Integrated Design and Process Technology, **IDPT 1999**
Printed in the United States of America, June, 2000
© 1999 Society for Design and Process Science

EVOLUTIONARY DEVELOPMENT OF FRAMEWORKS – FROM PROJECTS TO SYSTEM FAMILIES

Matthias Riebisch and Bogdan Franczyk*
Technical University of Ilmenau

***tranSIT GmbH Ilmenau**
University Essen

ABSTRACT

Object-oriented software engineering did not reach all productivity objectives expected in the beginning. A lack of methodical support results in low comprehensibility of code and documentation. Reusability was attained only in projects, in which a system family was the target. This paper examines different attempts with respect to their systematic support for development of system families. Based on Domain Analysis, there is introduced an evolutionary methodology for reaching multiple use of software engineering results. The approach starts from an existing system and offers a pragmatic and systematic way to describe common and variable parts of systems explicitly and comprehensively. Based on this descriptions, the development of systems with high adaptability and maintainability is attainable. Generative Programming is mentioned as a new software paradigm offering a way to simplify the implementation process by utilization of meta-programming.

INTRODUCTION

The object-oriented software paradigm was developed with the intent of firmly increasing the productivity of software development. Its revolutionary concepts like class and object, encapsulation, information hiding, dynamic polymorphism and late binding doubtless contributed to the increase of productivity in a number of software projects. The object-oriented software paradigm has facilitated the development of elegant and maintainable software with concepts like class libraries and frameworks. However, the analysis of several framework based projects shows that reusability is less utilized than expected. An example of a failed framework project is Taligent, in which case it was impossible to handle interdependencies between different frameworks and their specializations.

Successful projects pursue the target to develop a system family. In this way, an attempt is made to develop a class of applications in a systematic way. First, this article examines different attempts from research and practice with respect to their contribution to the systematic development of system families. An evolutionary methodology is introduced, based on domain knowledge (Wartik et al., 1992). The approach starts from usual project oriented conditions and offers a pragmatic and systematic way to the development of frameworks. In order to determine commonalities and variabilities of system families, a systematic procedure is necessary. On this basis, the development of software with broad reusability is attainable.

Generative Programming (Czarnecki, 1999) is presented as a further attempt to achieve reusability, radically turning away from conventional procedures by utilizing descriptions above the level of programming languages.

STATE OF THE ART

In its thirty-year history, software engineering has produced several software paradigms, each connected with expectations that reusability, adaptability, flexibility, control of complexity and performance of software can be attained. Practice has shown that, next to other subjective factors, the mastery of complexity and the abilities to support abstraction play a great part in the success of reuse.

OBJECT-ORIENTED PROGRAMMING

Despite of new concepts like classes and objects, encapsulation and information hiding, dynamic polymorphism and relationships, and object identity, the object-oriented software paradigm has not reached any pioneering progress in the direction of maintainability, reusability and development of system families. One of the most important reasons for this is that methods still provide insufficient support for the analysis and design of frameworks and components. Most frameworks are developed ad hoc and not really systematically. However,

the development of several applications in the same domain results in economic benefits by using the concepts of abstraction and generalization. Wirfs-Brock et al. (1990) describe the process of framework development as follows: "Good frameworks are usually the result of many design iterations and a lot of hard work".

Framework development is not sufficiently supported by object-oriented modeling languages and methods. The means of object-oriented design are appropriate for engineering of single systems. There is less support for describing concepts of domain analysis and for specifying the differences between similar systems. The software engineering process does not lead to planned development of reusable systems on a level above that of code.

In contrast to the traditional object-oriented analysis and design methods, there is a set of newer methods, such as OOram (Reenskaug et al., 1996) and Catalysis (D`Souza et al., 1998), which explicitly support modeling of frameworks and the application of design patterns. A contribution of OOram to framework modeling is the recognition that collaboration, instead of class, is the fundamental abstraction in object-oriented designs. A collaboration describes communicating objects, playing specific roles in a certain pattern. A composition of collaborations is more suitable for modeling a framework at the conceptual level than modeling it as a composition of classes. In Catalysis, the basic concepts are objects and actions. An object represents a cluster of information and functionality. Actions represent anything that happens, i.e. event, task, job, message, change of state, interaction, or activity. Catalysis places actions on an equal footing with objects, because independent design requires the careful consideration of actions and their results.

REUSABILITY USING FRAMEWORKS

In this context, a framework is an arrangement of classes for solving a task with some variants. In frameworks, the variation points are implemented by Hot Spots. Positioning of Hot Spots is done by so-called slide-in methods (Pree, 1996). The quality of frameworks is measured in terms of how the demanded variability is reached. The variability reached by Hot Spots enlarges the usability of frameworks. However, a systematic procedure for the specification of useful variation points is still missing. In the practical use of frameworks it often occurs that a needed Hot Spot is not available or not applicable. On the other hand, frameworks are also provided with unnecessary Hot Spots. Maintainability as well as clearness and understandability are reduced; effective variability is not increased.

The unsystematic procedure often leads to "Fragmentation Of Design". According to Czarnecki (1999), this is based on the fact that the implementation of design patterns (Gamma et al., 1995) is not possible in an adequate manner in common programming languages. A further reason for the limited comprehensibility of frameworks can be found in an insufficient application of the software engineering principle of "Separation Of Concerns". In frameworks, there is a mix of code for functions with code for purposes of synchronization, of distribution and of optimization. Kiczales et al. (1997) describe this situation, appearing especially in frameworks, as "Tangled Code". This results in a framework with low understandability and maintainability; it is almost impossible to adopt and, thus, is reusable in rare cases.

ASPECT ORIENTED PROGRAMMING

Aspect oriented programming (Kiczales et al., 1997) provides techniques to avoid "Tangled code" in frameworks. The aim is the separation of single aspects like synchronization, distribution etc. in the source code. The introduction of new aspects into program code, however, is connected with the refactoring of single program parts. Unfortunately, during the development of applications, coding of functions is done first. Coding of aspects is executed later, so refactoring leads to a loss of clear structure. The framework code becomes very complex.

Reusability of frameworks becomes impossible if their combination leads to overlapping of their functions. This effect appears if two frameworks implement identical aspects in different ways. Czarnecki (1999) calls such situations Object Collisions. Examples are Error Handling methods, Memory management Schemes, Synchronization Schemes etc. Again, this situation can be attributed to the ad hoc design of the involved frameworks.

An important deficiency in the framework design consists in the "Semantic Gap" between domain concepts and the used programming languages. The translation of domain concepts into programming languages causes a loss of design information. Thus the code of the frameworks is less understandable and, as a consequence, less maintainable. The transfer of the experiences contained in the framework into other implementations is often impossible. Therefore, during further development, the frameworks and the systems based on them age instead of mature.

DOMAIN ANALYSIS

Domain analysis models requirements at a level of abstraction above programming language and software architecture. It describes the application area, the so-called domain, by semantic evaluation of concepts and their connections (Neighbors, 1980). By means of this

abstraction, it is possible to omit irrelevant details during analysis and design and, thus, to master the complexity of software systems. The software designer works mainly by means of abstraction. Abstraction allows the description and the construction of multiple applicable solutions. The principle of abstraction is the most important basis of all software paradigms, even more important than the differences between them.

Commonalities and variabilities are significant for abstraction, according to Coplien (1998). Commonality and variability group abstractions with equal properties. An abstraction expresses a combination of properties. Commonality enables the implementation of properties common for different systems in the considered domain. The examination of system families in the view of commonality and variability is a main impact of domain analysis. In the conventional object-oriented analysis, abstractions are used to find objects (classes). In comparison, in domain analysis, families of abstractions are detected while modeling the application domain. Feature modeling is the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model, referred to as a feature model. Feature modeling constitutes the major contribution of domain analysis in comparison to conventional software engineering. Feature modeling helps us to avoid the situation that relevant features and variation points are not included, or that other features and variation points are included but never used.

Feature models provide an abstract, implementation independent, concise, and explicit representation of the variability contained in the software. A feature should have a concise and descriptive name. The name enriches the vocabulary for describing concepts and instances in the domain. By organizing features into feature diagrams, we actually build taxonomies. Features are primarily used in order to discriminate between instances of a system family. In this context, important characteristics of a feature are primitiveness, generality and independence. Features occur at any level, e.g. system requirements level, architectural level, subsystem and component level, and implementation level. Modeling the semantics of features requires some additional formalisms, e.g. object diagrams, interaction diagrams, state diagrams, etc.

A feature model consists of a feature diagram and some additional information such as a short semantic description of each feature. A feature diagram consists of a set of nodes, a set of directed edges and a set of edge decorations. Figure 4 shows an example. The nodes and edges form a tree. The root of a feature diagram represents a concept. The parent node of a feature node is either the concept node, or another feature or subfeature node, respectively. Feature diagrams allow us to represent concepts in a way that makes explicit the commonalties and variabilities among their instances. A common feature of a concept is present in all instances of a concept. Variability in feature diagrams is expressed by options and alternatives. Alternatives are shown by arcs. The nodes with attached variable features are called variation points (Jacobson et al., 1997).

This domain analysis based approach is not applicable where project organization demands a sequential phased model of the software development process. In these cases, the target and the provision of the budget are focused on the implementation of the single current task and not on the development of a system family. Development of similar systems is carried out independent of the previous work, even if further developments are intended. The independent process of every development cycle is represented in Figure 1 as cluster, as suggested by Henderson-Sellers et al., 1990. Essential phases of the process are requirements analysis and specification (Specif), design, implementation and integration (DesignImplem) as well as deployment and maintenance (Maintenance).

Measures for future reusability within a single cluster are to be financed only on a low volume. Ad hoc domain knowledge is not structured or not even available; a complete analysis of the domain is usually not practicable by time and cost reasons. Therefore, commonality and variability with respect to a system family can not be investigated systematically. The first development cycle (Cluster 1) implements only the current requirements (ReqSpec). Systematic measures for assuring variability are not possible. Nevertheless, if such measures are performed, they frequently prove unsuitable during later development. This is the reason why the willingness of management for future reusability measures goes down.

If there are available new requirements in a new Cluster, an attempt is made to continue the development of former results like design documents and source code (DesignDoc, SrcCode). The continuation should be more economic than development from scratch. The new requirements are compared to the former solutions at implementation level. A systematic comparison with the requirement descriptions is impossible due to their informal character. Implementing the new requirements is carried out by revision of the former solution. Lacking methodical support, this process leads to the loss of structure within the solution. Design principles are no longer recognizable. A documentation of principles with design patterns is impossible. The maturity process required for framework development does not take place. Instead of maturing the structure degenerates and the quality parameters of the solution become worse. Quality characteristics like clearness and maintainability are especially effected.

GENERATIVE PROGRAMMING

Generative programming is a novel approach to the systematic development of system families. It includes both, the development of a domain model and its implementation. The domain model defines the terminology of a domain by means of concepts and their relations. A domain model is implemented by refining the feature model, providing the base for the generator. The implementation of the domain model contains knowledge in form of domain concepts, rules of composition and relations to other domains (Eisenecker, 1997 and Czarnecki, 1999). Its transformation into an executable program is performed by a generator. Thus, problem solving with generative programming is performed on a higher level of abstraction. System families can be described at the level of domain modeling, making reusability of results possible at the same level.

Generative programming uses several techniques and couples them to a methodology. Domain-specific language techniques (Van Deursen, 1997) are used to improve clearness of program code, and to enable domain-specific optimizations and error checking. Separation of concerns is achieved by separating aspects from functional components by using aspect-oriented techniques. Configuration knowledge is used to map between the problem space and solution space. The implementation of automatic configuration often requires metaprogramming (Breymann, 1998). This technique can further be used to implement the necessary language extensions. Metaprogramming involves writing programs whose parts are related by the "about" relationship, i.e. templates in C++ or reflections in Smalltalk and Java.

Practical application of generative programming depends very much on the power and flexibility of the used generators. The generator for matrix algebra purposes developed by Neubert (1998) is an example.

SUGGESTED SOLUTION: EVOLUTIONARY DEVELOPMENT

Searching for competitive advantages, an increase of effectivity in software development is necessary. Multiple use of software fulfills this purpose. Development of system families by use of frameworks has been successful in cases where collecting experiences and generalization of requirements were conducted purposefully. Examples for successful frameworks are ET ++, SanFrancisco and Mathematica. The generalization of solutions is made on a level above the programming language, instead of reusing design documents and source code.

In the following, a methodology for the evolutionary development of system families is introduced. It starts from the conventional development process of a single system solution. During the further development of this system, the methodology guides systematically towards the discovery of commonality and variability and shows a track towards the creation of a system family.

The development cycle first considered on the track is shown in Figure 2 as cluster 1. It is performed in the conventional way and could even have a longer development history. The result is a new version of a system with documents, code etc. (DesignDoc, SrcCode).

At first, new requirements for the further development of the system (cluster 2) are recorded in a requirement specification (ReqSpec). They are compared to the requirements implemented in the predecessor system (cluster 1). Domain analysis methods are used to investigate commonalities and variabilities (Comm&Var Spec). The design of the predecessor system is analyzed to identify design decisions which are relevant for revision in cluster 2.

Discovering design decisions (DesignDecRec) is an essential step in the development track towards a system family. The investigation of commonalities requires an analysis of alternatives and motivations of former design decisions. This necessitates extensive knowledge about common solution principles.

In the implementation of a system, existing variation points can usually be easily identified. They can be implemented at different levels. Variation points in frameworks are often implemented at the code level as Hot Spots (Pree, 1996). Variabilities at requirements level can be expressed in Use Cases, as Jacobson et al. (1997) show. Templates are an example at code level in C ++. Modules modified by parameters and parameter files for configuration allow variabilities in runtime. The choice of components like DLLs are possibilities at link and installation time.

To represent solution principles comprehensibly, it is necessary to explicitly describe the design decisions. By comparison of the requirements between Cluster 1 and Cluster 2 some commonalities and variabilities can already be found. The increase in clarity leads to an improvement of structure in the solution. The description of solution principles can frequently be carried out by means of design patterns. Thus, the possibilities of generalization towards a system family increase; the effort for changes diminishes in correspondence. As a result, the quality of the solution e.g. in terms of maintainability is increased. This all leads to an evolutionary maturation during the process of development.

The next part of the development cycle of cluster 2, DesignImplem, is performed in a conventional way (see Figure 2). In this task, both the requirements of cluster 2 and the commonalities and variabilities found in cluster 1 and 2, are implemented. The system family properties in terms of feature diagrams are available for the next development cycles (cluster 3 etc.).

The experiences gathered in the use and maintenance of the systems have great significance for the development of a system family. Change Requests and other requirements of the users supply essential information concerning the further development of the system family. In addition, the changes over time within this domain are discovered. The set of available information is enlarged in terms of commonality and variability during domain analysis. An implementation of that generalization is possible during the following cycles of development.

The introduced process represents a track towards a pragmatic application of the concept of a system family demonstrating an evolutionary character. This track is going via Cluster 1 to Cluster 2 and 3, while further development and systematic completion of the commonalities and variabilities is carried out. The gray arrow in Figure 2 illustrates this.

The process of further development within every cluster takes place in an evolutionary way. The spiral model by Boehm (1988) (Figure 3) is a representation best suited for the needs of practical system development. The collection of domain specifications in the form of commonalities and variabilities is performed very similarly in an evolutionary process which is overlapped with the system development process.

During the development of a system family there are further similar processes which are dependent on the main development process, e.g. refactoring of a framework or proofing the practicability of a design decision. These cycles can be characterized using fractal extensions of the spiral model (Hesse, 1997).

The development of a system family is not only performed at concept level, but also at all other levels of the system development, such as requirements specification, design and source code. Evolutionary development takes place at each of these levels, providing two aspects: the content of the results on the one hand, and in its quality characteristics like maintainability, clearness and portability on the other hand.

Experience management is connected closely with the represented evolutionary generalization process. It plays an essential role for the further development, because a generalization in the system development is only attainable by strong interaction of the experiences of the editors. Vice versa, domain analysis also contributes to the systematic improvement of the technical know-how of the editors by providing the means of structuring and of methodical processing of their experiences.

EXAMPLE

An example of a project in the field of logistics may illustrate the introduced methodology. The objective of this project is the further development of a storage management system for automatic circular stockrooms. The system currently exists in approximately 50 different variants. The variety of software and their complexity are difficult to master; a systematic implementation as system family is required. This form of implementation is intended to serve as the basis for the development of new variants, thus permitting the acquisition of additional customers. The expansion of the *items* of storage by aggregations is discussed in the following as example for analysis, description and implementation of commonality and variability.

The recent version of the system can manage different types of *tools*, *semi products* and *products*, each with various features, and *containers*. There are several storage strategies. Figure 4 shows that part of the feature diagram. The notation used is introduced by Simos (1997) and is described in the section Domain Analysis. Storage management includes mandatorily the *concepts storage*, *storage object* and *storage strategy*. *Storage objects* can be *items* or *containers* or both together. *Items* can be *tools*, *semi products* and *products* as alternatives.

The next version of the system is planned for clinics. One of the requirements to be met by this version is to manage prepared sets of surgical instruments. This requirement leads to an expansion in the feature diagram concerning the concept *item*. Essential attributes of the concept *tool* are by now name, dimensions, and life. The sets of surgical instruments can be reflected as an aggregation of different items of the type *tool,* each described by its specific attributes.

To include the surgical instrument set, the shown taxonomy of concepts is extended by a *tool set,* which is defined as a set of *tools*. The feature diagram describes the expansion at requirement level. Use Cases describe business processes at the same level. The Use Cases for *stored input* and *stored output* are expanded to dissolve *tool sets* and put together single *tools*, initiating the corresponding business process for each *tool*. This expansion is made by variation points.

The implementation of the new feature and of the corresponding variation points in the design are shown with a class diagram (Figure 5). Here, the UML notation (Booch, et al. 1999) is used. The variation point expresses the additional specialization *tool set* and the composition relation to *tool*.

The variation points are implemented at the code level by the design pattern Strategy (Gamma et al., 1995) for variation of algorithms and by the design pattern State for state dependent variation of object behavior.

CONCLUSION

The introduced approach shows a pragmatic way to develop a system family. The methods of domain analysis

offer means of expression on a higher level of abstraction. This is used to achieve reusability on the level of requirements specification and design. This makes it possible to describe the variability and commonality of a system family. The result is an evolutionary process that leads to the maturation of a framework.

The mapping of variability and commonality from the requirements level to the design and code level is an expensive manual process, requiring comprehensive experiences and knowledge of solution principles, like design patterns.

Generative programming is a new software paradigm offering a way to simplify this implementation process. Following this approach, the application domain is first examined by means of domain analysis. During this analysis, variabilities and commonalities are specified. The necessary variabilities at code level are generated based on these specifications. Development of a system family is performed on the level of concepts and requirements. Thus the paradigm is designed to bridge the "Semantic Gap" between domain knowledge and program code. However, the paradigm has not proven its effectivity in practical use to date.

ACKNOWLEDGEMENTS

We wish to thank Wolfram Riebisch and Wilhelm Rossak for useful hints for the improvement of the paper. Thanks also to Torsten Hummel and Kelly Smith who helped to improve the english version.

REFERENCES

Boehm, B. W., 1988, “A spiral model of software development and enhancement”, Computer, May, pp. 61-72.

Booch, G., Rumbaugh, J., Jacobson, I., 1999, “The Unified Modeling Language – User's Guide”, Addison Wesley.

Breymann, U., 1998, “Designing Components with The C++ STL - A New Approach To Programming”, Addison Wesley.

Coplien, J. O., 1998, “Multi-Paradigm Design for C++”, Addison-Wesley.

Czarnecki, K., 1999, “Generative Programming”, Dissertation, TU Ilmenau.

Van Deursen, A., 1997, “Domain-Specific Languages vs. Object-Oriented Frameworks: A Financial Engineering Case Study”, STJA'97 Conference Proceedings, Technical University of Ilmenau, pp. 35-39,
Available at: http://nero.prakinf.tu-ilmenau.de/~czarn/generate/stja97/vandeursen.ps.

D`Souza, D. F., Wills, A. C., 1998, “Objects, Components, and Frameworks with UML – A Catalysis Approach”, Addison-Wesley.

Eisenecker, U., 1997, “Generative Programming with C++”, Proceedings of Modular Programming Languages, Linz, Austria, March , H. Mössenbeck, (Ed.), Springer-Verlag, Heidelberg, pp. 351-365.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley.

Henderson-Sellers, B., Edwards, J. M., 1990, “Object-oriented software systems life cycle”, CACM Vol. 33, No. 9.

Hesse, W., 1997, “From WOON to EOS: New development methods require a new software process model”, In: A. Smolyaninov, A. Shestialtynow (Eds.): “Proc. WOON '96/WOON '97, 1st and 2nd International Conference on OO Technology”, St. Petersburg, pp. 88-101.

Jacobson, I., Griss, M., Jonsson, P.,1997, “Software Reuse”, Addison Wesley.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., Irvin, J., 1997, “Aspect-Oriented Programming”, Proceedings ECOOP97 – 11th European Conference of Object-Oriented Programming, Jyväskylä, Finland, June 1997, Mehmet Aksit and Satoshi Matsuoka (Eds), LNCS 1241, Springer-Verlag.

Neighbors, J. M., 1980, “Software Construction Using Components”, Tech Report 160. Department of Information and Computer Sciences, University of California. Irvine, CA.

Neubert, T., 1998, “Anwendung von generativen Programmiertechniken am Beispiel der Matrixalgebra” Diplomarbeit, Technische Universität Chemnitz, (in German).

Pree, W., 1996, “Framework Patterns”, Sigs Publications.

Reenskaug, T., Wold, P., Lehne, O. A., 1996, “Working with Objects. The Ooram Software Engineering Method”, Manning.

Simos, M., 1997, “Organization Domain Modelling and OO Analysis and Design, Integration, New Direction”, Proceedings 3th STJA Conference (Smalltalk and Java in Industry and Education, Erfurt, September 1997), Technical University of Ilmenau, pp. 166-175.

Wirfs-Brock, Johnson, R., 1990, “Surveying Current Research in Object-Oriented Design”, Communication of the ACM, 33(9).

Wartik, S., Prieto-Diaz, R., 1992, “Criteria for Comparing Domain Analysis Approaches”, International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 3, September, pp. 403-431.

FIGURES

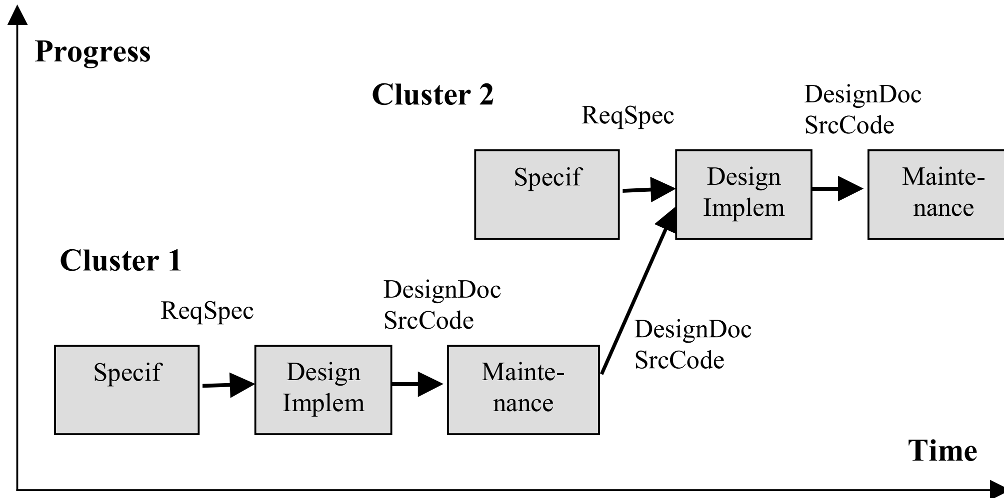

Figure 1 Conventional Process of the Further Development of a System

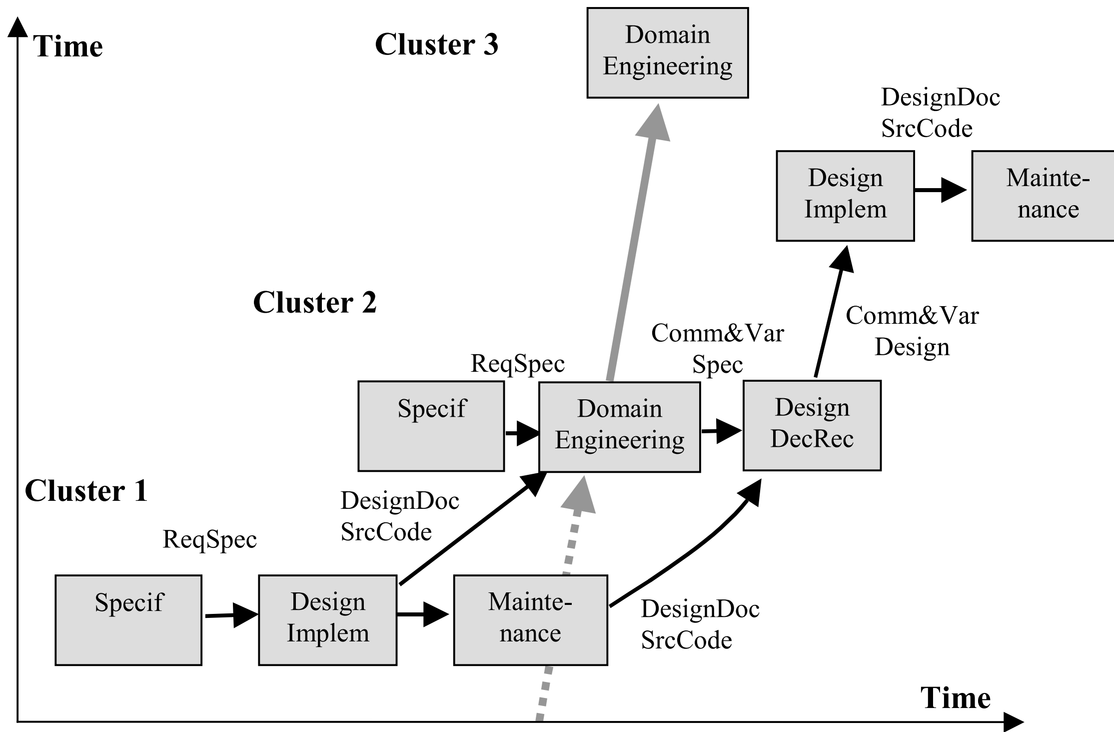

Figure 2 Evolutionary Process of development of a system family

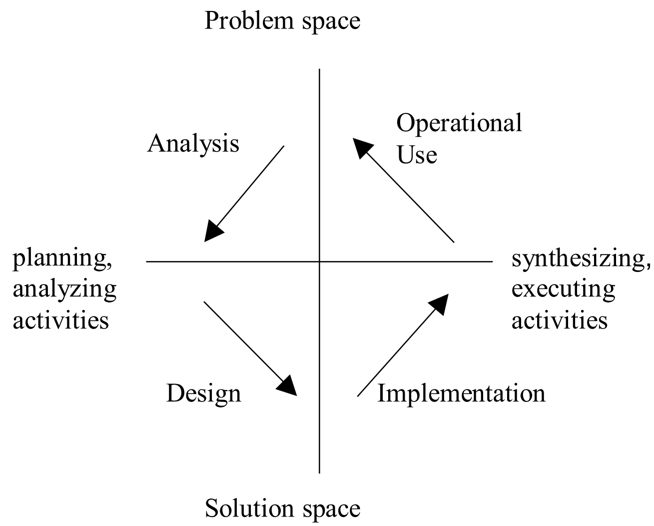

Figure 3 Phases of Generalization

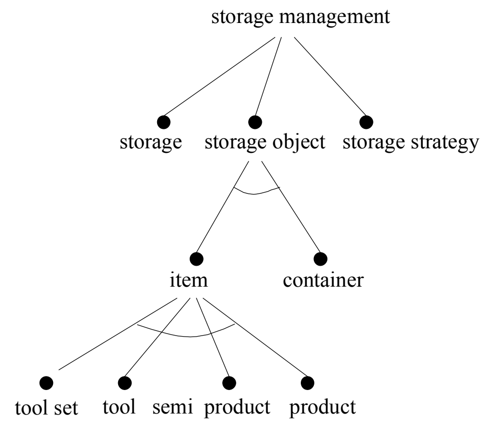

Figure 4 Feature Diagram, expanded by ToolSet

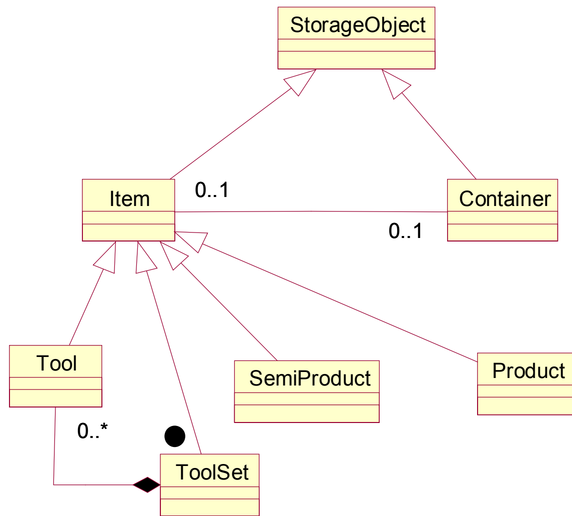

Figure 5 Instantiation of Feature Diagram