Implementing Product Line Variabilities

Michalis Anastasopoulos
Fraunhofer Institute for
Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
+49 (0) 6301 707 168
anastaso@iese.fhg.de

Cristina Gacek[1]
Centre for Software Reliability
Department of Computing Science
University of Newcastle
NE1 7RU Newcastle upon Tyne, United Kingdom
+44 (0) 191 222 5153
cristina.gacek@ncl.ac.uk

ABSTRACT

Software product lines have numerous members. Thus, a product line infrastructure must cover various systems. This is the significant difference to usual software systems and the reason for additional requirements on the various assets present during software product line engineering. It is imperative that they support the description of the product line as a whole, as well as its instantiation for the derivation of individual products.

Literature has already addressed how to create and instantiate generic product line assets, such as domain models and architectures to generate instance specific ones [1, 2, 3], yet little attention has been given on how to actually deal with this genericity at the code level.

This paper addresses the issue of handling product line variability at the code level. To this end various implementation approaches are examined with respect to their use in a product line context.

Keywords

Software product lines, product line variability, implementation approaches, implementing variabilities, traceability

1. INTRODUCTION

Software product lines have numerous members. Thus, a product line infrastructure must cover various systems. This is the significant difference to usual software systems and the reason for additional requirements on the various assets present during software product line engineering. It is imperative that they support the description of the product line as a whole, as well as its instantiation for the derivation of individual products.

To reflect a product line, the generic assets have to cover all elements the product family is built from, and their corresponding composition rules. Clarifying how the various parts may be combined is a very challenging task.

Generic assets differ from specific ones in the fact that they embrace common and variable product aspects. Generic assets can be instantiated, that is, product-specific assets can be derived from them. Specific assets on the other hand are uniquely mapped to product line members.

Products within a product line context consist of constant as well as variable elements. Dependencies between these elements do exist (e.g., elements may exclude each other or one element may make the integration of a second one a necessity). Hence only a subset of all combinations are correct and complete configurations. Therefore we have to restrict the product line infrastructure to cover only the set of valid family members.

[1] This work was done while the author was working for the IESE

A decision model structures the variability within a product line as a set of decisions to be resolved. Interrelations between decisions are captured as well. Such a model is also generic, that is, it can be also instantiated. In the instance all decisions are resolved. Every diversity in each generic asset must be connected to an open

decision in the decision model. The presence of a decision model is important because a decision model specifies an instance of each generic asset and thus a complete product line member.

Literature has already addressed how to create and instantiate generic product line assets, such as domain models and architectures to generate instance specific ones [1, 2, 3], yet little attention has been given on how to actually deal with this genericity at the code level. Variabilities and their composition rules must also be reflected at the code level, and be instantiated when creating individual product instances.

This paper addresses the issue of handling product line variability at the code level. To this end various implementation approaches are examined with respect to their use in a product line context. The rest of this paper is structured as follows: section 2 introduces some of the relevant related work, this is followed by a discussion on the types of variabilities to be supported and the requirements for their implementation support (section 3), we then introduce our framework for comparison of implementation approaches (section 4), present some examples of the techniques discussed (section 5), and briefly discuss evaluation criteria needed for implementation techniques (section 6). Section 7 comments on some of our practical experience and section 8 concludes the paper with a short discussion of the various research needs we have been able to elicit so far.

It should be noted that this paper's purpose is not to impart profound insights into the technologies but only to report on the possibilities of their use for handling code variabilities in a product line. Finally we must mention that the set of approaches presented here is not complete. We are certain that other approaches may also be advantageous in this context. We believe though that their majority bears upon the collection introduced here.

2. RELATED WORK

James Coplien in his book "Multi-Paradigm Design for C++" [4] presents a design method that takes advantage of the popular programming language's support for multiple paradigms, such as classes, over-loaded functions, templates, modules, and procedural programming. Each of those paradigms is described as being uniquely suitable for solving a specific class of design problems. This conclusion is particularly important for product line engineering because the book has a strong focus on problems related to the realization of commonalities and variabilities.

David Sharp has studied [5] existing object oriented techniques that facilitate anticipated variabilities between different software versions. Parameters of variations, which refer to the different aspects of software that can vary, are being initially identified. Tailoring techniques featured by languages like C++ and Ada95 are described next. Each technique is categorized with respect to the supported combinations of variabilities for each parameter of variation. Scope, flexibility, efficiency and applicability are some of the criteria by which the use of the explored techniques are judged.

Mikael Svahnberg [6] presents a taxonomy of the technical solutions to manage product differences, a historical essay of how components in a software product line can evolve and what mechanisms are used to support this evolution. From this he elaborates on the connection between evolution and variability. The implementation mechanisms are assigned to phases of the development lifecycle and consequences as well as examples of each mechanism are cited.

Mira Mezini in her Ph.D. dissertation [7] introduces a composition approach named Variational Object-oriented Programming (also known as VOP). She identifies different kinds of variations and proves that standard object orientation cannot deal with them effectively. Subsequently she defines an extension of the standard object oriented programming model in order to support context-dependend behavior variations.

Krzystof Czarnecki and Ulrich W. Weisenecker define Generative Programming as: "a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be manufactured on demand from elementary, reusable implementation components by means of configuration knowledge" [8] The authors provide a broad survey of domain engineering and implementation techniques that support the concepts of Generative Programming. Aspect-oriented Programming (AOP), Subject-oriented Programming (SOP) and VOP as well as Generic Programming, Generators and Intentional Programming are covered.

AOP and SOP are covered below. The goal of Intentional Programming (IP) is to develop a new kind of environment for transformational programming that permits software to be composed from a set of independent design decisions or "intentions", using domain-specific notations and optimization strategies [9].

3. REQUIREMENTS FOR IMPLEMENTATION SUPPORT

Every variability encountered in a product line context can be connected with a corresponding feature the support of which is varying under specific conditions. Since a feature implementation is usually spread across many source files and modules the relation between features and variabilities (variation points) is 1:N. This however does not hold for AOP and SOP as we will see in section 4.

Table 1 provides an overview of feature types and the criteria for inclusion of a feature in a product line instance. As we will see later on, knowing the type of a feature is important for the implementation because given a feature type a certain implementation technique can be recommended or not.

Table 1. Feature Types

Feature Type	Meaning
Mandatory	The feature must be always included.
Optional	The feature is an independent complement that may be included or not.
Alternative	The feature replaces another feature when included.
Mutually Inclusive	In order for the feature to be included specific other feature(s) must be included as well and vice versa.
Mutually Exclusive	In order for the feature to be included specific other feature(s) must be left out and vice versa.

The mutually inclusive and exclusive features are subject to specific constraints and have been separated from the other ones in the above table because they refer to different feature classes (e.g an optional feature can be also mutually inclusive).

As postulated in [5] variabilities may be primarily classified as positive and negative. Positive variability adds functionality while negative removes. Variabilities can be categorized as shown in table 2.

Table 2. Variability Types

Variability Type	Meaning
Positive	Functionality is added
Negative	Functionality is removed
Optional	Code is included.
Alternative	Code is replaced.
Function	Functionality changes
Platform / Environment	Platform or environment changes

Additional assistance in handling variabilities at the code level can be provided if the exact binding time of the variability is known. We classify binding time as follows:

- Compile-time: The variability is resolved before the actual program compilation (e.g., with preprocessor directives) or at compile time.
- Link-time: The variability is resolved during module or library linking (e.g., selecting different libraries with different versions of the exported operations)
- Runtime: The variability is resolved during program execution (e.g., depending on user rights functionalities get disabled or enabled with conditions in the code)
- Update-time or Post-runtime: The variability is resolved during program updates or after program execution(e.g., an update utility adds functionality to existing modules)

After having the feature type and the binding timings identified we also need to determine the parameters of variation [5], namely what exactly is varying, and the variation points, i.e., the locations where changes occur.

The main parameters of variation as identified in [5] are the interfaces and the corresponding implementations. We also consider the initializations of a module as a possible parameter of variation. Initialization sectors are often encountered in Delphi units where they are optional and contain statements that are executed on program start-up.

In order to ensure effective maintenance and systematic change management traceability must be provided from the architecture and the design to the code. Traceability is the ability to document and follow the life of a concept throughout system development. It is forward directed (post traceability: describing the deployment and use of a concept) as well as backward directed (pre traceability: describing the origin and evolution of a concept) [1].

A standard way of providing traceability is the establishment of cross reference data. Such references can be expressed as links or matrices where connections between the various artifacts in code and architecture/design are made explicit. This presupposes that the artifacts to be traced have been firstly spotted which brings up again the essential need of determining the variation parameters and points.

A major issue which we have faced in our project experience is the scalability issue. Many of the implementation approaches used in industry for handling variability in a product line suffer when new products are engaged in or when existing products are evolving. This impacts at first the overview over the variation points leading gradually to a degraded implementation structure. In many cases developers try to manage the upcoming growth and evolution by rejiging their implementation approaches or by introducing step-wise new ones.

Another important issue is the Separation of concerns (SoC). SoC states that important issues should be represented in programs intentionally (explicitly, declaratively and with little or no “extra noise”) and should be well localized. This facilitates understandability, adaptability, reusability and other qualities since intentionality and localization allow easy verification of the way a program implements its requirements. [8]

At the implementation level separation of concerns is very often achieved by separating constant from variant parts into distinct modules. However, when interdependencies between features and consequently between the implementing modules exist they should be explicitly made clear. Otherwise maintenance of the separated modules can cause serious defects. This recommendation holds for each one of the affected techniques presented below.

As we will explore in the following sections an implementation technique clearly counts on a chosen programming language. The decision of which language to use is typically resolved upon a product instantiation. Therefore setting up and describing a reference architecture for a product line as a basis for member instantiation is one of the fundamental activities in this area.

4. FRAMEWORK FOR COMPARISON OF IMPLEMENTATION APPROACHES

In this section various approaches for coding variabilities are presented. Characteristics of each approach are described and at the end of this section an overall comparison matrix is provided.

4.1 Aggregation / Delegation

Aggregation is an object oriented technique which enables objects to virtually support any functionality by forwarding requests they can normally not satisfy to so-called delegation objects which provide the requested services. To this end “delegating” objects must:

1. hold references to the delegation objects,
2. define the operations to be delegated and implement them by solely invoking the corresponding operations of the delegation objects

Variability can be handled by means of putting the standard or mandatory functionality in the delegating object and the variant functionality in the delegation object.

This technique can work with optional features but it is rather difficult with alternative ones. In the former case there is only one indirection while in the latter case many indirections are necessary at the points of variation.

Another problem with aggregation comes up when the number of variants (e.g., of an object function) starts to grow significantly. This growth necessitates in most cases additional delegation classes and probably source files. The situation gets even worse when combinations of the already existing delegations are required.

Aggregation typically causes the variability to be resolved at compile-time since the delegation classes and the indirections are defined upon compilation. Link-time resolution is however possible if indirections to external libraries are used. Runtime resolution can be achieved in conjunction with dynamic class loading and dynamic link libraries (see below). Update-time is also possible. Imagine an update utility replacing a delegation class file.

4.2 Inheritance

Inheritance can be utilized to assign base functionality to superclasses and extensions to subclasses. Inheritance is amenable to the following categories:

- Standard (class-based) Inheritance: A subclass is derived from one superclass and may introduce new attributes and operations or overwrite or wrap existing ones.
- Virtual Inheritance: Like standard inheritance except that virtual class members can be defined in the superclass and replaced dynamically in the subclass.
- Multiple Inheritance: A subclass derives properties from many superclasses.
- Mixin-based Inheritance: Mixins are similar to ordinary classes but they do not adhere to an inheritance hierarchy. They only define differences to existing classes. Therefore they cannot be instantiated. Mixins can be combined with existing classes in order to extend their functionality. Mixins are supported by the Ada programming language.
- Object-based Inheritance: Inheritance is shifted to the level of objects instead of classes. Object-based inheritance is widely used with Smalltalk where an object refers to itself with the variable self. Depending on the hierarchical level the variable is set to the right object. This is done by rebinding the self variable each time a message is sent.
- Parameterized Inheritance: The superclass is a parameter which is set to a defined class upon instantiation.

Separation of variabilities into derived classes can be achieved with all types of inheritance. However this means that the growth of the amount of different variabilities carries along automatically the growth of the amount of subclasses which in many cases leads to a complex, unclear inheritance tree.

Multiple options in terms of selecting many optional features of an object are also difficult to manage since they require multiple or mixin-based inheritance not being supported by many languages and bringing in new problems like name clashing (e.g., when two or more ancestor classes have the same method).

4.3 Parameterization

The idea of parameterized programming is to represent reusable software as a library of parameterized components. Component behavior is determined by the values parameters are being set to.

Parameterization avoids code replication by centralizing design decisions around a set of variables. It can be applied to make data types or object classes flexible. A typical example is the parameterized class “Stack” that holds a stack of elements the type of which can be set through a parameter. This becomes even more interesting when the actions to be performed are figured out at runtime depending on the values set (dynamic parameterization).

Parameterization can enhance reusability in a product line and also ease traceability to design decisions. However centralizing code by defining parameters is often a very complex task if not impossible. The strength of a parameter, that is the set of entities a parameter can embody, is a major factor that depends on the selected programming language.

All types of variability timing are supported by parameterization.

4.4 Overloading

Overloading means reusing an existing name, but using it to operate on different types [10]. This name or symbol can be assigned to functions, procedures or operators.

Overloading promotes code reuse under certain circumstances (e.g., if some type declarations of a code segment change overloading can assure that the affected places in the code will not produce invalid type errors). In spite that programs using overloading are error-prone because they are hard to understand, they produce ambiguities and they often mislead the programmers to reuse names for improper actions.

Overloading occurs at compile time. Other timings are considerable, we are not aware though of comparable implementations.

4.5 (Delphi) Properties

Properties in Delphi are attributes of an object. A property associates specific actions with reading or modifying its data. Properties provide control over access to an object’s attributes, and they allow attributes to be computed.

A special kind of delegation is achieved when properties are used to delegate implementation of an interface to a property in the class that claims to implement the interface. In this case the property refers to a class really implementing the interface. The selection of the referenced class can be done at runtime depending for instance on the user's profile.

4.6 Dynamic Class Loading

Dynamic class loading is a standard in Java where all classes are loaded into memory as soon as they are needed. The standard way the Java runtime accomplishes this can be extended and controlled in order to address additional issues like security while loading a class.

Dynamic Class Loading is interesting for a product line infrastructure because in that way a product can query its own context and that of its user, and decide at runtime which class versions to load. The traceability back to the decision model is therefore ensured.

4.7 Static Libraries

Standard static libraries contain a set of external functions that can be linked to an application after it has been compiled. The application and the library code get loaded in the same memory space. The signatures of the functions are known to the compiled code and therefore they must remain unchanged. The implementations though can change by selecting different libraries and thus providing some kind of variability support. Active libraries as presented earlier remove this static nature of "traditional" libraries.

4.8 Dynamic Link Libraries

DLL`s are libraries loaded when needed into applications at runtime. They can be useful for the selection of variant functionalities. A special case are the ActiveX controls. They are ready-to-run, parameterized components that can be loaded dynamically in a container application. The loading can be in the same process space (like ordinary DLL's) or in separated process spaces (like independent EXE's).

Separation of variability is reached by developing distinct controls. ActiveX components support code reuse since they can embrace other controls. The component-nature of an ActiveX-control implies that there are many design decisions connected to it. Therefore many of the techniques presented above may be relevant with the development of such components.

The decision of which control to load into an application is split across many others subdecisions. The traceability in this case is a subject of further investigation.

Other component models like Java Beans and CORBA components can also be similarly beneficial in a product line context.

4.9 Conditional Compilation

Conditional compilation enables control over the code segments to be included or excluded from a program compilation. Directives mark the varying locations in the code.

One major advantage of this technique is the encapsulation of multiple implementations in a single module. The desired functionality is selected by defining the appropriate conditional symbols.

Another benefit from the conditional compilation is the separation of variabilities that can be reached when include directives are used. Include's are used to insert complete source files into base files.

On the other hand conditional directives do not support recursion or any other kind of looping which makes advanced code selection algorithms impossible [8]. Besides that directives are usually intermingled in the code resulting on a lack of overview and on difficulties in determining the scope of each conditional definition.

Following the decisions related to a conditional code selection is also not easy, especially when standard preprocessors are used. A solution to that is to interrelate directives hierarchically with a central include file as a decision model at the top of the hierarchy.

Conditional compilation is accomplished at pre-compile time.

4.10 Frames

Paul Basset's Frame Technology [15] provides the means to maximize code reusability through the definition and use of adaptable entities called frames. The goal is to form hierarchical reuse assemblies of such entities. Frames are source files equipped with preprocessor-like directives which allow parents (overlying frames) to copy and adapt children (underlying frames). On top of each hierarchical frame assembly lies a corresponding specification frame which collects code from the lower frames and provides, after being processed, the ready-to-compile module or application source. The developers can select the frame assemblies specific to their needs.

A parent that wants to adapt a child must firstly copy the complete code from the child. The adaptation is achieved in terms of:

1. inserting or replacing code at predefined locations and/or
2. setting frame parameters

The hierarchical nature of each frame setting causes, just like with inheritance, a lack of overview when the structure evolves.

Separation of variability is achieved since overlying frames encompass the variants being inserted. However introducing frames in an existing product line code can fall into difficulties since code restructuring to a large extent is probably necessary.

Frame processing is performed at precompile-time, that is, the frame processor translates the files with the frame-directives into ordinary source files that can be compiled. Therefore the variability timing supported here is clearly (pre)compile-time.

4.11 Reflection

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. Contrary to the RTTI (runtime type identification) the compiler of a reflective program does not need to know about the modules to be controlled at runtime.

Reflection relates strongly to metaprogramming where objects in higher levels of abstraction (metalevels) are established to represent entities like operating systems, programming languages, processors, object models, etc. Reflection enables access to such metaobjects and therefore allows architecting flexible systems.

Reflection can be combined with dynamic class loading in order to load modules unknown until runtime, depending on the deployment context and invoke operations on these modules.

To a product line reflection is an appealing technique. Base functionality can be "reflected" and manipulated according to a configuration. Nevertheless reflective programs are from their nature difficult to understand, to debug and to maintain. Reflective techniques are therefore strongly recommended for special systems (e.g. object inspectors) but their use in other systems should be handled with care.

Manipulations through reflection are accomplished mainly at runtime. Program optimizations or transformation in general are also possible after informations during program execution are collected, namely at post-runtime. If however reflection is seen as a general metaprogramming principle, compile-time code manipulation is possible as well.

4.12 Aspect-oriented programming

Aspect Oriented Programming (AOP) [11] is a technique developed at Xerox PARC which enables the modularization of crosscutting concerns, namely aspects, as well as the integration of join points. Join points are the locations in systems that are affected by one or more cross-cutting concerns. The process of integrating join points involves describing how a cross-cutting concern affects code at one or more join points. The integration process is referred to as composition or weaving [13]. AOP supports the programmer in cleanly separating components and aspects from each other by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

A similar approach to AOP is the Subject Oriented Programming

(SOP) [12] approach that focuses on operation-level joining. That means that SOP considers operations as the majority of join points of concern [13]. SOP extends the object oriented paradigm by defining subjects as collections of classes or class fragments. The aspect decomposition is achieved by separating subject-specific code pieces from each other whereas rules determine how the later composition occurs.

Examples of system issues that can be handled with AOP are logging, synchronization or exception handling facilities. In these cases aspect actions are defined and executed dynamically based mostly on invocation events. An aspect can for instance call a specific logging facility just before an important operation which needs logging is executed.

Aspects cooperate nicely with object orientation and can be used to expand the properties of objects. An aspect can for instance introduce a new interface and the corresponding implementation into an object that was originally devoid of such a functionality.

Aspects can also be used to implement design patterns by introducing the pattern-specific code into an object or a set of objects. Some known problems with design patterns like object schizophrenia can be even circumvented with AOP [8].

System issues are often variant properties of a product line infrastructure. Basic products support basic facilities (e.g., no logging) while the improved versions provide additional facilities (e.g., logging, additional interfaces etc.). Consequently aspects can be used to ameliorate such distinctions.

With AOP mandatory functionalities can be implemented in a standard way while diversities can be encapsulated in aspects. The benefits can be accrued from the fact that aspect combinations as well as different interpretations of an aspect are easily realized. Traceability to a decision model is also promoted seeing aspects modularize resolved design decisions.

Some difficulties with AOP may come up when multiple options are needed. For example, when logging is encapsulated in an aspect, some product line members may support logging. If the logging facility must be further improved in some products, the aspect code must be replicated and a new aspect version must be produced. This could be bypassed if aspects were reusable in a hierarchical manner. AspectJ, an aspect-oriented extension to Java, provides a sort of aspect hierarchy where abstract aspects can be extended. To our knowledge the developers of AspectJ consider to change this scheme and view aspects like regular object classes.

An aspect-oriented compiler requires the source files and the involved aspects. Therefore, AOP relates to compile-time variability resolution.

If an aspect connects to a library this connection will remain and get composed together with the aspect and so resolution at link-time could be possible. Active Libraries [14] are an interesting development in that direction. They are "active" because they may adapt the linked modules according to their deployment context.

Runtime resolution with standard AOP is not possible since a program must be composed out of the involved aspects before it can run. Active libraries can change that too by optimizing aspect-oriented code at runtime.

4.13 Design Patterns

Design patterns can be exploited in a product line context since many of them indentify system aspects that can vary and provide solutions for managing the variation.

Object-oriented techniques along with parameterization are repeatedly employed for the implementation of design patterns. The code corresponding to a design pattern is in most cases intertwined with the rest of the code and spread over many components. This in conjunction with subsequent change-requests that cause the code to mutate make the traceability between pattern code and design disappear.

4.14 Approach Comparison Summary

In this section, the various approaches discussed above are succinctly compared by means of two tables. Table 3 presents their ranking according to the criteria we use, and Table 4 depicts which techniques can be supported by which programming languages.

5. .

Table 3. Comparison Matrix

	Interface					*Implementation*					*Inititalization*					*Timing*				*Other*		
	Positive	Negative	Optional	Alternative	Multioptional	Positive	Negative	Optional	Alternative	Multioptional	Positive	Negative	Optional	Alternative	Multioptional	Compile	Link	Run	Postrun	Scalability	Traceability	SoC
Aggregation / Delegation						❍	❍	❍	●	●	❍	❍	❍	●	●	❍	❍	❍	❍	●	●	❍
Aspect-oriented programming	❍	●	❍	❍	❍	❍	●	❍	❍	❍	❍	●	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍
Conditional Compilation	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍				●	●	❍
Dynamic Class Loading						❍	❍	❍	❍	❍	❍	❍	❍	❍	❍			❍		❍	❍	❍
Dynamic Link Libraries						❍		❍	❍	❍	❍		❍	❍	❍	❍		❍		❍	❍	❍
Frames	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍				●	❍	●
Inheritance	❍		❍		❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍		❍		●	❍	❍
Overloading	❍	❍		❍		❍	❍		❍		❍	❍		❍		❍				●	●	●
Parameterization						❍	❍	❍	❍	❍	❍	❍	❍	❍	❍	❍		❍	❍	●	❍	●
(Delphi) Properties						❍	❍	❍	●	●	❍	❍	❍	●	●	❍	❍	❍	❍	●	●	●
Static Libraries						❍		❍	❍	❍	❍		❍	❍	❍		❍			●	●	❍

Legend: ❍: possible; ●: ineffective / difficult; ?: questionable; blank: not possible

Table 4. Language Mapping

	C++	Delphi	Java	Smalltalk
Aggregation / Delegation	❍	❍	❍	❍
Aspect-oriented programming			❍	❍
Conditional Compilation	❍	❍	❍	❍
Dynamic Class Loading			❍	?
Dynamic Link Libraries	❍	❍	(JNI*)	❍
Frames	❍	❍	❍	❍
Inheritance	❍	❍	❍	❍
Overloading	❍	❍		❍
Parameterization	❍	❍	❍	❍
(Delphi) Properties		❍		
Static Libraries	❍	❍	❍	❍

Legend:

❍: possible ●: ineffective / difficult ?: questionable

blank: not possible * possible with Java Native Interface

5. EXAMPLES

In this section we provide selected examples of the techniques presented. The selection is based on project experience and has been necessary since a full set of instances providing practice in the application of the techniques would exceed this paper's limitations.

5.1 Aggregation / Delegation

As mentioned before applying delegation for alternative features introduces problems. Optional features can be managed easier in this case. Consider the following example: Object A defines and implements the functions A1, A2 and A3. A1 is mandatory for all product line members while A2 is optional and supported only by extended products. A3 is alternative meaning that an extended product provides an extended implementation of A3.

Delegation requires that all methods be defined. So suppose A1 is defined and implemented in all products. A2 is defined in all products but is really implemented only in the extended versions. A3 is defined in all products and implemented differently in different versions.

When A2 is invoked in the standard version an exception could be raised. The developer of the extended version could then implement A2 in a delegation class and provide one single indirection. In this way a separation of the standard from the extended code is achieved.

The body of A2 could be implemented as shown in Figure 1.

```
public void A2() {
    delegationObject.A2();
}
```

Figure 1. Code sample (delegation+optional features)

As far as traceability is concerned. Consider following entries in a decision model:

D1: Option "Functionality F2 is supported" {yes, no}

D2: Alternative "Functionality F3" {standard, extended, pro}

A traceability matrix of the following form could trace the unique indirection appearing with delegation of optional functionality back to the decision model. Matrixes are commonly used to maintain traceability information [16].

Table 5. Traceability matrix (delegation+optional features)

Decision	Function	Delegation Object
D1	A2	delegationObject

Lets see what happens in the case of A3. The following code sample illustrates an alternative body:

```
public void A3() {
    /* standard code */
    foo1();
    foo2();
    /* here comes an optional part that
    gets delegated */
    delegationObject1.foo3();
    /* standard part continues */
    foo4()
    /* another optional part */
    delegationObject2.foo5();
    /* standard part continues */
    foo6()
}
```

Figure 2. Code sample (delegation+alternative features)

Many indirections have been made necessary which is the first source of problems. The exception handling mechanism is more complicated since it must not interrupt A3's flow in case a delegation object is missing. The required information that ensures traceability is increased as shown in Table 6. In real-world examples the number of indirections needed to support such alternatives makes delegation rather unsuited to alternative features.

Table 6. Traceability matrix (delegation+alternative features)

Decision	Function	Delegation Object
D2.standard	A3	
D2.extended	A3	delegationObject1.foo3 delegationObject2.foo5

5.2 Conditional Compilation

The use of a preprocessor during conditional compilation automates the removal of parts that are not relevant for a variant. This requires that the preprocessor has access to an instance of the decision model. The following example illustrates this

```
#if D2.is yes
/* include the code */
/* for functionality F2 */
```

Figure 3. Conditional compilation and access to a decision model instance

Conditional compilation as illustrated in Figure 3 communicates directly with the decision model. Traceability information is however not easy to maintain (e.g. in a matrix) since code fragments must be referenced. Moreover standard preprocessors do not support conditions like the one used in Figure 3. Standard preprocessors include or exclude code based on the definition of conditional symbols.

Figure 4 provides an example of using conditional compilation to handle optional and alternative code. The included code can be separated from the generic body by using the #include directive. Maintenance of the separated files becomes however difficult especially when they depend on each other.

```
#ifdef F2
/* include the code
for functionality F2 */
#endif
/* include code for standard F3 */
#ifdef F3Extended
/* include code for extended version of F3 */
#else
#ifdef F3Pro
/* include code for pro version of F3 */
#else
/* use F3 standard code */
#endif
#endif
```

Figure 4. Conditional Compilation for handling variabilities

Another option is to use a central include file as a decision model as illustrated in Figure 5. This file includes decisions from other subordinate decision files that correspond to the source files. A hierarchy of decision files can be built that way.

```
FileA
#define FileA
#define PartA
/* standard code for PartA */
#include DecisionModel
/* standard code continues */
#undef PartA
/* rest of the code follows */
```

```
DecisionModelA
#ifdef PartA
#ifdef Functionality2
/* code for Functionality2 */
#endif
#endif
```

```
DecisionModel
#ifdef FileA
#include DecisionModelA
#endif
#ifdef FileB
#include DecisionModelB
#endif
```

Figure 5. Decision Model simulation with preprocessor directives

5.3 Design Patterns - Dynamic Link Libraries

The Builder Pattern is an example of a creational pattern that can be used for loading variant code at runtime. This pattern is suitable when the logic for constructing complex objects must be separated from the users of those objects and when this logic must facilitate the building of variations [18]. The structure of the pattern is depicted in Figure 6

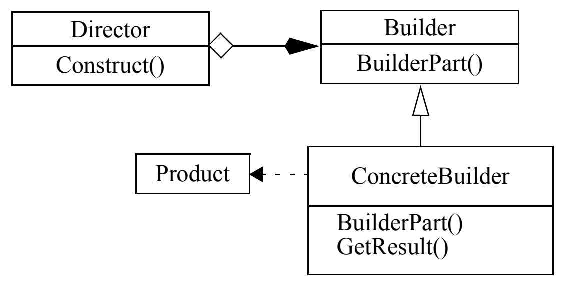

Figure 6. The Builder Pattern

using UML notation.

The Product represents the complex object to be constructed. The director coordinates the product's construction by calling the ConcreteBuilder objects. These objects implement the same interface for creating parts that is defined in the Builder abstract class. However each ConcreteBuilder builds different parts. Selecting a different variant of a ConcreteBuilder produces a different version of a part.

Dynamic Link Libraries can be combined with the builder pattern. The director chooses in this case the libraries to load depending on the desired part functionality. The loaded DLL provides the functionality for integrating the part. This may sound complex with standard DLL's, there are however special purpose DLL's like Packages in Delphi that ease up the integration [17].

This technique fits well when various parts of user interfaces are to be loaded in an application. Consider the following entry in a decision model:

D1: Option "Status Bar is supported" {yes, no}
D2: Alternative "Decoration" {standard, modern}
A traceability matrix could be easily maintained in this case (see (Table 7)

Table 7. Traceability Matrix (Builder Pattern+DLL's)

Decision	Function	Library
D1	LoadBar	Bars.dll
D2	LoadDecoration	Standard.dll, Modern.dll

6. EVALUATION

For evaluating an implementation technique criteria must be defined and a rating must be made that expresses the degree of satisfaction. For the time being we have defined criteria and accomplished a minimum evaluation as shown in Table 3. and 4. Extending the evaluation framework is a subject of our future work.

Fundamental qualities for comparing and assessing product line implementation techniques have been identified in [5]:

1. **Scope**: Scope identifies the smallest entity of tailorability supported.
2. **Flexibility**: The binding time of tailoring
3. **Efficiency**: The overhead required to support the tailorability
4. **Binary Compatibility**: The compatibility with precompiled libraries of common software

We extend the above list with the following attributes:

5. **Separation of Concerns**: Separating the variant from the standard code in a way that changes on both sides can be made effectively is important.
6. **Scalability**: The impact of code expansion
7. **Achieved Traceability**: The extent to which a tailored entity can be traced back efficiently to a decision model
8. **Ease of introduction**: Refers to the degree of system restructuring required to introduce and take advantage of a technique
9. **Language support**: Refers to the way existing programming languages can be used to apply a technique.
10. **Tool support**: The existence of tools that automate and facilitate the use of a technique is a considerable issue.
11. **Work arounds**: If introducing a technique appears difficult or impossible regarding the existing infrastructure what are the work arounds (e.g. simulation with other techniques) that can be engaged?

Another interesting aspect that can be beneficial for the evaluation is to discover primitive elements that techniques are made up of, and discuss these elements. Examples of these elements are listed below:

1. **Adding level of indirections**: Techniques like delegation add code indirections. The level these indirections occur at influences the end product. Delegation as an object oriented technique introduces indirections at the object level. Indirections at the method level are also possible. In the former case the indirection has an impact on the efficiency since a complete class must be loaded before a method can be invoked.
2. **Late binding**: Virtual inheritance uses late binding. The body of a virtual method is bound at runtime and not at compile time. Late binding influences the performance of the running program
3. **Early binding**: Type libraries are an example where early binding is used. These libraries contain statical information about methods. Clients that want to make invocations access the type libraries to obtain the required method identifiers. Early binding is generally faster than late binding
4. **Abstract Types**: Abstract classes are examples of an abstract type. Usually they only define functionality without implementing it. The implementation takes place in the subclasses. Abstraction helps creating common interfaces for accessing instances of abstract types. Common interfaces help ensure low coupling at points of variability [19].
5. **Composition mechanism**: These mechanisms can be used to implement system aspects separately and compose them to produce the end system. Such mechanisms are found in parameterization, inheritance and AOP. They influence coupling as well as system modifiability and they define the binding time of a composition [14].

7. PRACTICAL EXPERIENCE

In our project experience we came to the conclusion that:

1. deciding which approach is best suited when and
2. when to replace an approach with another are the major issues.

An example of code we have recently looked into is written in Delphi and uses exclusively preprocessor directives to handle the variabilities. The advantages and disadvantages of this approach as we described them earlier became evident. Many implementations are encapsulated in a relatively small amount of files but the overview of the variation points is a real distress.

We have tried to introduce other techniques in order to address this problem. The Delphi language restricted however our possibilities and therefore we have experimented with delegation, inheritance and frames. Other techniques like parameterization were also imaginable, they would raise though the code complexity.

The better separation of concerns achieved in some cases was balanced with the additional burden of source files. Besides, aggregation could not be applied in many cases where the delegation object was operating on itself. Multiple options are also essential and could not be managed effectively with inheritance and delegation.

Restructuring the system to a large extent was in each case a prerequisite, yet our industrial partner did not have the time nor people available to undertake such effort at this point in time. Consequently we have chosen to temporarily keep the preprocessor directives and to try to manage them better.

To this end we used include's to separate variant code. We also tried the option of having a central decision file as described in Section 5.2. This effort resulted to the separation of variant parts. However the number of directives did not decrease because code had to be broken down in numerous blocks and these blocks had to be defined. Hence the overview of the variations remained problematic. Hopefully, this decision will be revisited soon, and system restructuring will take place.

Frames have been also engaged in our experimentation. The technology showed a much better potential compared to the conditional directives. However the introduction of a frame preprocessor was necessary which had a negative impact. System restructuring was also a prerequisite in order to take full advantage of the technology.

The use of dynamic link libraries was also intended for loading parts of the graphical interface has been examined to a minimum extent. Practically we were able to present the technique and the conceived results as described in sections 4.8 and 5.3. A further evaluation is intended in our future work.

8. CONCLUSIONS

The systematic variability management at the code level is a rather immature field. Further work in this area needs to be done. This includes the refinement of the comparison framework presented here.

Migrating from systems with absent or poor variability management to systems that support it, is another concern needing to be addressed. This also involves suggestions for traversing from one technique to another.

Through our work we concluded that different approaches were needed to support different problems. No silver bullet was found. That means that techniques need to be mapped to known problems. Moreover the combination of available techniques is something that cannot be avoided. That makes research in this direction imperative.

Beyond the technical particulars cultural issues including implementation habits, company strategies and programmers' ideologies as well as the chosen implementation language can play a significant role and therefore must be taken into account.

Looking into an existing system and trying to improve it by adding variability management is an interesting aspect also for the area of software reengineering. Reverse scanners could be built that factor out the encountered variability, visualize it and help towards improvement. Future work in this field is considerable.

The set of research directions placed here could lead towards creating a complete framework that enables capturing all kinds of product line variability, tracing it back to design and architecture and composing it according to valid configurations. Scalability issues as well as supporting and combining various techniques would definitely be a concern in such an environment.

9. ACKNOWLEDGMENTS

We would like to thank all members of the Software Product Line Department at the Fraunhofer IESE for their precious feedback.

The work reported in this paper has been partially funded by the ESAPS project (Eureka Σ! 2023 Programme, ITEA project 99005).

10. REFERENCES

[1] Michalis Anastasopoulos, Joachim Bayer, Oliver Flege and Cristina Gacek, A Process for Product Line Architecture Creation and Evaluation PuLSE-DSSA – Version 2.0, Fraunhofer IESE Report No. 038.00/E, June 2000

[2] Cristina Gacek and Anton Vukovic, "Vital: Representing Software Reference Architectures," in Proceedings of the Fourth International Software Architecture Workshop (ISAW-4), Limerick, Ireland, pp. 105-110, June 2000.

[3] Joachim Bayer, Cristina Gacek, Dirk Muthig and Tanya Widen, "PuLSE-I: Deriving Instances from a Product Line Infrastructure," in Proceedings of the Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000), Edinburgh, Scotland, pp. 237-245, April 2000.

[4] James Coplien, Multi-Paradigm Design for C++, Addison Wesley, 1995

[5] David C. Sharp, "Containing and Facilitating Change Via Object Oriented Tailoring Techniques," to appear in Proceedings of The First Software Product Line Conference Denver, Colorado, August, 2000

[6] Mikael Svahnberg, Variability in Evolving Software Product Lines, Licentiate thesis, Blekinge Institute of Technology, Department of Software Engineering and Computer Science, Karlskrona, Sweden, 2000

[7] Mira Mezini, Variational Object Oriented Programming, Ph.D. Dissertation, University of Siegen, Germany, 1997

[8] Krzystof Czarnecki and Ulrich W. Eisenecker, Generative Programming Methods, Tools and Applications, Addison-Wesley, 2000

[9] Oxford University Computing Laboratory, Programming Tools Group, Intentional Programming Project (http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/intentional.htm)

[10] Peter Van der Linden, Expert C Programming, Deep C Secrets, Prentice Hall, 1994

[11] Gregor Kiczales et al, "Aspect Oriented Programming", Springer-Verlag, 1997, available under http://www.parc.xerox.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/

[12] Homepage of the Subject-oriented Research Project, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, see http://www.research.ibm.com/sop

[13] Harold Ossher and Peri Tarr, "Operation-Level Composition: A Case in (Join) Point," in Proceedings of ECOOP 1998 workshop on Aspect-Oriented Programming, Finland, pp 116-120, 1998

[14] K. Czarnecky, U.W. Eisenecker, R. Glück, D. Vandevoorde and T. Veldhuizen, "Generative Programming and Active Libraries", to appear in Proceedings of the Dagstuhl Seminar 98171 on Generic Programming, Schloß Dagstuhl, Germany, April 26-May 5, 1998, LNCS, Springer-Verlag, Berlin and Heidelberg, Germany, 1999, see http://www.prakinf.tu-ilmenau.de/~czarn/dagstuhl99

[15] Paul G. Basset, Framing Software Reuse, Yourdon Press Computing Series, 1997

[16] Karl E. Wiegers, Software Requirements, Microsoft Press, 1999

[17] Xavier Pacheco, The Builder Pattern, available under http://www.delphimag.com

[18] Erich Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[19] Oliver Lewis, Performance Issues of Variability Design in Embedded System Application Families, Ph.D. Dissertation, November, 2000, available under http://www.dcs.napier.ac.uk/~bill/oli/