

# The Use of Feature Modelling in Component Evolution

Kathrin Berg, John Müller, Judith Bishop  
University of Pretoria  
{kberg, jmuller, jbishop}@cs.up.ac.za

Jay van Zyl  
SystemicLogic Research Institute  
jay@systemiclogic.com

## Abstract

*In order to gain a competitive advantage in today's software engineering industry, it is necessary to reduce the time to market and the cost of development. It is essential to build systems that are flexible and adaptable to changing requirements. Reuse-driven development is a commonly used method for achieving these goals. Various software development approaches exist that aim at achieving reuse by determining the commonality and variability of a system family. Reuse alone, however, is not sufficient for enabling component evolution. A software system must be carefully designed to be maintainable and evolvable throughout its lifetime. This paper introduces and discusses domain analysis and the application of feature models in the context of product families as a solution to developing flexible systems that can accommodate change. Accompanied by a small-scale example, the concepts are illustrated in order to give a clear presentation of our case.*

## 1. Introduction

The software industry so often produces software systems that are built without the ability to incorporate changing and/or new requirements. The end users of these systems are becoming more and more demanding on a daily basis requiring software of high quality to be developed in shorter periods of time and within a reduced budget, that are flexible and adaptable to changes to address these demands. With technology persistently advancing and information technology demands constantly changing, the only certain factor in such environments is that change will occur. It is therefore important to take these changes into consideration when developing software systems. Accomplishing these goals using traditional development methods has in the past been unsuccessful and is difficult to achieve.

The focus is shifting from developing single systems to system families. A system family is a collection of systems sharing a common set of features that address the specific need of a defined domain [7]. System families are created to accommodate changes with less effort and to speed up the software production process. *Product line practice* is a popular approach used for developing system families. It helps in designing reusable components and in defining a common architecture for a system family. *Reuse* is a concept that has become well-known to all software engineers, especially those developing system families. Reusable components are not only used in product line development, but also other development approaches such as generative programming and aspect-oriented programming. These approaches exploit reusable components for the development of system families.

Although components are reusable and can be used as a common base for similar systems, they do not accommodate changes that might occur after deployment. Changing requirements and customizations enforce the need for components to evolve, and thus bring about variability amongst similar systems or different versions of systems. We suggest the use of domain engineering and feature modeling to efficiently and effectively deal with component evolution. Explicitly identifying the commonalities and variabilities of a family of systems proves to have great benefits for component development.

*Domain analysis* is a method used to identify the common and variable features of a family of systems. Once these features have been identified, they are presented in a feature model to all the stakeholders (people involved in the development process of a system, e.g. users, customers, developers, managers, etc.). *Feature modelling* is an essential part of domain analysis, and is used to document the commonality and variability of a family of systems. By mapping components to the features in a feature model, it is possible to incorporate changes in systems as they evolve. The above mentioned approaches were

considered, and by applying them to a case study, some of the problems were solved.

Section 2 of this document introduces the concept of *component evolution*, why changes are difficult to incorporate into existing systems, and the approach we use to deal with this problem. Section 3 describes features and feature models, and how they provide a foundation for changing system environments. Section 4 presents and illustrates a case study that deals with changing requirements by applying domain analysis and using features models.

## 2. Component Evolution

The software development industry demands software systems to adapt and evolve in changing environments, in order to be successful and gain a competitive advantage in relevant markets. The post-deployment phase of components in a software system consists mainly of maintenance, management and evolution - in essence, extending the lifetime of a component as long as possible, justifying its functional and financial level of success.

In traditional software engineering methods, such as the Rational Unified Process, little or no provision is made to accommodate change and evolution in systems and components after they have been deployed. Due to limited resources, be it financial or time-related, required changes are only considered once they become necessary. In many instances, changing requirements by evolving components involve a lot of effort. To evolve a component, means to alter its functionality. This can be done by either redesigning, altering or wrapping components, depending on the long- or short term need for change. Another option to evolve a component, would be replacing it with another component that offers the added or altered functionality. Finding a component that perfectly conforms to the current component infrastructure and can be redeployed in the place of the previous component is a difficult task. Occasionally, it is necessary to restructure and redesign the component architecture in order to incorporate changes in a system [15].

We have questioned the considerations for enabling component evolution and investigated some common component-based definitions as mentioned below.

A software component is “*a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard*” [4]. According to this definition, the elements that should allow evolution are the component model,

which “*defines specific interaction and composition standards*”, and the software component infrastructure into which the component is deployed. The component infrastructure is “*a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications*” [4]. Furthermore, a component interface is “*an abstraction of the behaviour of a component that consists of a subset of the interactions of that component together with a set of constraints describing when they may occur*”, and finally, a composition is “*the combination of two or more software components yielding a new component behaviour at a different level of abstraction. The characteristics of the new component behaviour are determined by the components being combined and by the way they are combined*”. From the definitions of a component interface and component composition, it is clear that in combining components, the dependencies on the component interface are increased.

Component evolution can therefore be divided into two main categories. These are the evolution of a component in which its interface changes, and the evolution in which the interface does not change. Well-designed interfaces become a very important and difficult task in both cases. Component inheritance and substitutability allow for functional and non-functional evolution that essentially falls into the first category, and the impact of change is localized within a single component. Component evolution in the second category can have severe effects on a system. The solution proposed in this paper falls into the first category, thus we do not consider the second category.

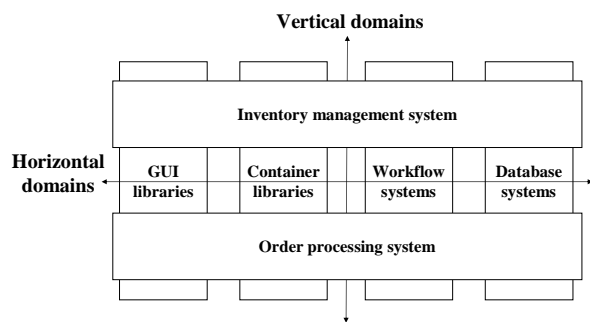
We propose the use of domain analysis and feature modelling to deal with some of the problems of component evolution as mentioned above. These will be introduced and discussed in the sections to follow.

## 3. Domain Analysis

A software system must be carefully designed to be maintainable and evolvable throughout its lifetime. Maintainability and evolvability must be considered during the early stages of analysis and design. Problems with this are that uncontrolled dependencies among components make it exceedingly difficult to modify or analyse a software system. It is critical that these dependencies among components are understood and controlled.

A domain, as defined by [5], is an area of knowledge. It includes the knowledge of how to build

software systems or parts of software systems in that area. Domain engineering is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems. It can be applied to a variety of problems, such as domain specific frameworks, component libraries and generators. Conventional software engineering concentrates on satisfying the requirements for a single system, whereas domain engineering concentrates on providing reusable solutions for a family of systems. Commonalities and variabilities need to be identified before the reusable solutions can be developed.



**Figure 1. Vertical and horizontal domains**

A family of systems can cover various domains. Vertical and horizontal domains exist. Vertical domains are areas organized around classes of systems, such as order processing systems, inventory management systems and payment systems. Horizontal domains are areas organized around classes of parts of systems. These parts of systems are classified according to their functionality, such as database systems, GUI libraries and workflow systems [5]. Reusable software, developed when applying domain engineering to vertical domains, can be instantiated to yield any concrete system in the domain. For example, a system framework covering the scope of an entire vertical domain can be developed. Applying domain engineering to horizontal domains produces reusable components. Vertical domains should be modelled using several horizontal domains, as illustrated in Figure 1.

Domain engineering consists of three process components, which are domain analysis, domain design and domain implementation. Domain analysis is the first process component that is responsible for identifying the commonalities and variabilities of a system family. It is of main concern to efficient and effective component evolution. It is a creative activity that involves collecting relevant domain information

and integrating it into a domain model. A wide variety of sources need to be used to gather sufficient domain information. Some of these sources include existing systems in the domain, domain experts, textbooks, prototyping, standards, technology forecasts, etc. [5]. A domain model is amongst other things, an explicit representation of the commonalities, variabilities and the dependencies between the variable features of the systems in a domain. One major output component of domain analysis is the feature model. By incorporating all relevant commonalities and variabilities of a domain into the feature model, features that are not currently mapped to any system requirement, but may eventually realize some future changes or customizations have already been integrated into the system's architecture and infrastructure, which are derived from the feature model during the domain design phase.

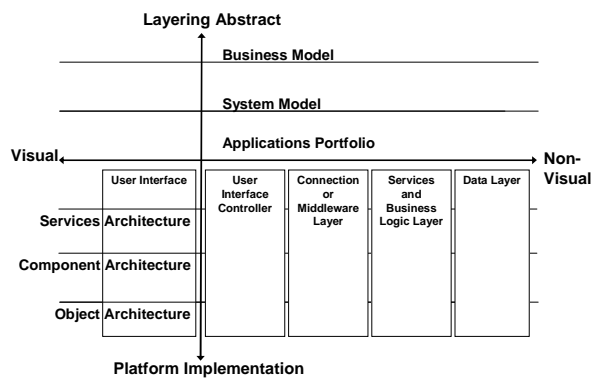
During domain design [5], the software architecture for a family of systems in the domain is developed and a production plan devised. A software architecture as defined by [1] is the structure or structures of a system, which comprise software components, the externally visible properties of those components such as the provided services and performance characteristics, and the relationships among them. The production plan describes how concrete systems will be produced from the common architecture and mapping its components to the features in the feature model. Domain Implementation [5] follows domain design and involves implementing the software architecture, its components and the production plan by using the appropriate technology.

### 3.1. Separation of Concerns

The principle of separation of concerns is known as the means for dealing with one important issue at a time [5]. It has long been used by software engineers to manage the complexity of software system development. Applying these principals to component architecture development, especially for larger systems, promises to have great benefits. The separation continuum, as is defined by as "a systemic view of a system to understand its parts and their relationship, through understanding vertical and horizontal continuums needed where abstraction and implementation are on the vertical continuum, and human and machine facing aspects to have attributes of adaptability, context and relevance are on the horizontal continuum." There are great benefits to understanding interrelated layers of a system and the levels of re-usability required in producing software intensive system families. One of the benefits is that

highly adaptive software systems can easily respond to changes in the market environment. Another benefit is that the time-to-market is decreased, since new software products can effortlessly be assembled from existing reusable assets. Operational optimization and the ability to innovate form part of the other benefits gained.

The horizontal continuum categorizes visual and non-visual aspects of a system or family of systems, and separates the user interface, the user interface controller, the connectivity, the business logic and also the data access from one another. These mostly only have an influence on the platform implementation aspects of the vertical continuum. At the one extreme of the vertical continuum are the abstract business concerns, whereas at the opposite extreme are the implementation details of the systems. The vertical continuum differentiates the layers needed to implement platform elements separately from the higher levels of abstraction needed at application levels, such as the business requirements. The levels along the vertical continuum are the business model, the system model, the applications portfolio, the services architecture, the component architecture and the object architecture. Figure 2 illustrates these concepts.



**Figure 2. The separation continuum**

Any aspect of a software system can be directly mapped onto all or at least some of the dimensions of the separation continuum. In every dimension there are different aspects to be considered and thus they have different concerns. During domain analysis the commonality and variability for these dimensions needs to be identified.

Separating the concerns of large systems families decreases the complexity significantly, but it does not help distinguish a relevant feature from one that is not. Not everything that could be a feature should be a

feature, as explained in section 3.2. For example, a user interface usually has amongst other things a background colour and a button. It is unnecessary to define the background colour or button size as a feature, unless it is a key distinctive characteristic of a product. So even though variations of background colour or of button size might exist, if it is not a significant feature, it does not have to be specifically identified. Careful consideration needs to be taken when analyzing the common and variable features of each dimension, which need to be mapped to implementation components at a later stage.

### 3.2. Features

A feature is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system. It is essentially an abstraction or product characteristic that both customers and developers understand [9]. It is important to determine what constitutes a good feature. Not everything that could be a feature should be a feature, as mentioned earlier. The unclear description of a features makes it difficult to formalize its precise semantic, validate results, and provide automated support. Feature descriptions need to be robust and expressive in order to map them to the relevant components. Features are mainly used to distinguish between choices of system properties, and not to describe any functionality in great detail [8]. They can occur at different levels, such as high-level system requirements, architectural level, subsystem and component level, and even implementation-construct level. According to [5] there are many different types of features. They can be separated into context, representation and operational features, or classified according to their binding times.

It is important to correctly identify features so that they can effortlessly be mapped onto component implementations. Do not identify unnecessary implementation details that do not distinguish systems in a domain. Developers tend to identify all implementation details and list them as features, although there are no variations among them. A feature model is not a requirements model that expresses detailed functionality. Focus should be placed on identifying system properties, factors, and assumptions that can evolve and differentiate one system from the next in the same domain. Features are generally divided into four main categories. These are capability features, operating environment features, domain technology features and implementation features [9] [10] [13], as illustrated in Figure 3.

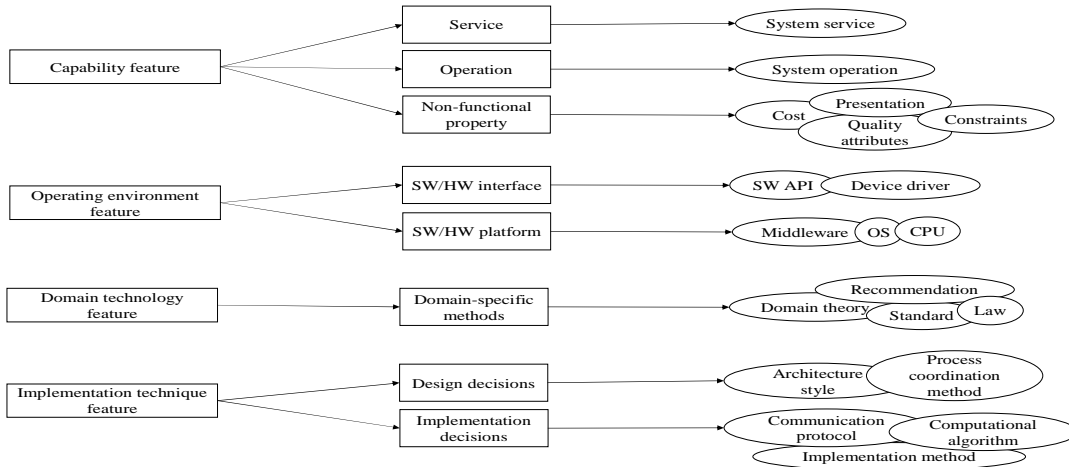


Figure 3. Feature categories (taken from [9])

#### 4. Feature Modelling

Once features are identified, they should be organized in a feature model. Feature models are used in domain analysis to capture and organize the commonalities and variabilities of systems in a domain in preparation for reuse access. The feature model represents variability in an implementation-independent way. Ideally feature models should represent requirements such as what a system must do, how it must behave, the properties it must exhibit, the qualities it must possess, and the constraints that the system and its development must satisfy [14]. They can contain features of different content at a time, for example, functional aspects, data-flow aspects, interaction aspects, synchronization aspects, etc. [5]. Feature models are usually part of other types of models used in domain engineering that together describe a reusable piece of software [12].

A feature diagram is a graphical representation of features and their relationships [6]. It has the form of a tree in which the root represents the concept being described and the following levels hierarchically denote the concept features and sub-features. A specific path down the tree represents a description of a concept instance. A sub-feature can only be chosen according to its specifier if and only if its parent-feature is included in the description of the concept instance. All features have a property that holds either a mandatory, optional, alternative or an or-specifier. A mandatory feature is compulsory and must be included in all concept instances' descriptions. The optional feature may or may not be included in a description. As shown in Figure 4, a filled circle, at the top of a node, identifies a mandatory feature, where an empty circle identifies an

optional feature. Alternative features denote a set of features of which only one feature may be chosen, where or-features denote a set of features of which at least one feature must be chosen, but more may be chosen as desired. A segment of a circle between the outer edges of a set of features denotes a choice out of a set of features. Alternative features are indicated by an empty arc, while or-features are indicated by a filled arc.

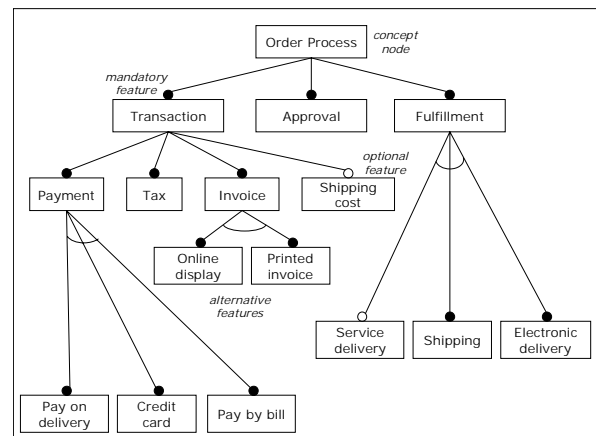


Figure 4. Example feature diagram

Composition rules capture the feature interdependencies and define which features may be combined and which may not. The two most common rules are the 'requires' and the 'mutually-exclusive' rules. The former depicts features that must be chosen together, where the latter depicts features that may not be chosen together. The composition rules should be included with the feature diagram and need to be considered.

For example, in Figure 4 the credit card feature requires the online display feature, the pay on delivery feature requires the shipping feature, and the shipping feature requires the shipping cost feature. The pay on delivery feature is mutually exclusive with the electronic delivery feature, which shows that they cannot both be selected in the same concept instance. The ‘implemented-by’ rule is represented in the feature diagram by a bold line that connects a feature with another one that implements it.

#### 4.1. Mapping Features to Components

The feature-oriented reuse method (FORM) [11] is an extension of feature-oriented design and analysis (FODA) that includes architecture design and object-oriented component development and assists in developing reusable and adaptable artefacts from product features [9].

As illustrated in Figure 5, FORM begins with feature modelling, where the resulting feature model serves as the basis for reusable and adaptable artefacts. During the architecture design activity, features are allocated to architectural components and the dependencies between them are specified. The functional architecture, constituting the architectural components, is refined into process and deployment architectures, which are used during component design.

### 5. The Views Case Study

Views [3] is a vendor independent extendable windowing system developed as a joint project between the University of Pretoria and the University of Victoria. This system was originally intended to be a small scale tool, used by beginner programmers to develop graphical user interfaces in the C# programming language without difficulty [2]. The system has since increased in popularity, and has

evolved into a rather complex system with the opportunity to expand its functionality. Realizing that the system development was unstructured and not documented, we investigated ways to organize future development in an efficient and effective manner. Domain analysis was used to explicitly identify the commonalities and variabilities of the relevant domain and model them in a feature model. By redesigning Views into a systems family, based on the common and variable features that are mapped to architectural and deployment components, it becomes flexibly, and adaptable to changing requirements, and can thus satisfy a greater market need.

Views defines XML tags and attributes which mirror those in the Windows Forms namespace of the .NET platform. It generates a graphical user interface for any program by sending the XML as a string parameter to the Views constructor. After the GUI is drawn, the user can interact with it via a small set of methods such as GetControl, GetText and SetValue.

The emphasis is entirely on the ease of use. Some of the advantages of Views are its small footprint, its user friendliness, and the ability to develop GUIs effortlessly. One distribution of Views runs on Windows, and requires access to the WinForms API provided by the System.Windows.Forms namespace. Views can also be run on the SSCLI/Rotor platform without requiring access to WinForms. Instead, it can use Tcl/Tk or QT to render the GUIs, achieving platform independence. Primarily, Views was to be developed in the C# programming language, but the option of using Java as programming language has been incorporated into the system family.

The root of the feature diagram represents the concept “Views”. Some of the main service and operational features identified for the system are GUI description notation, XML converter, Control constructor, Layout designer and GUI event model.

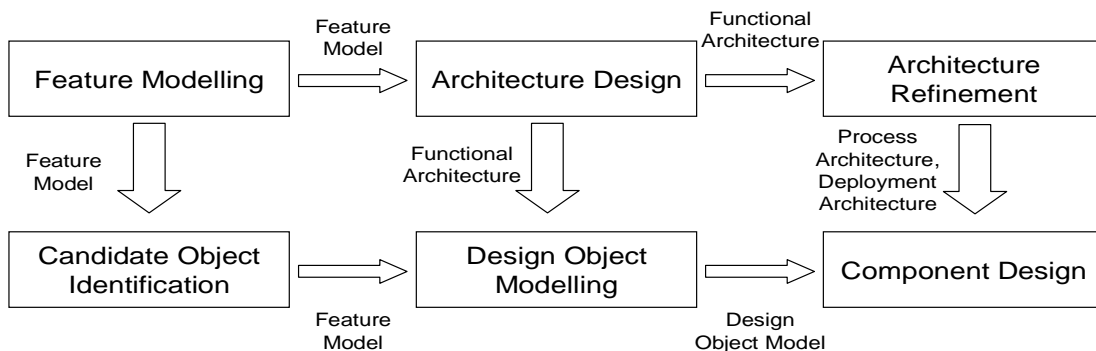
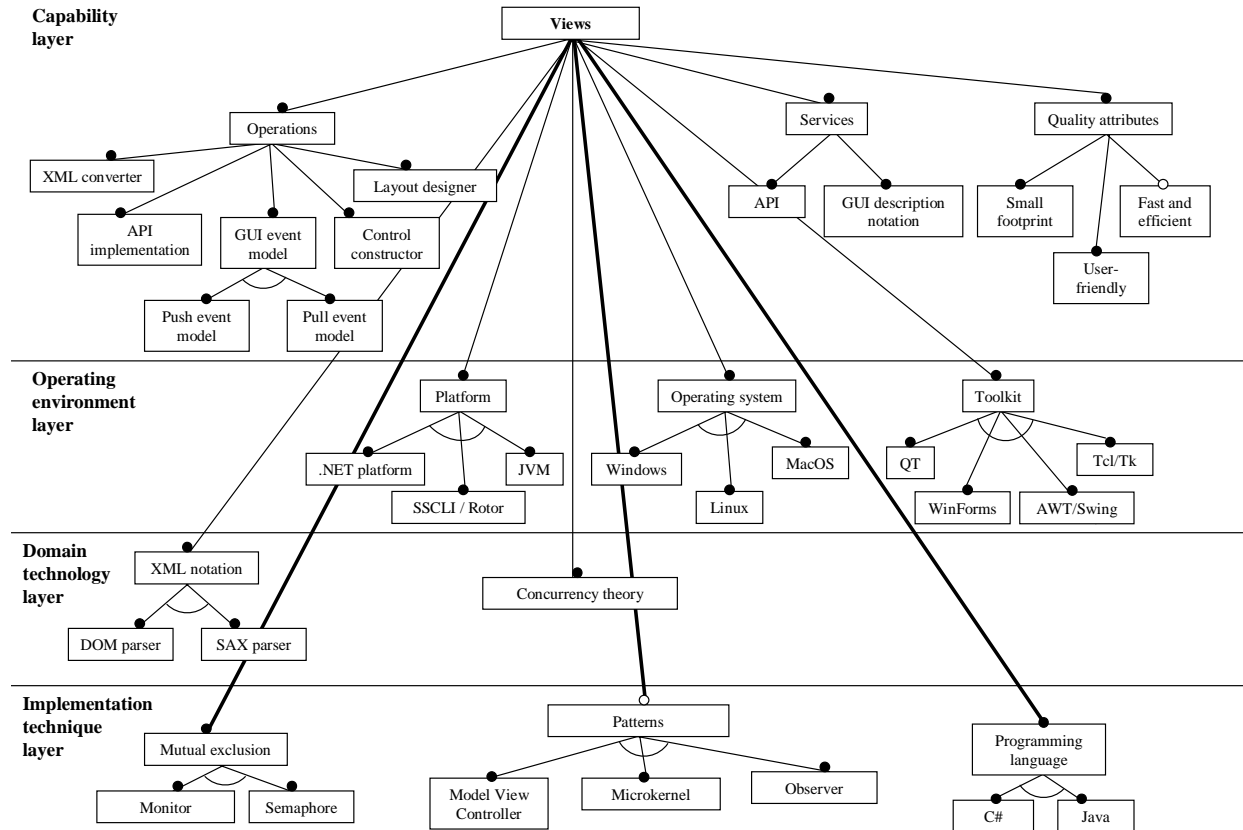


Figure 5. Asset development activities of FORM (taken from [11])



**Composition Rules:**

- Java requires JVM. JVM requires AWT/Swing.
- C# is mutually exclusive with JVM.
- .NET platform is mutually exclusive with Linux.
- .NET platform is mutually exclusive with AWT/Swing.
- SSCLI/Rotor is mutually exclusive with AWT/Swing.
- SSCLI/Rotor is mutually exclusive with WinForms.

**Figure 6. Feature model of the Views system**

Non-functional features of the system are user-friendly, fast and efficient and small footprint. XML notation, Platform, Toolkit, Operating system and Programming language are some of the other important features identified.

Suppose a user would like the Views system running on the Linux operating system. The feature model in Figure 6 would assist in presenting the configuration options for such a system. The user would have the option of choosing, for example, Linux as the operating system, C# as the programming language, Rotor as the platform and QT as the toolkit for the Views system. However, he could not choose, for example, Linux as the operating system, C# as the programming language, JVM as the platform and WinForms as the toolkit. For

this specific reason, the composition rules need to be considered before choosing a configuration for the system.

Various versions of the Views system can be customized by choosing one of the configuration possibilities in the feature model. The system can be built for different platforms, running on different operating systems, and using a different toolkit for the GUI. There are a few inter-feature-dependencies that are described by the composition rules included in the feature model. These composition rules directly relate to the composition rules for deploying the components. The feature model in Figure 6 illustrates the features that are of importance to the Views system.

## 6. Conclusions

During analysis of the Views domain, we have come across various issues. For example, some features that are not predictable at initial design time cannot be included in the feature model beforehand. It would therefore be difficult to take these into account at customization. However, due to the domain analysis and feature modelling before component design, components can be developed in such a way, that allows later changes to be effortlessly made, therefore reducing the time and costs of evolving components.

## 7. References

- [1] Bass, L., Clements, P., Kazman, R., *Software Architecture in Practice*, Addison-Wesley, December 30, 1997, ISBN 0-321-15495-9.
- [2] Bishop, J., Horspool, N., “Developing Principles of GUI Programming Using Views”, *ACM-SIGCSE*, Wiley, Norfolk, USA, March 03, 2004.
- [3] Bishop, J., Horspool, N., Worrall, B., “Experience with integrating Java with C# and .NET”, *Concurrency Practice and Experience*, to appear, 2005.
- [4] Councill, B., Heineman, G. T., “Definition of a Software Component and Its Elements”, Ch 1 of *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, June 8, 2001, ISBN 0-201-70485-4.
- [5] Czarnecki, K., Eisenecker, U. W., *Generative Programming – Methods, Tools and Applications*, Addison-Wesley, June 6, 2000, ISBN 0-201-30977-7.
- [6] Greefhorst, D., “Separating Concerns in Software Logistics”, Advanced Separation of Concerns Workshop, OOPSLA 2000, Minneapolis, October 2000.
- [7] Griss, M. L., “Product-Line Architectures”, Ch 22 of *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, June 8, 2001, ISBN 0-201-70485-4.
- [8] Griss, M. L., Favaro, J., d’Alessandro, M., “Integrating Feature Modeling with RSEB”, *Proceedings of the 5<sup>th</sup> International Conference on Software Reuse*, IEEE Computer Society, Victoria, Canada, June 02 - 05, 1998, pp. 76-85.
- [9] Kang, K. C., Lee, J., Lee, K., “Feature Oriented Product Line Software Engineering: Principles and Guidelines” Ch.2 of *Domain Oriented Systems Development: Perspectives and Practices*, Taylor & Francis, UK, December 6, 2002, ISBN 0-415-30450-4.
- [10] Kang, K. C., Lee, J., Lee, K., “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering”, *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, IEEE Computer Society, Edinburgh, Scotland, April 03 - 07, 2000, pp. 62-77.
- [11] Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures”, *Annals of Software Engineering*, Volume 5, J. C. Baltzer AG Science Publishers, Red Bank, NJ, USA, 1998, pp. 143-168.
- [12] Kang, K. C., Cohen, S. G., Hess, J. A., Nowak, W. E., Peterson, A. S., “Feature-Oriented Domain Analysis (FODA) Feasibility Study”, CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [13] Riebisch, M., Streitferdt, D., Philippow, I., “Formal Details of Relations in Feature Models using OCL”, *Proceedings of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS’03)*, IEEE Computer Society, Huntsville, Alabama, USA, April 07 - 10, 2003, pp. 297-304.
- [14] Software Engineering Institute. A Framework for Software Product Line Practice Version 4.1, <http://www.sei.cmu.edu/plp/framework.html>, Website accessed: May 2004.
- [15] Vigder, M., “The Evolution, Maintenance, and Management of Component-Based Systems”, Ch 30 of *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, June 8, 2001, ISBN 0-201-70485-4.