parameters or classes passed in and use them with several other classes in that component; this information is abstracted away from the reuser.

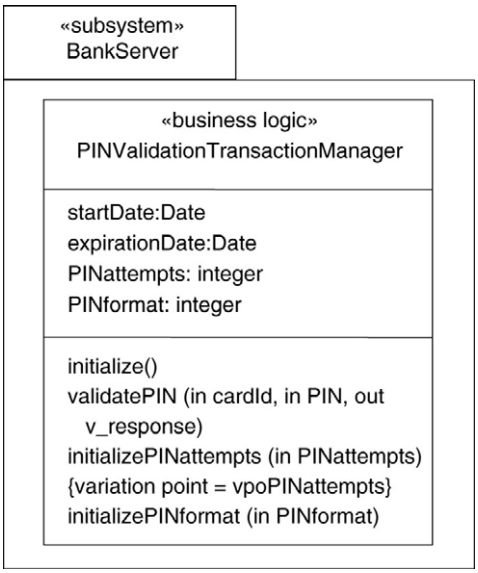

Fig. 12. Parameterization static variation point view for vpoPINattempts.

4.5. Variation point types

As stated previously, there are four variation point types that have been modeled thus far. The six variation points shown in Fig. 9 are used to illustrate how to model the different variation point types.

4.5.1. Parameterization variation point type

Parameterization allows the common core developer to build a component with the specific values of the parameters to be supplied by the reuser. The parameterization is supplied to common core component via an interface. The parameters are defined during the design stage and modeled in this view.

A generic parameterization static variation point view is shown in Fig. 10. The method used to supply these parameters is tagged with the variation point name in the common core component. This shows where the variation within the common core component is taking place without changing the common core component itself.

An example of a parameterization static variation point view along with its dynamic variation point view is shown in Figs. 12 and 13. The vpoPINattempts variation point as described in the requirements variation point view allows the reuser to define the number of attempts a customer has to enter the correct PIN. Fig. 14 shows that the reuser uses the initializePINattempts (in PINattempts) method to supply the parameter (the number of PIN attempts a customer is allowed) into the class PINValidationTransactionManager that belongs to the common core component BankServer.

Modeling parameter variability with variation points is done with the parameterization type variation point. Fig. 14 shows the vpoLanguage variation point. Fig. 14 is similar to Fig. 2, except for the explicit description of the variation point. In this case a UML tag is

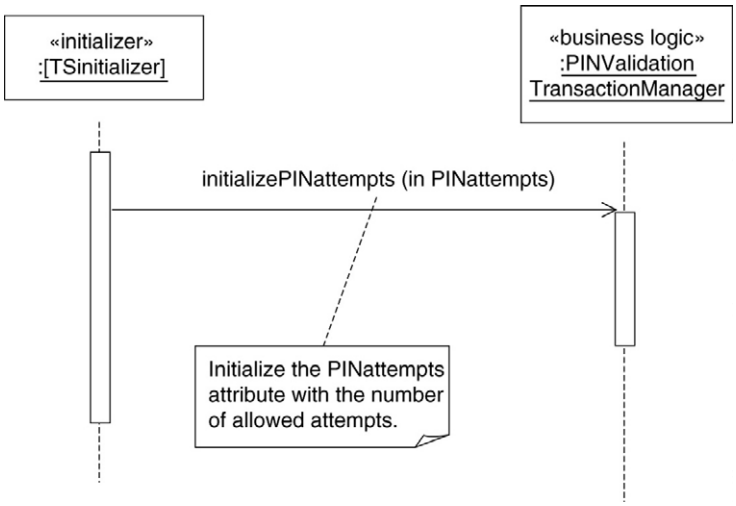

Fig. 13. Dynamic variation point view for vpoPINattempts.

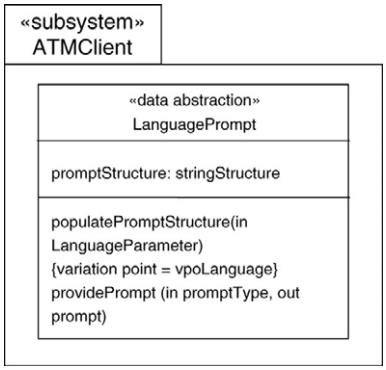

Fig. 14. vpoLanguage—parameterization variation point type.

added after the method populatePromptStructure (in LanguageParameter) to label the method in the class interface that the reuser uses to select the language for prompts.

4.5.2. Information hiding variation point type

Information hiding allows the common core developer to build a component with a common interface which hides the specific implementation of the interface. The interface is exposed to the reuser and the reuser is able to build a variant with the same interface to replace the functionality provided by the common core.

A generic information hiding static variation point view is shown in Fig. 15. The class containing the interface is tagged with the variation point name in the common core component. This shows that the class itself is the variation point and the reuser must create a class with the same interface in order to provide unique functionality.

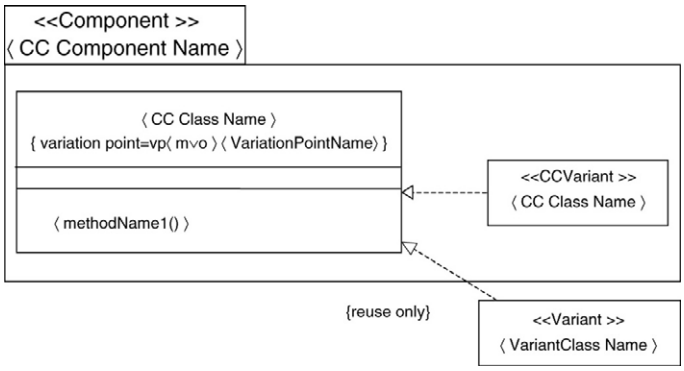

Fig. 15. Static variation point view for the information hiding type.

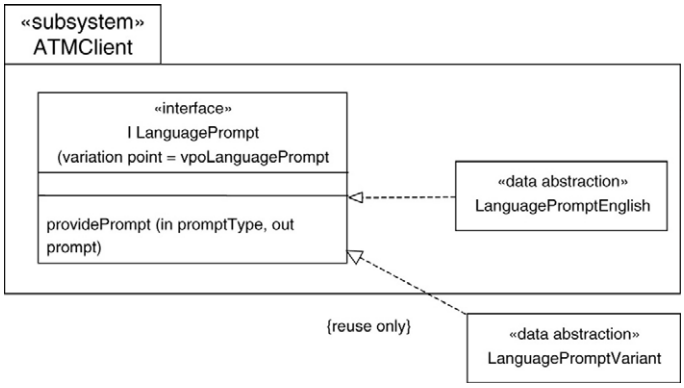

Fig. 16. vpoLanguage—information hiding variation point.

An example is shown in Fig. 16, which is similar to Fig. 3. In this case a variant is added to show that the reuser may provide additional variant classes that realize the ILanguagePrompt interface, i.e., with the same class interface but different implementations for each language. In addition, the constraint {reuse only} shows that the variant is supplied by the reuser or application engineer. This allows the reuser to build any variant as long as it complies with the variation point. Because the reuser now must understand the entire class that is being realized, the variation point tag is put under the class name.

4.5.3. Inheritance variation point type

Fig. 17 shows the static variation point view for the inheritance type. In this view the variation point model developer will supply the name of the common core component, the name of the class contained in the common core component, and the name of any attributes and methods which make up the class. The derived class name is to be supplied by the reuser. The inheritance relationship is constrained with “reuse only” to explicitly say that the common core component does not supply this variant, only the variation point.

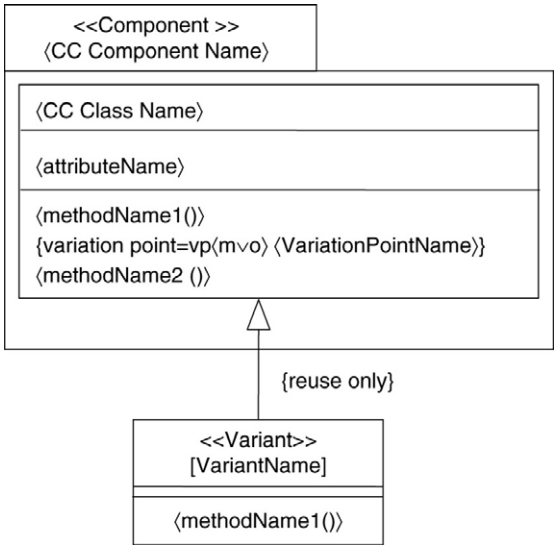

Fig. 17. Static variation point view for the inheritance type.

If the variant were to include other classes derived from other components, this would also be shown in this view and all methods or attributes would have the same variation point tag with the same variation point name to show all the parts of the variation point.

An example of an inheritance static variation point view along with its dynamic variation point view is shown in Figs. 18 and 19. The variation point vpoLanguage is defined in the requirements to allow the reuser to define any language to display on the ATM machine. This variation point is allocated to the class LanguagePrompt. The static variation point view in Fig. 18 tells the reuser to derive a new variant and override the initialize() method. The initialize() method sets the prompts to be displayed. The LanguagePromptVariant() constructor for the variant subclass will use the initialize() method to populate the Prompt Structure with all the prompts to be displayed in the language of choice.

The dynamic variation point view in Fig. 20 shows the reuser how to register the new variant so the common core component, ATMClient, will now use this new variant. Inheritance allows the core asset developer to build a component that can be specialized by the reuser. However, the new specialized class must now be known to the core asset so that it may be used. There are two basic ways to deal with a new variant, a Class Factory or a Broker. This example uses the Class Factory [12].

The static variation point view is used to illustrate the Modeling Inheritance Variability with Variation Points approach. Fig. 20 shows the vpoExpiredCard variation point. For vpoExpiredCard, the state machine is inherited in the constructor ATMControl(). This constructor calls the processEvent() method to set up the state machine for that variant. Since this will create a different message to be displayed the LanguagePrompt class must also be inherited. An initialize() method is inherited along with the class constructor. This allows the common core developer to create variants by changing the constructor,

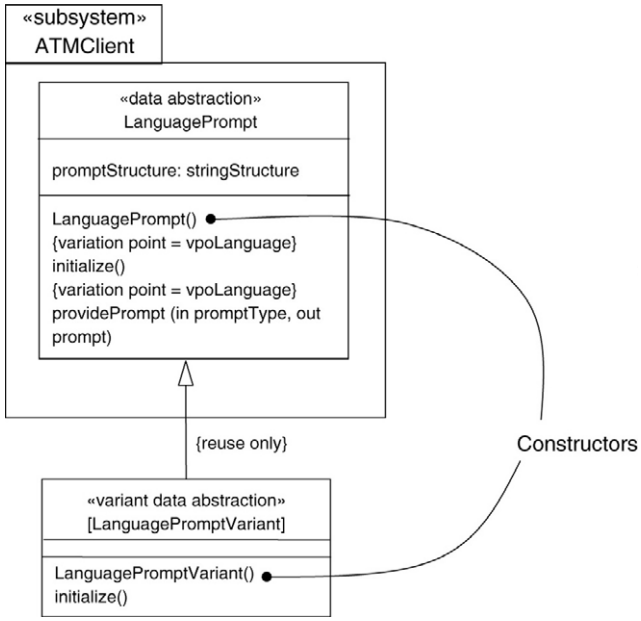

Fig. 18. Inheritance static variation point view for vpoLanguage.

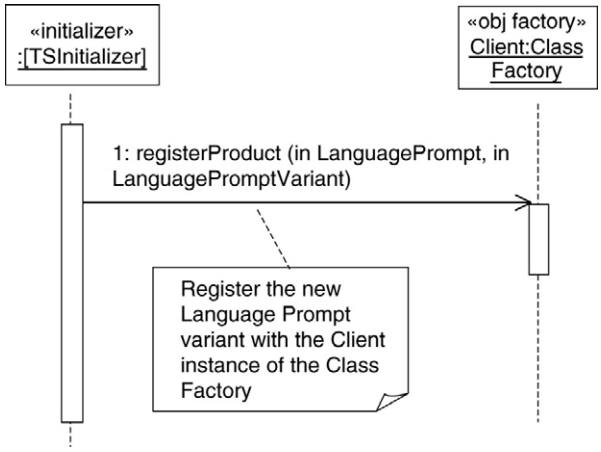

Fig. 19. Dynamic variation point view for vpoLanguage.

which calls the initialize method. The variant initialize() method will contain the specifics for the language prompts. This is different from Fig. 8 because the attributes cannot be inherited. Since the variant is built by the reuser, it is not known by the parent class or common core component. Therefore, it becomes known to the common core via a registration technique, which occurs during run time. The run time constraint means that an attribute cannot be inherited, therefore the initialize() method contains the new

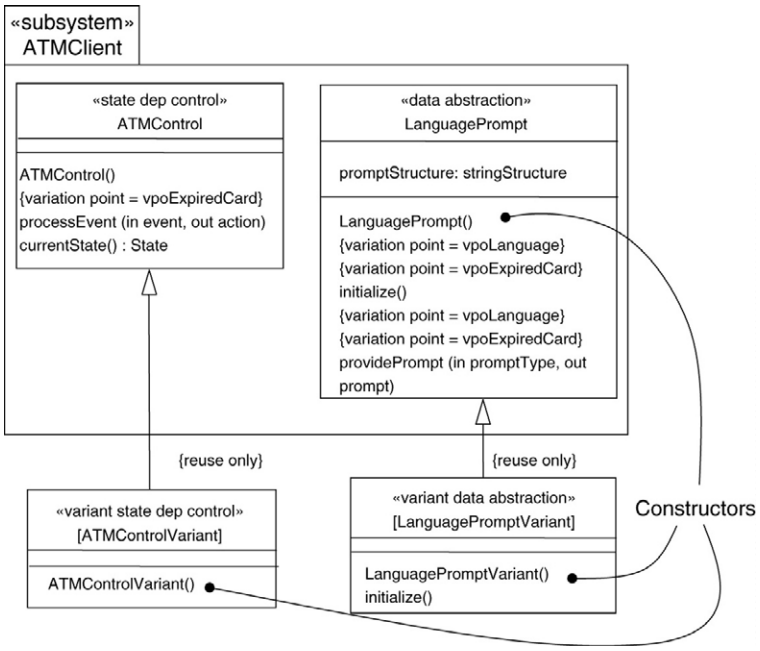

Fig. 20. vpoExpiredCard—inheritance variation point type.

language prompts. It should also be pointed out that when a common core component has two variation points, such as LanguagePrompt, the reuser implements one variant that addresses both variation points, e.g., if the language selection is English and the expired card choice is to confiscate the card, the appropriate prompt is "your card has expired and has therefore been confiscated—please contact your branch".

In the inheritance approach, the reuser selected a variant from a list of possible variants. Only the core asset developer understood that inheritance is used. In this case, the reuser must understand the variation point so they can build a unique one. Fig. 20 shows the reuser which method the reuser must override in order to build a variant. In addition, the common core must now contain a class factory or broker to manage the variants. During initialization of the application, the reuser will register his or her variants with the common core. That is how the common core components know which variant to use during execution.

4.5.4. Callback variation point type

Callback allows the common core developer to build a component that contains a callback to the reuser's code. This allows control to be passed to the reuser's code. The common core component contains a registry of callbacks. A callback is a pointer to a reuser provided function. When a common core method is executing which may have a callback, this method will check to see if a callback is registered and then call that method in the variant class. The common core components know about callbacks because the reuser registers the callback with the common core. Therefore, the reuser can perform whatever function is needed and return control to the common core.

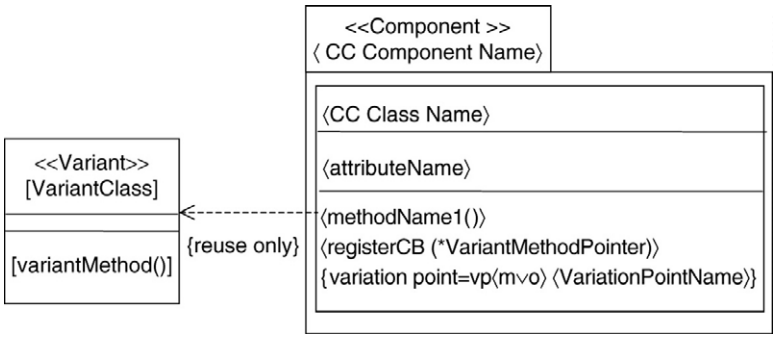

Fig. 21. Generic static variation point view for the callback type.

Fig. 21 shows a generic static variation point view for the callback type. The variation point tags show the parts of the variation point the reuser needs to know about. The reuser does not need to know which method is using the variant. The reuser needs to know how to build the variant and how to register it. Fig. 21 shows that the common core has a set of methods that may have a need to call a variant. It is not important which method calls the variant. The reuser builds the variant class. This variant class is a class that contains a member function. The callback is a pointer to that member function. The reuser then registers this callback with the common core component. This registration is done by a common core developer supplied interface. In this case, the interface or method is registerCB (*VariantMethodPointer). During registration, the reuser's code sends the common core component a callback that is a pointer to the variant's method. The method to register the callback is tagged. This is the method the reuser uses to register the callback that in turn extends the functionality of the common core component.

Since a callback is done with a callback to the reuser's variant class, the common core component is dependent on the variant. This is shown in UML by a dotted arrow. Another part of the reuser's code registers the variant.

An example of a callback static variation point view along with its dynamic variation point view is shown in Figs. 22 and 23. The requirement variation point view described vpoGreeting as requiring the reuser to supply a greeting unique to his or her bank. Fig. 22 shows that the reuser must create a variant which contains this new greeting. Fig. 23 shows the reuser how to register this new variant. Customer Interface may now call the new variant, Greeting, to retrieve the greeting for display.

Unfortunately the callback variation point type is dependent on the ability to use a pointer to a function. Since this capability is not contained in the Java™ programming language, an interface is used instead. Figs. 24 and 25 show the static and dynamic variation point views for vpmGreeting using the Java implementation. An interface is modeled as a class because it contains a method () the reuser must implement in their variant. Fig. 24 shows that the reuser creates a variant and the variant realizes the IGreeting interface. Java does not allow an interface to be instantiated. Therefore an instance of the variant is registered with Customer Interface. This registration is shown in Fig. 25.

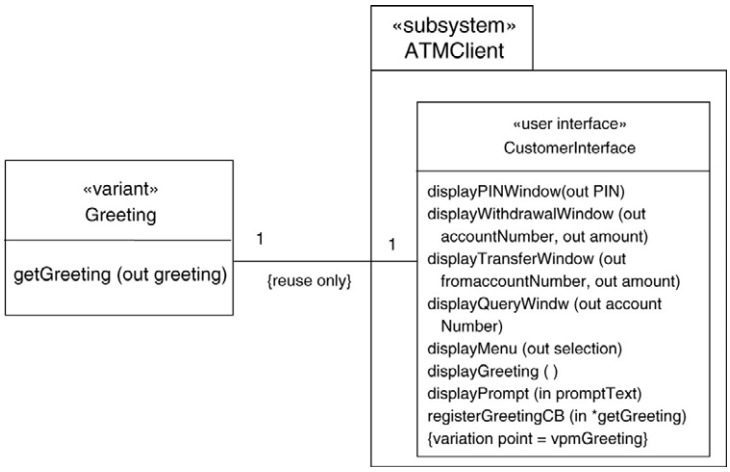

Fig. 22. Callback static variation point view for vpmGreeting.

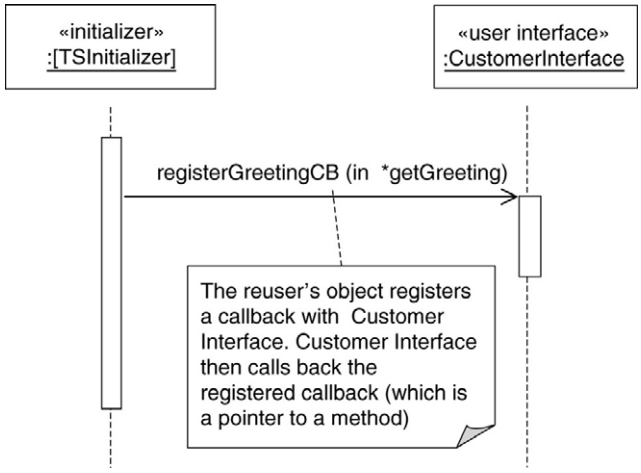

Fig. 23. Dynamic variation point view for vpmGreeting.

5. Proof of concept

A proof of concept exercise was done to demonstrate that variation points modeled using VPM could be implemented. The proof of concept exercise was performed to create a system given the variation points modeled using VPM as part of the software architecture. This exercise implemented the ATM problem using the Java programming language. Six variation points were implemented as illustrated in Fig. 9. A core asset was developed first and tested in an application engineering environment. Once the core asset was completed, four target systems were created. This exercise achieved its objective, which was to

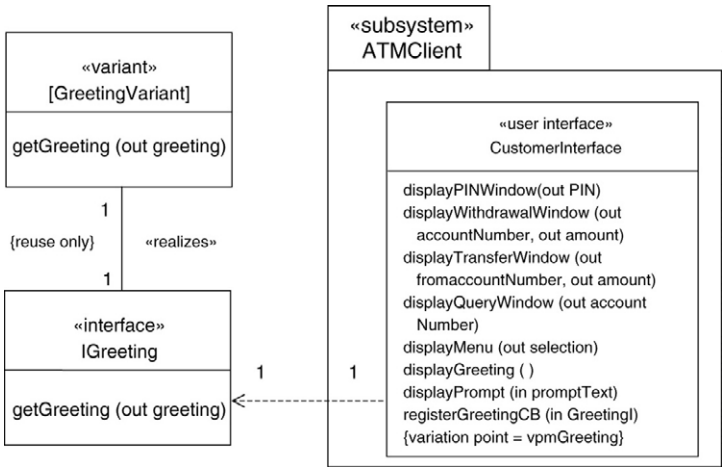

Fig. 24. Static variation point view for vpmGreeting—Java implementation.

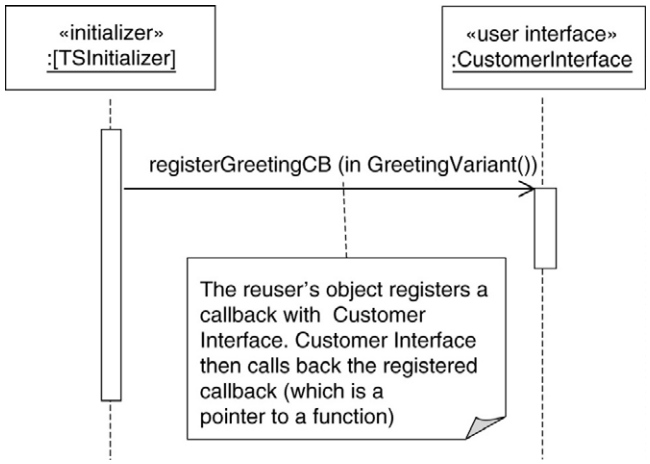

Fig. 25. Dynamic variation point view for vpmGreeting—Java implementation.

demonstrate that the variation points could be implemented using the variation points modeled using VPM.

One of the findings from the proof of concept involved the inheritance variation point type. Our first attempt was to override the parameter promptStructure in the variant. However, since the variant is registered with the core asset, the binding between the core asset component and the variant is dynamic. Overriding an attribute is not supported in dynamic binding. In order to use dynamic binding, a virtual function must be declared in the core asset component. The variant must then inherit this function. This resulted in Fig. 18 which shows the initialize() method along with the constructor which calls the initialize() method. This implementation allowed the variant to be registered and therefore

binding it to the core asset component. The implementation as a proof of concept exercise is discussed in more detail in [27].

In addition to the ATM implementation, a survey was conducted on a case study, the Control Channel Toolkit (CCT) [8]. CCT is a software asset base commissioned by the National Reconnaissance Office (NRO) and built under contract by Raytheon. CCT was built using the modeling variability using variation points approach; however, Raytheon did not have a method for modeling the variation points and exposing them to the reusers. This research surveyed both the developers of the CCT and one of its reusers. All 12 participants had experience using components that contain variation points, but not modeling them. So in the process of creating the components and reusing the components neither developer benefited from a variation point model. Therefore they were able to estimate the amount of work they spent understanding the variation point without a model to estimate any savings a model might provide. They were asked to evaluate each of the four views in VPM.

Two people felt that there was no usefulness or savings in at least one view. One reuser felt that the component variation point view was not needed but the static and dynamic variation point views were. Once CCT developer felt that the requirements variation point view and the component variation point view were good enough and the static and dynamic variation point views were not needed. All other responses indicated usefulness and savings of all views. In summary, every respondent found usefulness or savings in at least two of the four views. The majority of the participants felt the savings were in the 1–10% range. For a program of this size, this is significant savings of effort.

6. Conclusions

It is important to understand the different approaches to modeling variability and building a product line. This paper has described four different approaches to modeling variability, using parameterization, using information hiding, using inheritance, and using variation points. During the up front study to determine if a product line is feasible, the approach to modeling variability should be considered. The degree of flexibility could result in acquiring more reusers or it could result in impacting the expected time to market.

If the number of applications is limited, then the added flexibility is not as pertinent. However, if the system is a complicated one where the core asset developer could not possibly create all the variants, then modeling variability with variation points is very advantageous. It is critical for the reuser to understand the variation points and how to use them to create variants. For this reason, a modeling approach, such as the VPM described in this paper, is crucial.

Modeling variability using variation points exposes the variation points to the reuser. This approach provides the reuser with the most flexibility as it allows them to create their unique variants and maintain them. The advantage with modeling variability using variation points are less resources to develop the core assets or common core and to allow the reuser to create a unique variant not supplied with the common core or considered by the common core developer. The disadvantage of this approach is the extra management needed to understand and manage when a variant should become part of the common core or whether it should be left as unique to an application. This is in contrast to the other

approaches that supply all the possible variants. In these cases, the application engineer would choose which variant they would like to use. The advantages of these approaches are the decrease in the time to market because the application developer does not need to create a variant, only choose a variant.

No matter which approach is taken, the VPM can be used to document the variation points for the product line stakeholders and thereby aid in the maintenance and evolution of the core assets.

Acknowledgements

The authors thank Dave Rine for his valuable advice and Vinesh Vonteru for his implementation of the VPM proof of concept prototype. Acknowledgements are also due to Addison Wesley Publishing Company for permission to use material from the Banking System Case Study in Chapter 19 of "Designing Concurrent, Distributed, and Real-Time Applications with UML", by Hassan Gomaa, Addison-Wesley Object Technology Series, 2000.

References

[1] C. Atkinson, J. Bayer, O. Laitenberger, J. Zettel, Component-based software engineering: the KobrA approach, in: Proceedings, 3rd International Workshop on Component-based Software Engineering, 2000.
[2] C. Atkinson, J. Bayer, D. Muthig, Component-based product line development: the KobrA approach, in: Proceedings, 1st International Software Product Line Conference, 2000.
[3] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J. DeBaud, PuLSE: A methodology to develop software product lines, in: Proceedings of the Fifth Symposium on Software Reusability, 1999.
[4] J. Bergey, G. Campbell, P. Clements, S. Cohen, L. Jones, B. Krut, L. Northrop, D. Smith, Second DoD Product Line Practice Workshop Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October, 1999. Available online at http://www.sei.cmu.edu/publications/.
[5] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1999.
[6] J. Bosch, Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, Harlow, England, 2000.
[7] L. Brownsword, P. Clements, A Case Study in Successful Product Line Development, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October, 1996.
Available online at http://www.sei.cmu.edu/publications/.
[8] P. Clements, L. Northrop, Software Product Lines Practice and Patterns, Addison-Wesley, 2002.
[9] M. Coriat, J. Jourdan, B. Fabien, The SPLIT Method, Alcatel/Thomson-CSF Common Research Laboratory, France, 2000.
[10] W. El Kaim, Managing variability in the LCAT SPLIT/Daisy model, in: Proceedings, Product Line Architecture Workshop SPLC1, August, 2000.
[11] W. El Kaim, S. Cherki, P. Josset, F. Paris, Domain analysis and product-line scoping: a Thomson-SCF product-line case study, in: Proceedings, Software Product Lines: Economics, Architectures, and Implications, June, 2000.
[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[13] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
[14] H. Gomaa, Object oriented analysis and modeling for families of systems with the UML, in: Proc. IEEE International Conference on Software Reuse, Vienna, Austria, June, 2000.

[15] H. Gomaa, G.A. Farrukh, Methods and tools for the automated configuration of distributed applications from reusable software architectures and components, IEEE–Software 146 (6) (1999).
[16] B. Hayes-Roth, A domain-specific software architecture for adaptive intelligent systems, IEEE Transactions on Software Engineering (1995).
[17] I. Jacobson, M. Griss, P. Jonsson, Software Reuse-Architecture, Process and Organization for Business Success, ACM Press, New York, NY, 1997.
[18] K. Kang, Feature-Oriented Domain Analysis, Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
[19] D. McComas, S. Leake, M. Stark, M. Morisio, G. Travassos, M. White, Addressing variability in a guidance, navigation, and control flight software product line, in: Proceedings SPLC1, Product Line Architecture Workshop, August, 2000.
[20] M. Morisio, G. Travassos, M. Stark, Extending UML to support domain analysis, in: Proceedings, 1st International Software Product Line Conference, Pittsburgh, PA, 2000.
[21] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.
[22] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, MA, 1999.
[23] D. Sharp, Reducing avionics software cost through component based product line development, Product Line Issues Action Team (PLIAT) Workshop, 1998.
[24] M. Davis, Adaptable Reusable Code, Proceedings of the 1995 Symposium on Software Reusability, August, 1995, ACM Press, 1995.
[25] Software Technology for Adaptable, Reliable Systems (STARS), Product-Line Application Engineering Guidebook, Boeing STARS Technical Report, CDRL 0512, Advanced Research Projects Agency (ARPA) STARS Technology Center, Arlington, VA, July 1993.
[26] D. Webber, H. Gomma, Modeling variability with the variation point model, in: Proceedings International Conference of Software Reuse, Austin, TX, 2002.
[27] D. Webber, The variation point model for software product lines, Ph.D. Dissertation George Mason University, 2001.
[28] D. Weiss, C. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, Reading, MA, 1999.