Generative Programming
Using
Frame Technology
(revised version)

by

Marco Emrich

29. December 2003

University of Applied Sciences Kaiserslautern
Department of Computer Science
and Micro-System Engineering
Department „Digital Media“

Advisor: Prof. Dr. Ulrich W. Eisenecker
Co-Advisor: Prof. Dr. Stan Jarzabek
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Such designations may be used throughout this report without being explicitly marked as trademark.

The author has taken care in the preparation of this document, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.
My sincere thanks go to following people and institutions for providing helpful information and being good partners in long lasting discussions and workshops:

- Rüdiger Schilling and Cord Giese, Delta Software Technology GmbH, and
- Dirk Muthig and Thomas Patzke, Fraunhofer Institute for Experimental Software Engineering,
- Thomas Wollny, Mathias Henss, and the team of the project “GP-WEB”,
- Christian Endler, Max Schlee, and Matthias Schönwald, University of Applied Sciences of Kaiserslautern

Special thanks go to my advisors for always being a source of positive and creative criticism:

- Ulrich W. Eisenecker, University of Applied Sciences of Kaiserslautern
- Stan Jarzabek, National University of Singapore.

Special thanks for their generous financial support and for the possibility to gain insight in their company internal technologies go to Delta Software Technology GmbH.

I especially appreciate the patience of my love Simone, my family and my friends while I was working on this document.
# Table of Contents

1 Purpose of this Diploma Thesis .................................................................................................................. 7

2 Generative Programming ................................................................................................................................. 9
  2.1 Introduction to Generative Programming ................................................................................................. 9
    2.1.1 What is Generative Programming ? .................................................................................................. 9
    2.1.2 What is a Software-System Family ? ............................................................................................. 10
    2.1.3 What is a Generator ? .................................................................................................................. 11
  2.2 Domain Engineering ................................................................................................................................. 12
    2.2.1 Introduction to Domain Engineering ............................................................................................... 12
    2.2.2 Domain Analysis ............................................................................................................................. 13
    2.2.3 Domain Design .................................................................................................................................. 14
    2.2.4 Domain Implementation .................................................................................................................. 15
    2.2.5 Interaction between Domain Engineering and Application Engineering ....................................... 16
    2.2.6 Domain Engineering Methods ........................................................................................................ 17
  2.3 Feature Modeling ......................................................................................................................................... 18
    2.3.1 Introduction to Feature Modeling ................................................................................................. 18
    2.3.2 Feature Models ............................................................................................................................... 19
    2.3.3 Feature Diagrams ............................................................................................................................. 20
    2.3.4 Specialization of Feature Diagrams ................................................................................................. 24
    2.3.5 Tools for Feature Modeling ............................................................................................................. 24

3 Frame Technology ............................................................................................................................................. 27
  3.1 Introduction to Frame Technology ............................................................................................................. 27
    3.1.1 Overview of Frame Technology ....................................................................................................... 27
    3.1.2 Frames and Slots .............................................................................................................................. 29
    3.1.3 The Organization of Frames ............................................................................................................ 30
  3.2 Different Base Concepts ............................................................................................................................... 35
    3.2.1 The Adapt Concept ......................................................................................................................... 35
      3.2.1.1 Overview ............................................................................................................................... 35
      3.2.1.2 What is the Generator in Context of the Adapt Concept ? .................................................. 36
      3.2.1.3 Export ..................................................................................................................................... 36
      3.2.1.4 Configuration of an Adapt Concept-Based Software-System ............................................. 37
      3.2.1.5 Detailed Example .................................................................................................................. 38
      3.2.1.6 Conclusion ............................................................................................................................ 42
    3.2.2 The Abstraction Concept ................................................................................................................... 43
      3.2.2.1 Overview ............................................................................................................................... 43
      3.2.2.2 What is the Generator in Context of the Abstraction Concept ? ........................................ 46
      3.2.2.3 Export ..................................................................................................................................... 47
      3.2.2.4 Configuration of an Abstraction Concept-Based Software-System .................................... 49
      3.2.2.5 Detailed Example .................................................................................................................. 51
      3.2.2.6 Conclusion ............................................................................................................................ 52

4 History of Frame Technology ......................................................................................................................... 55

5 Frame Technology in the Context of Generative Programming .................................................................... 59
  5.1 Technology Projection for Generative Programming Using Frame Technology ..................................... 59
  5.2 Using Frame Technology for Single Software-Systems ....................................................................... 62
  5.3 Using Frame Technology for Software-System Families ....................................................................... 68
Generative programming (GP) is a paradigm that becomes more and more important in modern software engineering. Along with the popularity of GP, the number of technologies GP can be projected to increases steadily. Examples for such technologies are intentional programming (IP) (see [CE00]), aspect-oriented programming (AOP) (see [CE00]; [AOSA]), static meta-programming in C++ (see [CE00]) and template language (TL) (see [Cle01]). Most of these implementation technologies are, however, not easy to handle or have other drawbacks. Static meta-programming in C++, for example, is not easy to learn and lacks features like debugging. Frame technology, on the other hand is an emerging technology with an also increasing popularity and has been labeled “easy to use” by most people who tried it. Therefore, the intention of this work was to bring “good things” together and to start research on how frame technology can be used as an implementation technology for generative programming. This has indeed proved very successful as the remainder of this work will show.

The second reason for starting an extensive study on frame technology was, that there is no other document available which addresses both existing concepts of frame technology and their differences. Therefore, besides the examination of the interaction between frame technology and generative programming, this work is also meant as a survey of frame technology and further outlines its perspectives.

This thesis has three major parts. Chapter 2, explains the essentials of generative programming. It outlines the possibilities and perspectives of generative programming, focuses on its core - the generative domain model, and introduces its driving forces – domain engineering and feature modeling.

Chapters 3 to chapter 7 are about frame technology. The two different concepts of frame technology are presented in chapter 3 - the abstraction concept and the adapt concept. The history of frames is outlined in chapter 4. Chapter 5 provides the connection between frame technology and generative programming. Chapter 6 is a survey and a comparison of available frame processors which are necessary tools for the development with frame technology. Finally chapter 7 explains how object-oriented programming (OOP) interacts with frames.

Chapter 8 and chapter 9 can be considered the last part of this thesis. They introduce techniques for frame technology which are distilled experiences from practical projects that apply frames. Chapter 8 explains thereby the theoretical foundation, while a catalog of techniques is provided in chapter 9.

Parts of this research were carried out for the PoLITe-project (see [EEE+03a]). PoLITe is a combined effort of the Frauenhofer Institute Experimental Software Engineering (IESE) and the University of Applied Sciences, Kaiserslautern, Zweibrücken. It was supported by the Stiftung Rheinland-Pfalz für Innovation, Germany (grant no. #15202-386261) (see http://www.mwwfk.rlp.de/stiftung_innovation/). Other parts of this research were supported by Delta Software Technology GmbH who also carry out research in the field of frame technology and generative programming.
2 Generative Programming

2.1 Introduction to Generative Programming

In the beginning every product was crafted by hand. This is true for industrial products and also for software-systems. The history of software development can be compared to the history of industrial revolution. During the time of industrial revolution manufacturing processes were evolved. In 1826, interchangeable parts became introduced by John Hall. These parts were used for the assembly of rifles. In 1901 Ransom Olds invented the assembly lines, which 1913 became popular due to the work of Henry Ford. In 1961 General Motors installed the first industrial robot. From thereon, we can speak of automated assembly lines, which are commonly used in the industry today. The main advantage of automated assembly lines in todays industry is, that not only many products can be produced fastly (Economy of Scale), e.g. a large amount of cars daily, but also many different variations of products (Economy of Scope), e.g. different types of cars.

The evolution of software development however, has not reached this state today. Software is still widely handcrafted. Components, which are the counterpart for interchangeable parts, appeared and became more widely known in the last 30 years of the 20th century [McI68; Ude94]. But the assembly of these parts is not automated yet. Almost every software company today develops software-system variants still manually. Components are well known and popular, but lack the efficiency to completely satisfy present day software requirements. The discrepancy between the software produced and the software needed is enormous. Generative Programming tries to close this gap by providing technologies to build the equivalent of automated assembly lines for software [CE00].

2.1.1 What is Generative Programming

"Generative Programming (GP) is 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 automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge."

- by Czarnecki & Eisenecker [CE00, p. 5].

Generative Programming (GP) is basically a paradigm, which is used to model software in such a way that its assembly can be mostly automated, ready to become automatically assembled. The core of GP is the generative domain model, which consists of three parts: problem space, configuration knowledge, and solution space [CE00, p. 5].

The problem space covers problem specific concepts and features. The solution space holds reusable components, which become assembled during the generation process. The configuration knowledge finally, contains all necessary information how elements of the solution space have to be assembled, to meet the requirements of the problem space. Therefore it can be considered some kind of mapping between problem space and solution space, but it is even more than that. Today, in classical software development, a customer specifies his requirements, which an analyst takes as input for his work. These requirements are not documented in reusable form, like a domain specific language. A human mind has to analyze and to translate these specifications. Examples for such specifications are UML diagrams and other artifacts of software design. Based on them other humans develop a software-system in a long and time consuming process. Such processes are often
not even documented and are in no way reusable or machine processable (to become automatic). This means that design and process knowledge, which describe how domain specific problems evolve to parts of the software-system become lost. Therefore GP tries to capture as much of this knowledge as possible in a formalized and computer processable form. In the generative domain model, this knowledge is called configuration knowledge [CE00, p. 5] (see Figure 2.1).

Because GP is a paradigm and not a technology itself, there are many technologies, which can be used to apply generative programming. These technologies need to be used in a special way, which means there is a projection from GP to the specific technology. Some of them are even made with the intention to be used for GP. Among others the following technology projections are available for generative programming:

- **Aspect-oriented programming with Java and AspectJ** [Mac01]
- **Frame technology** [ES02]
- **Template meta-programming in C++** [CE00]

It is important not to confuse GP with **Generic Programming** or **Automatic Programming**. Generic Programming focuses on the reuse quality of components, which is also true for GP. But in GP this is just one principle, which may be applied to the components in the solution space. GP goes farther by providing possibilities to also automatically assemble these components, the **configuration knowledge** [CE00, p. 8].

Automatic Programming has the pretension to automate the whole process. This requires artificial intelligence techniques and a huge amount of domain knowledge, even for smaller projects. GP instead distinguishes between different levels of automation, providing the possibility for the developer to choose the “right” amount of automation [CE00, p. 12].

### 2.1.2 What is a Software-System Family?

One of the main aspects of GP is to build not only single software-systems, but **software-system families** [CE00]. A software-system family is a range of products from the same problem domain [Wit96, cited from CE00]. It could be for example a suite of CAD-applications specialized for different requirements (a CAD-system for designing cars has different requirements than one for airplanes, but still shares many common features). With the application of GP it is possible to...
Develop a software-system family by including information from problem space, configuration knowledge and components and architecture of the solution space. Whenever a single system is ordered by a customer, it is not necessary to build it from scratch. Only the exact specification has to be given in the problem space to receive a new variant. This can of course be done on a smaller scope too, for building libraries or modules.

If GP is used for generating whole software-systems based on user specifications (just the requirements, not including domain specific rules and concepts) only, we speak of software factories. A software factory is the software counterpart of an automated assembly line in manufacturing. It is a metaphor adopted from the function of real factories. As soon as a software factory is completed for a specific family, it is possible to produce many different products automatically. A long time and money consuming development process relying on human handcrafting is no longer needed.

As you can see by development of software-system families instead of single software-systems, there are economical advantages. Generative Programming establishes the economies of scope [Wit96, cited from CE00] in software engineering. Now there is less time and effort needed for the development of a variety of products [CE00, p. 13]. Knowing that, the question that comes to mind is: When is the extra amount of work, which is required to build a system for producing software-systems of a family (e.g. a software factory) first, amortized by the savings mentioned above? At the present day, it is difficult to determine such a break-even point. If your intention is to build only a single system with low maintenance expected, it is perhaps not worth the extra effort. But it depends on the system. Even in a single software-system, time could be saved, if there are many redundancies, which could be covered by the use of generative tools [CE00].

It is also important to know that a difference, can be made between system families and product lines. A product line is a group of products designed for a specific group of customers [Wit96, cited from CE00]. A system family on the other hand is a group of products sharing common features specific to its domain. Therefore it is possible to use the same basic elements for different members of a system family. An example for a product line is a set of hygiene items for men: A shower gel, a perfume, and so on. A system family could be a set of shower gels with different requirements: long lasting, strong, unobtrusive, for men, for women, and so on. An example for a software product line would be a set of applications for designing cars: a CAD-system, a CAE-system, a crash simulation application, and so on. Whereas a software-system family could be different kinds of CAD-systems for specific requirements, as already mentioned above. A software product line could, however be created based on a software-system family, which bears the advantage, that the application of modern reuse technologies is possible.

2.1.3 What is a Generator?

“A generator is a program that takes a higher-level specification of a piece of software and produces its implementation. The piece of software could be a large software-system, a component, a class, a procedure, and so on.” - by Czarnecki & Eisenecker [CE00, p. 333].

In all technologies which can be used to implement generative programming there is some kind of generator involved. In GP, most of these generators are used to implement the configuration knowledge of the generative domain model. Therefore generators are important for generative programming, but are also common in other areas. CASE tools use generators to convert for example UML class diagrams to an implementation skeleton in an object-oriented language. There are GUI builders, which use a generator to produce an implementation for the graphically created user interface. Even a compiler can be considered a generator, because it takes code from a high-level programming language (like C++) and produces low-level code (like machine code) [CE00].

There are three advantages generators can provide:
2.1 Introduction to Generative Programming

- Raising the intentionality of system descriptions,
- computing an efficient implementation, and
- avoiding the library scaling problem.

“Raising the intentionality” means that the high-level code or specification (which may also be graphical) contains only the concept of what was originally intended. Details of the implementation, which would clutter the specification, making it too complex are left out. The generator provides this details. The advantages of the clean code/specification is, that it is much easier to understand, to maintain, to modify, and so on. “Computing an efficient implementation” means to let the optimizations or even the whole implementation, which can be very different from the specification in some cases, be made by the generator. “The library scaling problem” (see [Big94]) is the problem, that libraries are either big and difficult to handle, because variabilities and features are always implemented as independent components, e.g. classes, functions and so on, or inefficient in performance, because variabilities and features are well reused, providing a clear structured library, but also a slow command chain (functions/methods calling other functions/methods). With generators this problem can be avoided, because the command chains of generated libraries can be kept as short as possible. This is true, because the resolution of the command chains is already done by the generator during construction time, whereas classical libraries have to handle a lot more calls at runtime [CE00].

2.2 Domain Engineering

2.2.1 Introduction to Domain Engineering

“Domain Engineering provides the necessary foundation for Generative Programming: It focuses on system families and helps you to model the problem space, find the implementation components, and model the configuration knowledge.” - by Czarnecki & Eisenecker [CE00, p. 135].

In order to understand Domain Engineering (abbreviated DE), it is important to understand the special meaning of domain in context of this work. The term domain specifies an area for software-systems that share a large amount of common knowledge. Examples for domains are card games, financial software, CAD-systems, text editors, communication, and so on. Such domains are called business domains [AF98, p. 187] or vertical domains [CE00, p. 20]. They are opposed to technical domains [AF98] (or horizontal domains [CE00]) which focus on the functionality-bearing parts of software-systems [CE00]. Examples for technical domains are stacks, maps, search algorithms, user interfaces, database connections, and so on. In this work, by the term domain, the business domain is meant, because this type of domains offers the reuse possibilities which can be exploited by frame technology. A complete definition of domain is provided by [CE00]: “An area of knowledge that (1) is scoped to maximize the satisfaction of the requirements of its stakeholders, (2) includes a set of concepts and terminology understood by practitioners in that area, an (3) includes the knowledge of how to build software-systems (or parts of software-systems) in that area.”

As already mentioned, different software-systems of the same domain share a large amount of common knowledge. A few examples are provided here: Each card game uses cards, card piles, and terms like cards in hand or cards on table. Almost all CAD-systems use some graphical representation matters. Financial software always needs to deal with money. It is therefore quite obvious that the commonalities which are shared by software-systems of the same domain can be reused for faster software production and easier maintenance. Based on this fact, efforts were made to develop structured processes that systematically take advantage of the reuse possibilities of
domain knowledge. These processes are generally known as *Domain Engineering*. The definition for Domain Engineering from [CE00] reads: “*(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 (i.e., reusable work products), as well as providing an adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems.*” Domain Engineering is also very important for generative programming, because it allows to model problem space and configuration knowledge and assists in finding the building blocks for solution space [CE00]. It can be divided into three major parts of activity: *Domain Analysis (DA)*, *Domain Design (DD)*, and *Domain Implementation (DI)*.

### 2.2.2 Domain Analysis

Most Domain Analysis methods start with the process of finding and specifying a domain. This process is called *domain scoping*. The result of domain scoping is called *domain definition*. A domain definition specifies the focus of a domain exactly. The focus of a domain is identified by the *commonalities* and *variabilities* of the domain [Cle01]. Consider, for example, the domain of card games. The members of the family of all card games have many commonalities, like cards, draw piles, players, that players can hold a set of cards in their hands, and so on. There are also many variabilities, like which set of cards is used or when cards are allowed to play. If the analyst decides to change the variability of which set of cards is used to the communality of e.g. the common 52-cards-deck, he also changes the focus of the domain. The domain now defines the family of all card games that uses a 52-cards-deck instead of the family of all card games. It is possible to further decrease, increase, or shift the focus of the domain. A smaller domain definition allows a higher reuse due to an increased number of commonalities but also takes away a number of possible software-systems [Cle01]. A larger domain definition decreases the number of reuse possibilities while it increases the number of possible systems by adding variability. Therefore, it is very important to specify the focus of the domain as exactly as possible to gain maximum reuse without sacrificing desired software-systems.

Another important step in domain analysis is to build a *domain lexicon* [CE00] also called *domain terminology* [Cle01] or *domain dictionary*. A domain lexicon collects all important terms of the domain and provides an exact definition of these terms in context of the actual domain. It is important for the developers to have such a lexicon in order to ease communication. The terms of the domain lexicon will be used in various documents during the whole Domain Engineering process. Therefore, it is necessary to have an exact definition.

Also produced during the analysis process is a *concept model*. Concept models capture the basic knowledge and the concepts of the domain [CE00]. Concepts for the domain of card games, for example, are that players take turns, that there are several actions available including drawing or moving cards from or to another pile, and so on. Concept models can be described by various diagrams (i.e. state diagrams, ER-diagrams) or plain text [CE00].

The most important result of the Domain Analysis is the *feature model*. Feature models are used to specify requirements or components of a software-system family (and therefore a domain) and their dependencies. Feature Models are explained in detail in the next chapter.

It is, in general, a good idea to analyze at least three different, potential members of a software-system family in order to gain reliable results. This is known as the *rule of three* [But02]. These members should be as different as possible. If, for example, the domain of all card games\(^1\) have to be analyzed, good candidates for examination would be *Poker*, *Bridge* and *Eleusis* (see [Mcl02] for

---

\(^1\) Only cards games are meant which can be played with available standard decks (containing usually 32, 52, 56 or 112 cards). This does not include games that use commercially licensed or custom made cards. So, for instance, *Poker* (see [Mcl02]) is included, but *Uno* (see [HOC]) is not.
more information on these games). It would be a bad selection of candidates for this domain to choose, for instance, only the games Anaconda, Trees and Psycho. These games belong to the small domain of Draw Poker Variants\(^2\) [McI02], and are, therefore, very similar to each other.

The final result of the Domain Analysis is called domain model. It consists of the before mentioned components domain definition, domain lexicon, concept model and feature model [CE00].

### 2.2.3 Domain Design

The domain design includes a common software architecture for all systems of the domain and a production plan [CE00]. A common software architecture provides a large amount of reuse as long as it is suited for all members of the software-system family. A further assistance in finding an appropriate software architecture is to use architecture patterns as described in [BMR+96].

A production plan is necessary to describe how a specific software-system has to be assembled and how the components will be plugged in the common architecture [CE00]. The assemblage can happen rather manually or rather automatically [CE00; Coh99]. Generative Programming has the goal to do the assemblage fully automated [CE00]. This allows that a specific member of the software-system family can be ordered by a customer and the member is built (and potentially shipped) automatically without the aid of a human developer, even if this individual member has never been ordered before. For an example of such an automatic ordering system see the project GP-WEB in section 8.3 “Examined Projects“ or [BEH+02].

---

\(^2\) The family of Draw Poker Variants is a subset of the family of Poker Variants which is in turn a subset of the family of Vying Card Games which is a subset of Showdown Card Games which is a subset of Comparison Card Games [McI02]. The family of Draw Poker Variants contain only twelve known members while the domain of all card games contain more than 600 members [McI02].
2 Generative Programming

2.2.4 Domain Implementation

Domain Implementation is the final step in Domain Engineering. It includes the realization of the common architecture and the components as well as an implementation of the production plan [CE00]. It is possible to use a framework as common architecture and to develop the components for the integration in this framework with a typical component technology (see [Gri98]) while the integration of the components is done manually for each ordered member of the software-system family. GP, however, aims as already mentioned for the full automation of the integration process [CE00] by implementing an appropriate production plan. It is, in fact, even often the case that the components itself are automatically created out of a specification or at least automatically customized in order to improve the communication with other components to fit ideally into the framework or to be further specialized for its role in the architecture of a specific software-system (i.e. a member of a software-system family). This also reduces the need for glue code and usually increases performance by transferring decisions from run-time to construction time (see the optimization techniques in chapter 9 “Catalog of Techniques” for practical examples). For the full automatic realization of a production plan, a generator technology is needed. Frame technology which is the main topic of this work and will be explained in detail later on is such a technology.

---

3 This does not mean that Domain Engineering is a “waterfall”-like process applied in three separate phases. It is an iterative process with Domain Implementation as the last step of most iterations.
2.2 Domain Engineering

Figure 2.2 provides an overview of further technology projections (implementation technologies) for generative programming.

2.2.5 Interaction between Domain Engineering and Application Engineering

Domain Engineering can be applied to develop all kinds of software-systems. It is well suited for development of applications, libraries, cluster systems, frameworks, and so on. If the goal of an actual project is to develop an application, then there is an additional method needed besides Domain Engineering – *Application Engineering*. Domain Engineering usually results in analysis documents (i.e. feature models, tables, domain lexicon, and so on), a common architecture, implementation components and probably one or more generators. It, however, does not result in an application. Therefore, Application Engineering is required. Application Engineering is the general term for methods for the development of applications and includes its own processes of analysis, design and implementation if it is used for *single-system development*.[4] Fortunately, if a Domain Engineering method is used, the results of the Domain Engineering can be used to improve and shorten Application Engineering. The results of domain analysis can be included in the requirement analysis for the application. There is no need to redo the whole analysis. It can, however, happen that the actual application has new requirements that have not been identified during the domain analysis. In this case the requirements have to be included in a new iteration of the domain analysis process. The application design benefits from the common architecture for the domain while the application implementation process consists of merely integrating already implemented components of the domain. Figure 2.3 provides an overview of the interaction between Domain Engineering and Application Engineering. Usually there are several iterations of the whole engineering process. New requirements become registered and analyzed, then included in the design and finally in the implementation.

---

4 *Single system development* is the opposite of the development of software-system families. The focus is on a single system, not on a set of systems of the same domain.
Domain Engineering Methods

Domain Engineering is not a process itself. It is a general term for a number of methods following the same principles and goals (as described above). Some methods have a strong focus on the analysis part (e.g. FAST) while other methods also include a detailed process for design or implementation (i.e. DEMRAL). Not all methods include all phases and typical processes of Domain Engineering (e.g. many processes lack feature modeling). Some methods also include a suit of tools supporting them. The decision which method should be used depends heavily on the project. In some cases it might be necessary to develop a custom method based on a set of features and processes from other methods. A survey about domain engineering methods can be found in [CE00, p. 44] and [Har02]. For further information of some of the individual Domain Engineering methods a list with references is included in here:

- **Capture** [Bai93]
- **Domain Analysis and Reuse Environment (DARE)** [FPF98]
- **Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL)** [CE00; Cza98]
- **Domain-Specific Software Architecture (DSSA) Approach** [CT93; Cla94]
- **Draco** [Nei83; Nei87; Nei98]
- **Family-Oriented Abstraction Specification, and Translation (FAST)** [CW97; WL99], see also [Cle01]
2.2 Domain Engineering

- Feature-Oriented Domain Analysis (FODA) [KCH+90], see also [SEI97] and [SEI03]
- Feature-based Software Engineering Business [GFA98]
- Organization Domain Modeling (ODM) [SCK+96; Sim97b]
- Product Line Software Engineering (PuLSE) [Fra]

2.3 Feature Modeling

2.3.1 Introduction to Feature Modeling

“Feature modeling is a must if you engineer for reuse” - by Czarnecki & Eisenecker [CE00, p. 82].

Figure 2.4 Feature Diagram from FODA
(Adapted from [KCH+90])

Feature modeling is a technique that is used during Domain Analysis process (see section 2.2 “Domain Engineering”). Generative Programming relies heavily on feature modeling and would hardly be possible without. It was first introduced by the FODA method (Feature-Oriented Domain Analysis) [KCH+90] which is a well known method for Domain Analysis and was used as basis for other DA methods like ODM, DSSA and MBSE [CE00]. Together with the feature modeling process a notation for feature modeling was introduced – the feature diagram. Figure 2.4 shows a feature diagram from [KCH+90]. The notation, however, has changed since 1990. There are many different variants of the notation available at present. Examples for the different notations are described in [CSJ+92], [DB02], [DFG97] and [ML03]. In this work, the well known and widely accepted notation from [CE00] will be used. It is based on the extensions to the FODA notation of K. Czarnecki [Cza98]. There are even additional extensions available for the [CE00] notation which add notation elements for multiplicity [CBU+02; RBD+02] and attributes [CBU+02]. These are, however, not used in this work because the notation from [CE00] is sufficient. These additional extensions also add complexity to the feature diagram notation which are not necessary in this

---

5 RSEB is an abbreviation for Reuse-driven Software Engineering Business.
context. Extended notations are available in [RBD+02] and [Bed03]. It is, however, advisable to use the improved notation from [Bed03] for projects of practical relevance, because the extensions were developed to address specific issues encountered in practical experiences [CBU+02]. There is even a tool available supporting the new notation from [Bed03] – Captain Feature (see section 2.3.5 “Tools for Feature Modeling”).

2.3.2 Feature Models

The definition typically used for the term feature is the one from [Ame85]: “(A feature is) a prominent or distinctive user-visible aspect, quality or characteristic of a software-system or systems”. A more proper definition in the context of Domain Engineering is the one from [CE00]: “(A feature is) a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances”. This means that a feature is always identified in context of a specific domain (the one which is analyzed). It also means that a feature must be clearly distinguishable from other features. This allows to use features to express the commonalities and differences between members of a software-system family (or more general: between concept instances) [CE00].

The term feature model is best described by the definition from [CE00]: “A feature model represents the common and the variable features of concept instances and the dependencies between the variable features”. A feature model also contains the results of a Domain Analysis process (if the process uses feature modeling) and provides an overview of systems which can be regarded as members of a system family of a domain. A feature model consists of the features presented in a feature diagram and additional information for the diagram as well as for the features. The exact structure of a feature diagram is explained in the next section. The additional information for the feature diagram includes for instance constraints. Constraints are used to exclude invalid member software-systems from the family, which are otherwise permitted by the feature diagram (which usually already excludes many unwanted members). A feature can have many more additional information. The most important properties are listed in the following. For a complete description see [CE00].

- **Semantic Description**

  The semantic description describes what the specific feature is, and for what it is used. An example in context of the feature model of a car for the feature color could read like this: ”The color in which the visible parts of the car are lacquered”.

- **Rationale**

  The rationale explains why a feature was modeled as a feature and why it should be included in the software-system family. This information is a typical result of the domain analysis process and should be kept in the feature diagram for later reference issues. The rationale can also include a note that explains in which cases the feature should be selected during the specialization process and in which not.

- **Stakeholders**

  The stakeholder of a feature can be a customer who demanded the feature or a designer or analyst who identified and modeled the feature. Even a developer to which the implementation of the feature is assigned can be the stakeholder. It is important that there is always a contact person which can be asked about details of a specific feature. It is also advisable to note the type of stakeholder (e.g. customer, analyst, developer, and so on). Usually a list is provided that contains all stakeholders interested in the feature.

- **Exemplary Systems**
2.3 Feature Modeling

During the domain engineering process usually a few different possible systems of the domain have been analyzed. The exemplary systems identify such system that include the actual feature.

- Binding time

The binding time specifies when the feature becomes bound [CE00, Cle00]. Binding a feature means to decide if an optional feature is available or not, or in case of alternative/or-features which feature(s) is/are available. The most important time periods we can distinguish are construction time, installation time, initialization time and run-time. Construction time describes the time period during which the software-system becomes generated, assembled, compiled, and so on. Installation time refers to the period during which the software system becomes embedded in a larger software system (i.e. an operating system, a virtual runtime environment, a component container, and so on). The initialization time period means the start-up phase of the software-system. If such a phase is available, features can be bound depending on environment influences before the actual execution of the system. Run-time describes the time span from start of the execution of an actual software-system (member of a software-system family) until its end. Construction time can be further divided into generation time and compile time. With some implementation technologies (like static C++ meta-programming\(^6\)) generation and compilation happen in one process and cannot be distinguished easily. In the context of frame technology, however, these are two completely different processes (see section 3.1.1 “Overview of Frame Technology”).

2.3.3 Feature Diagrams

A feature diagram forms a tree consisting of feature nodes, edges and edge decorations. Feature nodes represent the features while the edge decorations define the semantics of the edges. A feature node is thereby presented as a box containing the name of the feature. Feature diagrams can be used on different levels of the engineering process. They can be used on the highest level, representing a whole software-system family of, for example, a CAD-system and they can also be used to represent all variabilities of an implementation class of, for example, vectors which are used in the CAD-system family. Each level between high-level specification and actual implementation is a possible field of application for feature diagrams. It is also possible to split a large feature diagram in a set of smaller ones. In this case, the root node of a sub-feature diagram matches a feature in the more general diagram. It is useful to split feature diagrams in sub-feature diagrams by using the level of abstraction as split criteria.

The root of each feature or sub-feature diagram represents the concept of the actual diagram and is therefore called concept node. This means, for example, that a concept of a high-level feature diagram that describes a whole software-system family is the name of the family (e.g. “CAD-System for Car Construction”). If the feature diagram represents the variability of an implementation class, the name of the class might be an appropriate name (e.g. “Vector”) for the concept.

Below the root node (concept node) there are feature and sub-feature nodes that represent the variabilities and commonalities of the root node (i.e. the concept). Each of these feature nodes may have its own sub-feature nodes representing its own variabilities and commonalities, and so on. The edge decorations express how the feature node depends on its sub-feature nodes.

---

\(^{6}\) See [CE00] and [EEE+03a] for more information on static meta-programming with C++.
Mandatory Features

Figure 2.5 Feature diagram with mandatory features
(Adapted from [Bed03])

Mandatory features specify which sub-features a feature must have. A mandatory is represented by a feature node that has a small filled circle on top of its box. Figure 2.5 shows a feature diagram with mandatory features. Feature A has the mandatory features B and C as sub-features while B has the mandatory feature D as sub-feature. If feature A is present, B, C and D are also present.

Optional Features

Figure 2.6 Feature diagram with optional features
(Adapted from [Bed03])

Optional features specify which sub-features a feature can have. An optional feature is represented by a feature node that has a small unfilled circle on top of its box. Figure 2.6 shows a feature diagram with optional features. The features B and C are optional sub-features of A while D and E are optional sub-features of C. If feature A is present, the features B and C may be present or may not. If feature C is present, the features D and E may be present or may not. If C is not present, its sub-features can not be included too.
2.3 Feature Modeling

Alternative Features

![Feature diagram with alternative features](image)

Figure 2.7 Feature diagram with alternative features

(Adapted from [Bed03])

Alternative features allow to choose one sub-feature out of a number of given sub-features. This matches the logical operation exclusive-OR. To specify features that are alternative to each other, an arc connects their edges. Figure 2.7 shows a feature diagram with alternative features. The feature A has the alternative sub-features B, C and D. Only one of the features B, C and D can become a legal sub-feature of A. It is neither allowed to select two or three sub-features, nor to select no sub-feature at all.

Or-Features

![Feature diagram with or-features](image)

Figure 2.8 Feature diagram with or-features

(Adapted from [Bed03])

Or-features allows to choose one or more sub-features out of a number of given sub-features. This matches the logical operation OR. To specify that features interact as or-features, a filled arc connects their edges. Figure 2.8 shows a feature diagram with or-features. The feature A has or-sub-features B, C and D. One or more of the features B, C and D can be selected as actual sub-features of A. It is, however, not allowed to select no sub-feature at all.
Figure 2.9 shows an example of a feature diagram. The concept the feature diagram depicts is the concept of a car. Therefore the name *Car* was chosen for the root node. Each car must have a car body, an engine and a transmission. Therefore, *Car body*, *Engine* and *Transmission* were modeled as mandatory features. However, not every car needs a tow-bar. It is optional and therefore modeled as optional feature in the diagram. The transmission can be automatic or manual, but not both. Hence, they are modeled as alternative features. The engine can be either electrical or (as usual) powered by gasoline. There are also a few cars that have a so called hybrid-motor which uses gasoline as well as electricity depending on the situation. There are, of course, no cars which run without an engine. So it is appropriate to model these features as or-features.

**Comparison to Domain Engineering Notations**

There are other means to represent the results of a domain analysis process. Other methods usually uses tables like the domain engineering approach FAST [CBU+02] (for more information on FAST, see [Cle01] and [CE00]). This has, however, disadvantages of the form that complex interdependencies of features can not be recognized easily [CBU+02].

Another graphical notation is to use UML class diagrams (see [OMG]) for representing feature models. The simple approach of using UML with the default meaning of its notation elements is problematic. Feature diagrams do not express how a feature is implemented, but UML does [CE00]. Feature diagrams only represent the connections and dependencies between the features, but not in which way the feature will be implemented. UML does explicitly distinguish between various ways of implementation-relevant connections like associativity, aggregation and generalization. It is, however, possible to use UML to represent a feature diagram by using stereotypes [Cla01]. Such an UML-based feature diagram is, however, not as significant and easy to handle as a native one ([CE00] notation) [CBU+02], but it allows to use UML CASE-tools for feature modeling (see section 2.3.5 “Tools for Feature Modeling”).
2.3 Specialization of Feature Diagrams

Specialization is the process of selecting variants of a software-system family. It is done by successively removing variability from a feature diagram [Bed02]. This means to dissolve groups of or-features and alternative-features and to remove optional features or change them to mandatory features. A full specialization of a feature diagram contains only mandatory features and therefore no longer any variability. It represents a specific variant of a software-system family. A partial specialization of a feature diagram still contains some variability but not the whole variability of the original feature diagram that represents the software-system family. A partially specialized feature diagram represents a (sub-)set of members of a software-system family. A configuration for a feature diagram is a full specialization or a partial specialization and represents a member of a software-system family. A partial specialization can therefore represent a sub-set of members of a software-system family, as already mentioned, or can be declared to be configuration.

2.3.5 Tools for Feature Modeling

There are three kinds of tools that can be used to create feature diagrams:

- UML CASE-tools
- Meta CASE-tools
- Dedicated feature modeling tools

As already mentioned UML-based feature diagrams have a few disadvantages [CBU+02]. The main disadvantage is that it is not possible for the tool to check, whether a feature model is valid or not [CBU+02]. Therefore, it is not very advisable to use UML CASE-tools for feature modeling. It is, however, sometimes the only option because some companies restrict the use of new tools (often for good reasons). If an UML CASE-tool has to be used, you should make sure that the additional information which can not be modeled with the tool is stored by other means (e.g. tables).
2 Generative Programming

*Meta CASE-tools* allow to define a notation and semantics on a meta level. Then they can be used on the standard level to create diagrams with such a custom defined notation. It is, of course, possible to implement the definition of feature diagrams with meta CASE-tools [CE00]. The graphical representation, however, do not exactly look like the notation for feature diagrams suggests (see Figure 2.10). Examples for meta CASE-tools are *MetaEdit+* (available from [MCC]) and *GME 3* (the successor of *GME 2000* available for free from [ISIS], for more information on *GME 3* see [BGL+01]).

*Dedicated feature modeling tools* are specifically created for feature modeling. If such a tool is available, it is probably the best choice. They render features diagrams accurately according to the notation and allow to include additional information for features.

Unfortunately, there are not many dedicated feature modeling tools available at present. One such dedicated feature modeling tool is *AmiEddi*. It is available from [CE] for free. Figure 2.11 shows the example of the feature diagram of a car in *AmiEddi*. The successor of *AmiEddi* called *Captain Feature* is currently being developed. The source code of *Captain Feature* is available from CVS of [ELM]. More information about its capabilities and its technical implementation can be found in [Bed03]. It also includes support for configuration and specialization of feature diagrams. In another project which should be mentioned here, the frame processor *ANGIE* (see section 6.1.1 “ANGIE”) is used to build parser generators using the XML-based save-files of *AmiEddi*. The generated parser checks if a specific configuration in HUTN (see [OMG]) is a valid configuration for the feature model (modeled with *AmiEddi*) which has been used by *ANGIE* for the generation of the parser. More about the *HUTN-Parser Project* can be found in [Soe02] (see also the *HUTN-Parser Project* in section 8.3).

By and large it can be said, that having a tool which stores the feature diagram in a way that a generator technology enabled tool (especially a frame processor) can further process is an advantage which should not be underestimated.
3 Frame Technology

3.1 Introduction to Frame Technology

Frame Technology can be well characterized by citations from its pioneers.

"The frame processor, its supporting utilities, and a frame repository" - definition of "Frame Technology" - by Paul G. Basset [Bas97, p. 345].

"Frame Technology works on the principle of constructing customized systems by applying changes to generic code components called frames." - by Stan Jarzabek [JS00].

"Frame Technology enables parameterized code fragments to become handled as objects, including their modeling as well as their implementation." - by Cord Giese, Delta Software Technology [DST].

3.1.1 Overview of Frame Technology

Every now and then a new idea is born. Some of these ideas die before someone tries to make use of them. Other ideas were thoughtfully evolved and structured by the creator but never adopted by others. And a few ideas are of such a quality that they become widespread. So it happened with the procedural and the object-oriented paradigm, just to name two of many. It always took years to convince people of new ideas, about 20 years in the case of object-oriented programming, but it was possible because of ambitious researchers and developers, ready to explore the unknown. Nowadays nobody can predict if that happens for a new technology, but new things have to be in use before they can be evolved. Frames are such a new technology. Frame technology can not really be called new, because the first implementations were already made in the eighties (see chapter 4 “History of Frame Technology”). It is, however, still a rather unknown technology which became more popular not until the last few years. The general interest in frames is also currently increasing. Despite the fact of being widely unknown and experimental, frame technology has already proved successful and has shown reasonable results in industrial use [Bas97] (see also the projects section of chapter 6 “Available Frame Processors”).

Frames are a technology, like object-oriented programming (OOP). In general, it is, however, not being considered a paradigm itself. Frank Sauer proposed Frame Oriented Programming (FOP) based on frame technology and called it a unification between object-oriented programming and aspect-oriented programming (AOP) [Sau02]. Basset explains how to do analysis and design for a frame based software-system [Ba97], but there are no general processes with the maturity like OOA or OOD for object-oriented programming. It is, however, possible to use frames as a technology for working with the generative paradigm instead [ES02]. All generative processes, like domain engineering or application engineering can be used with frame technology this way. In fact, there are so-called “technology projections” for generative programming using the frame technology. It is therefore possible to use frame technology for the construction of software factories (see section 2.1.2 “What is a Software-System Family ?”). Whenever software-system families have to be developed, a software factory is needed. The use of frame technology to build different customized systems is called inter-application variability. But frame technology is not limited to software factories, it is also useful for reuse inside a single software-system. Using frame technology this way, we speak of intra-application variability [CE00].
Frame technology is a specific kind of generator technology. A generator called frame processor takes a high level specification (consisting of frames, functions\textsuperscript{7} and internal configuration data – these terms will be explained later) and produces the source code for a software-system from this specification. There are already frame processors available from different vendors (see section 6.1 “Frame Processors in Detail”). These processors belong to one of the two elementary concepts: the adapt concept by Basset [Bas97] and the abstraction concept which was developed by Delta Software Technology [DST].

Figure 3.1 shows the basic principle of development with a frame processor. The specification\textsuperscript{8}, the frame processor will take, consists of an assembly of frames, functions and configuration data (e.g. an xml-based configuration file). The generated source code is target language code and can be further processed by an appropriate compiler. The result can be an application, a server, a binary component or any number of parts of a software-system. It is also possible to use a non-code language as the target language (e.g. a natural language like English, or a markup language like HTML or XML-FO). In this case no further compilation is required. The generated source code is already the final product. This is useful for the generation of non-software artifacts like documentation.

\textsuperscript{7} Functions are only available in the abstraction concept.

\textsuperscript{8} Specification in this context means the complete software specification which is used as input for the frame processor. Specification in another context can also mean external configuration data only (see Specification in the Glossary).
3 Frame Technology

3.1.2 Frames and Slots

“A frame is a component in any programming language that can be reused, not only as-is, but as adapted by other frames to fit a new application. A frame contains (1) program commands and variables and (2) frame commands and variables. The frame commands can add to or subtract from other frames’ capabilities as the application requires.”

by Paul Basset [Bas97]

‘Frames are code modules as objects” - [DST]

The textual files which are a part of the specification that a frame processor takes as input (see Figure 3.1) consist of frames. In case of an abstraction based frame processor these files also contain functions which are explained in detail later (see section 3.2.2 “The Abstraction Concept”). The frames encapsulate the target language code (C++, Java, English, and so on). The entirety of text lines which does not consist of target language code is called the frame code (this includes the functions).

The anatomy of a frame is different for each of the available frame processors (see Listing 3.1 and Listing 3.2). There are however some commonalities which will be explained later. Each frame consists of a unique identifier, called frame-name and its specification, called frame-text [Bas97]. The frame-text are lines of text that contains the code in the target language and special commands for the frame processor. These commands are called frame commands and are considered part of the frame code. There are frame commands realizing the concept of slots which is a central mechanism of frame technology. A slot is a placeholder, variation or insertion point where the original frame-text can be modified. There are different types of slots depending on the used concept. Some slots allow the modification of its content (e.g. C++ code) by replacement, addition or subtraction [JS00]. Other slots can be filled by frame variables that behave like variables of a programming language. The canonical use of this concept is to fill placeholders with specific code, like a preprocessor macro does, or like a template substitutes a data type. Therefore you can imagine a frame processor as some kind of mighty preprocessor. It can, however, do much more than just replacing values. Slots may take many different types of content. The content of a slot can be as simple as a single numerical value and as complex as another frame or in some cases even multiple frames.

Besides the embedding of slots, there are frame commands for many different tasks. One task is the manipulation of slots. In the adapt concept, there are frame commands which can insert in, delete, and replace slot content, while other slots are modified by assignments. Using the abstraction concept, the frame commands for the manipulation of all types of slots are assignments. Other commands compare values of slots or frame variables, or even iteratively insert content into slots (for more information about available commands see 6.1 “Frame Processors in Detail”). Depending on the frame processor all or nearly all commands for control flow in a procedural programming language are also available as frame commands. In contrast to the procedural commands, however, the frame commands are processed at generation-time instead of runtime.
3.1 Introduction to Frame Technology

Listing 3.1 A XVCL frame with Java code
(Adapted from [Nus02])

```java
<x-frame name="Human" language="java">
<set var="HUMAN" value="Human"/>
<brack name="HUMAN_PARAMETERS"/>
class <value-of expr="?@HUMAN?"/>
{
    String name;
    int Age;
    double weight;
    double height;
    <brack name="HUMAN-BODY">
    int Head = 1;
    int Legs = 2;
    </brack>
    <brack name="HUMAN-NEW-ATTRIBUTES"/>
}
</x-frame>
```

Listing 3.2 An ANGIE frame with C++ code
(Adapted from [Eis02a])

```cpp
#ifndef
#define
#include <iostream>
using namespace std;
struct <{ } Name <} H
#define <=Name <=H
#include <iostream>
struct <=Name <=
< { } Components <>
    <=Name <= () {
        cout <="<Name >" << endl;
    }
Coachwork cw;
< { } Motor <>
m;
< { } Gear <>
g;
< { } Towbar <>
tb;
};
#endif
End Frame
```

3.1.3 The Organization of Frames

Frames do not exist on their own. To be useful for eventually becoming part of a software-system, there must be some way to connect them. The modification of a frame is done by frame commands of other frames or of the frame itself. A frame can adapt or aggregate other frames by referencing them in its slots, sometimes controlled by commands of a third frame or - in case of the abstraction concept – a function. This way frames are linked together, forming a specific structure. This structure is called frame hierarchy [Bas97]. It can be represented by a graph with nodes (also called vertices). Each of these nodes represent a frame. If two frames are connected by an edge, it means that the super-ordinated frame adapts/aggregates the sub-ordinated one.

A frame hierarchy is, however, usually not a static structure. It builds up during the processing of frames by the frame processor. Which frames are actually aggregated or adapted is not known

---

9 The term adapt is used within the adapt concept, while aggregate is related to the abstraction concept.
10 Precisely, in the abstraction concept each node represents a frame instance instead of a frame.
3 Frame Technology

before construction time, because the slots often uses variables to specify their content rather than static names.

The graph that represents the frame hierarchy is often regarded as rooted tree. This is, however, not true, as the following discussion will prove. A rooted tree, is a (free) tree in which one of the vertices (the root) is distinguished from others [CLR90]. This is true for a frame hierarchy, because there is exactly one special vertex that is called the root. A (free) tree is a graph meeting three conditions [CLR90]:
1. It is connected.
2. It is undirected.
3. It is acyclic.

The first condition is always true. The frame hierarchy has to be connected, so that all frames are part of the system. Disconnected frames are frames with no path to the root and therefore they will not be used by the frame processor\(^\text{11}\). The second condition depends on the chosen representation. A frame hierarchy could be represented as an undirected or a directed graph as well. If a directed graph is chosen as representation, the graph is not a tree. If an undirected graph is chosen as representation, the last condition needs to be checked: Is the graph acyclic? Consider the following

\(^{11}\) Single frames are, however, possible. Regard them as a frame hierarchy consisting of the root frame only. Within the abstraction concept they can be used e.g. for storing configuration data for other frames without being connected to them (see configuration frame in 3.2.2.4 Configuration of an Abstraction Concept-Based Software-System).

Figure 3.2 Organization of frames
3.1 Introduction to Frame Technology

example: There is a frame A which is adapted/aggregated by the frames B and C, which both in turn are adapted/aggregated by a frame D. Because the graph is undirected, there is a way A->B->D->C->A (see Figure 3.3). This means the graph is not acyclic and therefore not a tree.

On the other hand it is much more appropriate to represent a frame hierarchy with a directed graph, because an adaptation/aggregation only works in one direction. In this case, the graph is also acyclic. A cycle is not allowed, otherwise there would be an infinite recursion forcing the frame processor to process until out of memory (see Figure 3.5). This means a frame hierarchy can be well represented by a directed acyclic graph (DAG).

For convenience, however, the tree is still used. This is possible by using the order (high = closer to the root, low = more distant from the root) of the tree, which we obtain by neglecting the exceptions mentioned before.

The lower a node is in the hierarchy, the lesser it is context sensitive and therefore the higher is its re-usability [WJS+01]. In a frame hierarchy representing a large software-system, the lowest frames commonly handle basic functionality like operating system code, networking protocols and so on. Higher frames in such a hierarchy consist mainly of application code. In context of the adapt concept, the highest node, the root of the tree, is a specification frame (short: SPC) [Bas97]. In the abstraction concept the highest node can be a root frame [Gie02c] or also a SPC which is considered as special type of a root frame in this context. A SPC is some kind of master blueprint for specifying the customization of its frame hierarchy [Net02a] (see Listing 3.3 for an example of an adapt concept-based SPC). The term SPC is originally derived from the adapt concept, but applies to the abstraction concept as well\(^\text{12}\) (see Listing 3.4 for an example of an abstraction concept-based SPC). The term root frame is used in relation with the abstraction concept and is explained in detail there (see 3.2.2 “The Abstraction Concept”). Please notice that the abstraction concept has also other options for configuration besides SPC and root frames (see 3.2.2.4 “Configuration of an Abstraction Concept-Based Software-System”).

\(<?xml\ version=\"1.0\"?>\)
\(<!DOCTYPE\ x-frame\ SYSTEM\ \"file://c:\xvcl_1.0_beta2\dtd\xvcl_1_0.dtd\">\)
\(<\!\--\)
\<x-frame\ name=\"KINGDOMTABLE\">
\<set-multi var=\"COLS\" value=\"COL1,COL2,COL3,COL4\"/>
\<set var=\"COL1\" value=\"KINGDOM_NAME\"/>
\<set var=\"COL1TYPE\" value=\"String\"/>
\<set var=\"COL1_LENGTH\" value=\"32\"/>
\</x-frame\>

\(12\) A SPC is rather unusual in context of the abstraction concept. In this context, it happens just coincidentally that a root frame appears in a form that it can be called SPC.
When a frame of the hierarchy is viewed independently, it forms its own sub-hierarchy, which is a subtree of the frame hierarchy's tree structure. The frame itself is the root of that sub-hierarchy [Jar01]. Therefore it can be considered a SPC in the case that it is used for configuration issues. Sub-hierarchies are also called frame layers. Frame hierarchies are separated into layers to form logical groups of software parts with different computational aspects. For example there can be a layer for a graphic-engine, a layer for a database, a layer for the user interface, a layer for the business logic and so on [WJS+01].

Multiple frame hierarchies form a framework\footnote{The term framework should not be mistaken as framework in context of the object-oriented programming. It has a totally different meaning there.} [CJ99], like a forest consists of multiple trees in graph theory [CLR90]. A framework contains all frames of the system necessary to build a customized software-system [CJ99].

Because frame hierarchies are based on the graph-theoretical concept of a DAG, the frame technology terms have corresponding ones in the graph-theoretical model (i.e. root corresponds with SPC/root frame and so on). Table 3.1 summarizes the matching terms. Therefore it is also possible to adapt the graph-theoretical terms that express the relationships between frames. The
3.1 Introduction to Frame Technology

terms used in graph theory are *ancestor*, *descendant*, *parent*, and *child*. The corresponding terms in frame technology are very similar as the following definitions will indicate.

**Ancestor**
1. *Ancestor* is a term used in the context of *graph theory*. Any other *node* on the *path* of a *node* to the *root* is called an *ancestor* of that *node* [CLR90]. That *node* is called *descendant* of the other *nodes*.

2. *Ancestors* (or *ancestor frames*) of a specific frame are all frames that are located above this specific frame in the *frame hierarchy* along its *path* to the *root*. This includes the root (see Figure 3.2). An *ancestor frame* can also be called *adapter frame* in context of the adapt concept, because it adapts other frames [Bas97, p. 99; Net02].

**Descendant**
1. *Descendant* is a term used in the context of *graph theory*. A *descendant* of a *node* is a *node* with the other *node* on its *path* to the *root* [CLR90]. The other *node* is called its *ancestor*.

2. *Descendants* (or *descendant frames*) of a specific frame are all frames that are located below this specific frame in the *frame hierarchy*. [Bas97, p. 99; Net02].

**Child**
1. *Child* is a term used in the context of *graph theory*. A *node* is called the *child* of another *node* if they are connected by an *edge* while the *path* to root of that *node* contains the other *node*. This means the child *node* is more distant from the *root* than the other [CLR90]. The other *node* is called its *parent*.

2. A *child* (or *child frame*) is the next descendant frame of a specific frame. This means the next frame below the actual one in the *frame hierarchy* [Bas97, p. 99; Net02].

**Parent**
1. *Parent* is a *term* used in the context of *graph theory*. A *node* is called the *parent* of another *node* if they are connected by an *edge* while the *path* to root of that *node* does not contain the other *node*. This means the parent *node* is closer to the *root* than the other [CLR90]. The other *node* is called its *child*.

2. A *parent* (or *parent frame*) is the next ancestor frame of a specific frame. This means the next frame above the actual one in the *frame hierarchy* along its *path* to the *root* [Bas97, p. 99; Net02].

<table>
<thead>
<tr>
<th>Graph theory term</th>
<th>Corresponding term in frame technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>frame hierarchy</td>
</tr>
<tr>
<td>node/vertex</td>
<td>frame</td>
</tr>
<tr>
<td>sub-tree</td>
<td>sub-hierarchy</td>
</tr>
<tr>
<td>root</td>
<td>SPC/root frame</td>
</tr>
<tr>
<td>leaf</td>
<td>leaf (frame)</td>
</tr>
<tr>
<td>ancestor</td>
<td>ancestor (frame)</td>
</tr>
<tr>
<td>descendant</td>
<td>descendant (frame)</td>
</tr>
<tr>
<td>child</td>
<td>child (frame)</td>
</tr>
<tr>
<td>parent</td>
<td>parent (frame)</td>
</tr>
<tr>
<td>forest</td>
<td>framework</td>
</tr>
</tbody>
</table>

*Table 3.1 Corresponding terms of graph theory and frame technology*
3 Frame Technology

Figure 3.2 provides a graphical example of the organization of frames and the corresponding terms. If used with actual frames, such a diagram is called *x-framework* [Jar03]. Please notice that three types of diagrams are used within frame technology. A *feature diagram* shows the variability of a software-system (or only of a part thereof), a *x-framework* shows the connections between frames, and the various *UML diagrams* provide views of the generated source code and of the source code of generators. There is, however, no general mapping between these diagrams, but it is possible to define rules and relations which establish such a mapping. A simple exemplary rule would be to map each root-frame/SPC in the frame graph to one class in the UML class-diagram.

3.2 Different Base Concepts

As already mentioned, there are two different concepts that frame processors are based on. These two concepts have been developed independently, but the ideas of both concepts are based on Minsky’s “A Framework for Representing Knowledge” [Mi74]. The concepts share many commonalities as explained above (see 3.1.1 “Overview of Frame Technology”), but have also many differences. The core of the first concept is the application of *break/variation point*, *adapt* and *insert* commands. Henceforth, it is called *adapt concept* in this documentation. Please notice that this is no officially agreed naming convention but simply used here for distinguishing the concepts. The other concept is based on the interaction of frames and functions operating on these frames. In this context *function* is a general term which can be further specialized in free function, script-function (also called script) or method depending on the frame processor and the context of use. The main aspect of the latter concept is the application of an abstraction mechanism similar to that of object-oriented programming. Therefore, it is called *abstraction concept* here (please notice that this is no official naming either). The main differences of these two concepts is the way how and when frames become connected. The organization of frames (see section 3.1.3) can be achieved by different means with distinct focuses.

3.2.1 The Adapt Concept

3.2.1.1 Overview

The adapt concept was invented by Basset [Bas97] and the *Netron* company. It is used by *FrameProcessor*, *Netron Fusion*, *FPL* and the *XVCL Frame Processor* [NN01; Net02-2; Sau02; Pat02]. There are different kinds of slots. One kind of slot inserts a frame variable in the frame-text. Frame variables can hold strings (including code lines) or numeric values. They are similar to variables in programming languages, but their evaluation is done during construction time, not during the runtime of the application to be generated. For example, a slot that embeds a frame variable can be used for the name of a class which becomes generated (see Listing 3.6, line 9). The command for setting variables to a value or an expression is the “set-var”-command. The command for a slot that embeds the frame variable in the frame-text is the “value-of”-command [Jar03].

There is another kind of slot that holds whole frames. This is done by the adapt- or copy-command [Bas97; NN01]. The adapted frame has variation points (which are also considered a type of slot) which can be filled, removed or changed by the adapting frame. It is also possible not to change the variation point at all. In general it is a good idea to specify many variation points and apply changes only when needed. Such a variation point is called *breakpoint* in XVCL (<break name="aName">) [NN01] and *Netron Fusion* (.break) [Bas97] and just *variation point* for the *FrameProcessor* (VP) [Pat02]. The adapting frame has the choice of modifying the variation point. This can be done by the insert-commands. The commands insert-before and insert-after append the variation point on the appropriate end, while insert removes the original content of the variation point and replaces it by the new content, which may also be empty (a

---

14 The determination of variation points can be done by the application of domain engineering (see 3.2 Domain Engineering).
3.2 Different Base Concepts

remove). This technique can also be applied to insert changed variable values in sub-frames (see lines 13-15 of Listing 3.6).

There are some additional commands provided by adapt-based frame processors which can be considered slots, because they also have the possibility to change the frame-text. These commands include, for example, the while- and select-commands [Jar03].

It is important for the adapt concept that it provides a mechanism that can be used for expected changes as well as for unexpected ones. For expected changes, a breakpoint has to be inserted in the place where a difference is known and changes have to be applied by the adapting frame. However, it is also possible to enclose (without much effort) interesting parts of code, like a method or the core algorithm of a method in breaks, where no change is expected, at the present time.

While working with unexpected changes, it is even possible to build a software-system without planning for reuse in the beginning. By inserting some variation points, the frames become reusable for many unforeseen tasks later. This way, reuse can even become induced in legacy systems [Bas97]. It is however better of course to develop with the reuse idea in mind from the beginning.

In the latter case, the variation points can be identified during the domain engineering early in the project. Therefore no unnecessary breaks will clutter the host code, thus preventing it from becoming more complex and also more difficult to read and maintain.

3.2.1.2 What is the Generator in Context of the Adapt Concept?

In the adapt concept the frame processor itself is often referred to as just “the generator”. The adapt concept-based generator takes no additional external files as input. The files it takes contain the frames and slots as well as the configuration data that is contained in SPCs (see section 3.2.1.4 “Configuration of an Adapt Concept-Based Software-System”).

3.2.1.3 Export

In order to receive source code in the target language, a frame hierarchy needs finally to be exported. In the adapt concept, this is done by executing the frame processor with the SPC of the frame hierarchy as argument (e.g. java -jar %xvcl%\xproc.jar StandardVectorSPC.S). One generation process produces one file that contains source code. For each individual file, another SPC must be used. If the same SPC is used, the same source code file is generated. It is possible to use different SPCs for the same frame hierarchy. This is useful for obtaining differently configured variants of the same source code. In the example provided in section 5.3 “Using Frame Technology for Software-System Families“ this is used to produce Java vector-classes with different dimensions (e.g. classes for two or three dimensional vectors). The name of the exported source file can be specified by a frame command in the appropriate SPC. In XVCL, for example, the outfile attribute has to be specified in the x-frame element (see section 6.1.6 “XVCL processor”).

Another term that is important in context of the export command is the term frame instance. It has a similar, but different meaning in the two frame technology concepts. The meaning in context of the adapt concept is provided here: When a frame is processed by the frame processor, its commands are executed and its slots become filled. This happens directly before the export. The result is an instantiated frame, a so called frame instance. It is free of previous frame commands and the content consists of target language blocks only. The frame lost its generic character and is now specific [Bas97, p. 88].
3 Frame Technology

3.2.1.4 Configuration of an Adapt Concept-Based Software-System

The configuration data is used for the problem space of the generative domain model (see section 5.1 “Technology Projection for Generative Programming Using Frame Technology”). Using the adapt concept all configuration data can be changed in an appropriate SPC. Each SPC holds the configuration for its frame hierarchy. To assemble a system from a given frame hierarchy, the frame processor starts with the SPC and works its way down the tree, filling slots and executing commands [Ba97]. In a software-system family, the SPC is the configuration for one specific system. This system can be tailored for specific needs by altering the SPC. There are even frame libraries containing templates or prototypes for SPCs which can be filled out by customers to fit their individual needs (see Figure 3.6). There could be for example a web-shop in the Internet selling some kind of application. Each buyer comes finally to an order page which allows him to specify his special configuration (Custom Specs) for his application by use of HTML-typical buttons and list fields. The order page is in fact a design template. The entered data then gets sent to the server, where a human programmer, or better a utility builds the SPC from this specification. Finally, the SPC and its frame hierarchy become processed, generating the customized source code ready for further processing (that is, for example, compiling it to a binary and sending it to the buyer).

Figure 3.6 How to use a SPC

(Adapted from [Bas97, p. 19])
3.2.1.5 Detailed Example

To get a better understanding of the adapt concept an example is provided from the vector domain (see 5.2 “Using Frame Technology for Single Software-Systems” for more details). Figure 3.7 shows a feature diagram wherein all features are already realized in Listing 3.5, except the feature of having specialized unit vectors (red circle). A unit vector is a vector for which all elements but one are zero. The non-zero element must have the value 1, i.e. (0,...,0,1,0,...,0). The new feature should add an implementation class for unit vectors which provide an optimized storage for these vectors. The previous feature design could have been already made without planning for unit vectors. To extend the system with unit vector classes, only the differences to the other vector classes have to be specified (“same as, except” principle). Listing 3.6 shows the frame for the generation of unit vector classes. The differences are the class attributes, the constructor, and some methods. In line 11 the array which saves the values of a vector is replaced by an integer which just saves the position of the only value which is 1. The code of the constructor is replaced to set the start value of a unit vector (line 14), instead of an array of values of another vector (see the lines 27-30 of Listing 3.5). The getComponent and setComponent methods have changed (lines 16-24 and lines 25-40) and the component operations add and sub are removed (lines 41-43). Finally, a new method has been added to retrieve the position of the one-value directly (line 41).

---

Figure 3.7 Feature diagram for vector classes extended by unit vectors

---

15 The vector for which all elements are zero is not a unit vector, but it should also be allowed to stored such as vector as an unit vector object for performance an convenience reasons. Therefore an implementation trick is used. The storage “-1” as positioning value mean an vector for which all elements are zero.

16 add and sub can be useful in the case that they return an instance of a common base class or interface (see section 5.3 “Using Frame Technology for Software-System Families”), but are deleted here for the ease of the example.
3 Frame Technology

Listing 3.5 A frame for generating vectors (Vector.XVCL)

```xml
<Frame TECHNOLOGY="Frame">
  <value-of expr="?@VECTOR_CLASSNAME?"/>
  <Value expr="VALUES"/>
  <Value expr="?@VECTOR_CLASSNAME?"/>
  <Value expr="?@COMPONENT_TYPE?"/>
  <Value expr="?@COMPONENT_TYPE?"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
  <Value expr="VALUES"/>
3.2 Different Base Concepts

The SPCs for this example are shown in Listing 3.7, Listing 3.8 and Listing 3.9. Figure 3.8 illustrates, how the frames adapts each other. The processing of the StandardVector-SPC generates the Java class file StandardVector.java, while the processing of the UnitVector-SPC generates UnitVector.java. Both class files share parts of the StandardVector-Frame. This means, reuse has been applied to the UnitVector-class. It is further possible to increase the amount of reuse in this example if vectors with other component types are needed. In the two SPCs there is configuration data present which specifies float as component type. By providing additional SPCs for the different component types (e.g. int, long or double) additional vector classes can be exported. This can easily be achieved by changing the value of the frame variable COMPONENT_TYPE.
This example further demonstrates that there is no general mapping between the different diagram types. The variability of a vector (see the feature diagram in Figure 3.7) has no direct influence on the realization of these variations in a class structure (as shown in the UML class diagram in Figure 3.9). The reuse optimized assembly of frames (as shown in Figure 3.8) also differs. In this case one dependency can be recognized: Each SPC has become a class in the class diagram. However, this is no general rule. The export of a SPC could result in multiple classes (or large pieces of otherwise structured code in the case of another paradigm than that of object-oriented programming) as well as in only small parts of code (or text which is not even code).

![Figure 3.8 Export of adapt concept-based frame hierarchies](image-url)

Listing 3.7 SPC of the Vector-Interface (VectorInterfaceSPC.S)

```xml
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///c:\XVCL_1.0_beta2\dtd\xvcl_1_0.dtd">
<x-frame name="VectorInterfaceSPC" outfile="Vector.java">
  <adapt x-frame="VectorInterface.XVCL">
    <insert-after break="VECTOR_PARAMETERS">
      <set var="COMPONENT_TYPE" value="float"/>
    </insert-after>
  </adapt>
</x-frame>
```

Listing 3.7 SPC of the Vector-Interface (VectorInterfaceSPC.S)
3.2 Different Base Concepts

3.2.1.6 Conclusion

The main advantage of the adapt concept is its capability for handling expected and unexpected changes very well and conveniently. Parts of the code can be easily replaced, removed or extended by the use of insert commands. Because of its capability for handling unexpected changes, it is well suited for - but not limited to - classical single-system development processes and legacy systems. It can be applied to projects which use development processes targeting software-system families (like GP) as well. Another advantage arises of the quite simple structure of the adapt concept driven frame hierarchies. Missing the dynamic possibilities of the abstraction concept, they are more limited (e.g. adapt concept-based frame instances can not be relocated), but therefore also easier to understand. This can be an important advantage.

Figure 3.9 One possible implementation of the feature diagram in Figure 3.7
3 Frame Technology

Table 3.2 shows a comparison of some important frame commands of the adapt concept-based frame processors. It is provided to see the differences in syntax. Most commands shown have additional parameters not covered here. For a full list of frame commands and their parameters, see the tool's appropriate manuals.

<table>
<thead>
<tr>
<th>FrameProcessor syntax</th>
<th>Netron Fusion syntax</th>
<th>XVCL syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTFILE aFilename</td>
<td>frame</td>
<td><code>&lt;x-frame name=&quot;aFrameName&quot; outfile=&quot;aFilename&quot;&gt;...&lt;/x-frame&gt;</code></td>
</tr>
<tr>
<td>VP aSlot</td>
<td>.BREAK aSlot</td>
<td><code>&lt;break name=&quot;aSlot&quot;&gt;...&lt;/break&gt;</code></td>
</tr>
<tr>
<td>VP_END</td>
<td>.END-BREAK aSlot</td>
<td><code>&lt;adapt x-frame=&quot;aFrameName&quot;&gt;...&lt;/adapt&gt;</code></td>
</tr>
<tr>
<td>ADAPT aFileName</td>
<td>.COPY aFileName</td>
<td><code>&lt;insert break=&quot;aSlot&quot;&gt;...&lt;/insert&gt;</code></td>
</tr>
<tr>
<td>INSERT_AFTER</td>
<td>.INSERT-AFTER</td>
<td><code>&lt;insert-after break=&quot;aSlot&quot;&gt;...&lt;/insert-after&gt;</code></td>
</tr>
<tr>
<td>INSERT_BEFORE</td>
<td>.INSERT-BEFORE</td>
<td><code>&lt;insert-before break=&quot;aSlot&quot;&gt;...&lt;/insert-before&gt;</code></td>
</tr>
<tr>
<td>REPLACE</td>
<td>.REPLACE aVariable BY anExpression</td>
<td><code>&lt;set-var=&quot;aVariable&quot; value=&quot;anExpression&quot;/&gt;</code></td>
</tr>
<tr>
<td>SELECT</td>
<td>;aVariable WHEN aValue</td>
<td><code>&lt;select option=&quot;anExpression&quot;&gt;...&lt;/select&gt;</code></td>
</tr>
<tr>
<td>WHILE</td>
<td>;aVariable END-WHILE</td>
<td><code>&lt;while using-items-in=&quot;?@MultiVariable?&quot;&gt;...&lt;/while&gt;</code></td>
</tr>
</tbody>
</table>

Table 3.2 Examples of important commands of adapt concept-based frame processors

3.2.2 The Abstraction Concept

3.2.2.1 Overview

The abstraction concept was invented by the Delta Software Technology GmbH [DST02a]. Their tool ANGIE (see chapter 6 “Available Frame Processors”) realizes this concept. Another tool, XFramer (see also chapter 6) has adapted this concept. XFramer was inspired by ANGIE, but works like a preprocessor.
3.2 Different Base Concepts

Abstraction concept-based frame technology is similar to object-oriented programming, while frames are not used during runtime but during generation-time. ‘Frames are code modules as objects’, states Delta Software Technology [DST]. This means that frames contain basic code building blocks and can be used like objects in an object-oriented environment. Frames and frame instances can be distinguished. Frames correspond to classes while frame instances correspond to objects. Therefore frame instances of the abstraction concept can have a full life cycle during the generation process. The name of this concept is derived from its principle of abstraction. Classes express the abstract ideas of actual objects. In the same way, frames express the abstract ideas of actual frame instances.

Also adapted from object-oriented programming is the interface concept. A frame has an interface. The interface is the sum of the frame's slots/frame variables. Only slot-contents can be changed from outside the frame. The actual values of the frame variables are thereby considered the state of the frame.

The abstraction concept furthermore unifies the different kinds of slots. There is only one kind of slot which is equivalent to the various slot types of the adapt concept. It can hold frame variables of various data types depending on the frame processor. There are data types for strings (this includes code lines), numeric values and frames [DST02d].

As already mentioned, a frame in the abstraction concept is similar to a class in object-oriented programming. A frame can be instantiated statically (XFramer with C++ as frame language only) by declaration or dynamically by the appropriate frame command (CreateFrame() in ANGIE, new in XFramer) from within a function or an adapting frame [DST02d]. In the case of all frames being instantiated by other frames using declarations, a static frame hierarchy is built. It is, however, possible to organize the frames dynamically. First, the frames become instantiated, then they will be connected by functions in a function-controlled way [DST01] (see Figure 3.10). In contrast to the adapt concept the connection of the frames can be done from the outside. Functions can even change already instantiated frames by altering their slots by inserting new values to them. In order to do this a slot can be changed like an attribute of an object in an object-oriented language.

![Figure 3.10 Instantiation of frames](Adapted from [DST02b])
In the case of frame-holding variables this even allows to change the structure of the frame hierarchy after the instantiation, resulting in completely dynamic frame hierarchies.

The abstraction concept further distinguishes between different types of frame commands. There are flags, procedural commands and function calls.

**Flags**

A flag does not have the intuitively expected form of a command. Despite this, a flag is considered a type of frame command. There are flags for various tasks. All frame processors provide flags for the embedding of slots. Depending on the frame processor, there are flags available for optional code blocks (see section 6.1.1 “ANGIE”), for the embedding of binary components and other specialized tasks. Typically, the syntax of an opening flag is a bracket followed by a symbol that describes the specific command (i.e. <! for slots). By convention, the closing flags use the same symbols in the opposite direction (i.e. !> for slots). Between an opening and a closing flag, there is specific information for this type of flag (i.e. the slot-name in the case of a slot-flag). In ANGIE and XFramer, the syntax of these flags can be changed by the user. This is useful if the target language code contains symbols which are identical to the standard flags. The comment-tag of HTML, for example, looks like <!-- some comment -->. This tag would raise a conflict with the XFramer flags <!. If the XFramer flag is, however, set to e.g. {@, the problem is solved.

**Procedural Commands**

Procedural commands are responsible for the control flow. They can always be used in functions and methods. In ANGIE, procedural commands are not allowed in frames while in XFramer they are not allowed in target language blocks (started by BEGIN_FRAME_TEXT). Typical procedural commands are if, for and while.

**Function Calls**

Function calls invoke functions which have been defined before. Functions can be called from the definition of other functions (including the main function). In ANGIE function calls are allowed also in the declaration part of a frame. Each declaration part is at beginning of a frame in ANGIE. Frame variables are declared and initialized there. A function may be called in order to initialize a frame variable. Each function must be defined. Three different types of functions can be defined: intrinsic functions, library functions and user-defined functions.

**Intrinsic Functions**

An intrinsic function is a function which belongs to the frame processor [Gi03; EGS03]. It is elementary and an essential part of the frame processing language. Typical intrinsic functions are functions for the export of frames or for controlling the generation process.

**Library Functions**

Library Functions are external functions that can be used during the generation processes. There is no limitation on what these functions can be used for. The ANGIE-package already includes many libraries (e.g. a library for processing XML-Documents). With XFramer, any library available for the applied frame code language (i.e. Java, C++ and C#) can be used.

**User-defined Functions**

Like many programming languages, abstraction based frame processors allow user-defined functions. This is the place where the major part of the generation process is implemented. The definition of such functions is not different from the definition of typical functions in general.

---

18 The term function also means method here. Methods are only available in the context of XFramer.
3.2 Different Base Concepts

programming languages. They have parameters, return values, and so on. In **ANGIE** these functions are called *script-functions*. In **XFramer** they are *free functions* or *methods*. If **XFramer** is used in conjunction with Java, there are no free functions. There are only methods which belong to classes or frames. If **XFramer** is used in conjunction with **C++**, free functions and methods are both available.

### 3.2.2.2 What is the Generator in Context of the Abstraction Concept?

![Figure 3.11 Principle of an abstraction based frame processor](image)

Figure 3.1 shows the basic principle of development with a frame processor. A more detailed view of this principle in context of the abstraction concept is provided in Figure 3.11. An abstraction concept-based frame processor does not produce the source code as output itself. Instead, it produces a *custom generator* which in turn generates the source code. By strictly applying the GP definition of a generator (see 3.1.3 What is a Generator?), there are several generators in this figure. The compiler is a generator, as well as the frame processor. However, by speaking of *the generator*, in context of the abstraction concept typically the custom generator that produces the source code is meant. In contrast to the compiler and frame processor which are general generators suited for many problems, this generator is specialized for a specific domain and problem. Additionally the combination of frame processor and custom generator is also often just called generator. Therefore it is important to understand which generator is meant in a specific context. There are two possibilities how the frame processor and the custom generator are associated. The first possibility is that the frame processor works similar to a compiler. It takes the frames and functions as input and generates a standalone program – the customized generator. Running the
customized generator produces the source code in the target language. This is the way \textit{XFrmer} works if C++ is used as the frame code language (see section 6.1.5 “\textit{XFrmer}”). The other possibility is that the frame processor consists of two parts – compiler and interpreter/virtual machine. The compiler translates the frames and functions to an intermediate code which is then interpreted by the virtual machine of the frame processor. This is the way \textit{ANGIE} works. \textit{XFrmer} also works this way, if Java is used as the frame code language.

In Figure 3.1, it is shown that a high level \textit{specification} is the input of the generator that consists of frame processor and custom generator. This specification can be further distinguished as Figure 3.11 illustrates. One part of the specification consists of \textit{frames and functions} (including internal configuration data). This part is the specification for the frame processor. It is necessary to build the custom generator. The other part is the \textit{external configuration data} (sometimes just referred to as \textit{the configuration}) which is parsed during the generation process by the custom generator. Therefore, it is the specification for the custom generator. The configuration data is used to specify the exact configuration for the produced variant. If for example, the generator produces image processing applications (see the ABA-prototype in [Sle02]), the configuration data can be stored in a XML-file that specifies which features are supported, on which target platform and architecture the application runs, what kind of optimizations should be applied, and so on. In the generative domain model the term \textit{specification} (with the meaning of \textit{configuration data}) is used to describe the elements of the \textit{problem space}. Therefore it is important to understand the context, in order to figure out the meaning of the term \textit{specification}.

3.2.2.3 Export

It is also true for the abstraction concept that in order to receive source code in the target language, a frame hierarchy needs finally to be exported. The abstraction concept allows to do an export at any time during the generation process, not only at its end. With the export command of the abstraction concept, a function can decide at any time to export one or more instantiated frames to actual source code [DST02d] (see Figure 3.17 for an example of the relation between instantiate and export). When a frame instance is being exported, all frame variables become bound. Along with the frame instance, the underlying frame hierarchy is exported, making the frame instance a root frame (see 3.2.2.4 “Configuration of an Abstraction Concept-Based Software-System”). The export function is an \textit{intrinsic function} of the frame processor. Its syntax depends on the frame processor. There are different export-functions for different output media. The most common use is the output to a file, but it is also possible to export to a frame variable in order to reprocess the result of the export. The export command which exports to a file takes the file name and file-type/file-extension as parameters. A single generation process can produce any number of code containing files. This enables many different methods to build a generator. To get a better understanding, one easy and common method to use this concept together with an object-oriented language is provided in the following.

The frames have to be designed first, with – ideally but not necessarily - each one representing one class. Some additional (helper) frames as parts of these classes might be useful too. In Java for example, each frame is a .java-file containing just one class, while in C++ there are frames for the header files (usually .h) containing a class definition and each of them is accompanied by a matching frame for the code file (usually .cpp or .cc) containing the implementations. Then functions are provided for each class defining frames, or in smaller projects, one function for all frames. The frames contain slots for possible connections, which will be established by the function(s), depending on the specification. After the connection is done, each function exports the instantiated frames of the frame which it is responsible for. Or in the case of only one function, this function does all the exports for its frames. Please notice that it is useful to export one frame as
3.2 Different Base Concepts

different classes on different places in the object hierarchy with its slots filled with different values. Class hierarchy and frame hierarchy are therefore completely different.

The prerequisite for the abstraction concept, with the ability to change already instantiated frames and export them multiple times, is a multi-phased (see 6.1.1 “ANGIE”) generator engine [DST02b, p. 15]. The next section therefore describes the original idea of Delta Software Technology’s frame technology based multi-phased generator. Please notice that only the theoretical principle of this is explained. It is different from the practical realization of the abstraction concept as explained earlier in this chapter.

---

**Multi-Phased Generation**

A single-phased generator (see Figure 3.12:2) works sequentially. It takes its input (see Figure 3.12:1) which consists of specific parameters and commands mixed with code of the target language and processes it. In case of a frame processor these are the frames and functions. During the evaluation of the input source, the generator can use data from a repository (see Figure 3.12:3) to enhance or check the input. A repository in this context means any kind of data storage that contains configuration data (e.g. a database or a XML-file). Finally, it produces an output (see Figure 3.12:5) with the production process running mostly in the same order as the input is structured. These outputs are often volatile. Every piece of generated source code is instantaneously released and can not be further processed, or evaluated for the processing of other source code [DST02b, p. 15].

To understand the concept of a multi-phased generator, it is necessary to be familiar with the concept of a single-phased generator first. Please notice that with the term generator in this section, generators in general according to the generator definition of GP are meant, instead of the custom generator explained in 3.2.2.2 “What is the Generator in Context of the Abstraction Concept?“.

A frame technology-based multi-phased generator instead does multiple steps for generation. Parts of the specification (see Figure 3.13:1) (i.e. frames and functions) become evaluated by the generator (see Figure 3.13:2), while data from the repository (see Figure 3.13:3) is used to enhance or check the input. Frame instances become created (see Figure 3.13:5) by the application of frames and functions (see Figure 3.13:4). Thereafter, additional generation processes (see Figure 3.13:6) can further process the generated results (like frame instances). Hereby an unlimited number of generation steps is possible, including cyclic generation (inserting the last result in the same process) (see Figure 3.13:7). The frame instances (see Figure 3.13:5) act like a kind of temporary
3 Frame Technology

repository during this process. At last, export-functions (see Figure 3.13:8) are called that create the output files. These functions can also be called at any time and repeatedly during the generation process [DST02b, p. 16].

Delta Software Technology states that multi-phased generation is very flexible and offers many advantages, e.g. support for aspect oriented programming [DST02b, p. 15].

3.2.2.4 Configuration of an Abstraction Concept-Based Software-System

The abstraction concept provides various options for the configuration of a frame-based software-system. As already mentioned, the configuration data is part of the problem space of the generative domain model. In the abstraction concept configuration data can be as simple as function parameters or as complex as a script in a domain specific language that is provided by an external file. The available options to store and evaluate configuration data are external files, root frames, configuration frames, function parameters and command line arguments.

External Files

External Files can store any kind of configuration data. These files become processed by the frame processor at generation-time. The processing can be further assisted by the application of frame commands (mostly library functions) that support the format of the configuration file. In ANGIE, for example, there are library functions which provide support for XML and HUTN\(^\text{19}\) [Gie02b]. They are typically used for specification of a software-system with a domain specific language. In the ABA-prototype [Sle02], for example, graphical user interfaces which have been specified in an external XML-file are generated by ANGIE (see Figure 3.14 and Figure 3.15). XFramer can use generally available libraries for C++ or Java to support domain specific languages (e.g. Xerces [ASF] for XML).

\(^{19}\) Human-Usable Textual Notation (HUTN) is an upcoming standard of the OMG [OMG].
3.2 Different Base Concepts

Root Frames

A root frame is the highest node (root) in a frame hierarchy which will be exported [Gie03]. Therefore not every root of a frame hierarchy is actually called root frame. Because an abstraction concept-based frame processor can process several frame hierarchies at once, there may be more than one root frame. A root frame can store configuration data for its underlying frame hierarchy. In ANGIE this can be done in a very convenient way by the application of scoped variables [Gie02a] (see Listing 3.4 for an example). A root frame, however, does not necessarily need to store configuration data. Its main purpose is to control the export of its frame hierarchy. A root frame that stores configuration data can be considered a SPC. In the abstraction concept it is, however, unusual that only SPCs are used for configuration issues [Gie03]. Please notice that the term root frame applies to frames as well as to frame instances.

Configuration Frames

A configuration frame is a frame that stores configuration data. Frames can contain static variables that hold the configuration data which can be queried by other frames. An example is an unconnected frame holding global information which is queried from different frames contained in several frame hierarchies. Please notice that a root frame which stores configuration data is also called configuration frame.

Function Parameters

Configuration data can also be stored in form of function parameters. High level functions or constructors for frames are especially suited (see line 152 of Listing 5.1).

Command Line Arguments

One convenient method for global configuration data are command line arguments. Abstraction concept-based frame processors start the generation with the execution of the main function. The main function can take user arguments from the command line. Therefore global configuration issues like memory or speed optimization behavior, for example, can be changed very easily.
3 Frame Technology

A small example (see Figure 3.16) for this could be a frame containing a vector blueprint. At first, the frame's function fills the slots of the frame to instantiate a vector-base class. This could be done by filling a slot class-name with "Vector", a slot "extending" with "empty", the method slots with help frames which contain methods for the basic vector (this means connecting them), and so on. Then it instantiates another frame, this time for a specialized vector which automatically normalizes on creation. This could be done respectively, by filling the class-name slot with "NormVector", the "extend" slot in a form that the class derives from "Vector", the methods slots with "empty", because the methods will be inherited by the class-structure and by attaching another help frame which contains a normalization algorithm, to the constructor slot. Another specialized vector can have a very different purpose, but may reuse the normalization algorithm (which may, but does not have to be a method) as part of another operation. More specialized vectors could be derived from the base vector in the same way. Therefore a class hierarchy is created out of a class frame and some helper frames which is very different from its frame hierarchy (see Figure 3.17).

3.2.2.5 Detailed Example

Figure 3.16 UML-classdiagram of the result of a "vector"-generator using the abstraction concept-based method
3.2 Different Base Concepts

3.2.2.6 Conclusion

The advantage of the abstraction concept is that it has many methods to generate code in a strictly controlled way. If during the development of a generator a decision has to be integrated, there is always an appropriate control structure which can be used in a function. Abstraction concept-based development behaves more like a kind of object-oriented meta-programming than just connecting frames.

Table 3.3 shows a comparison of some important frame commands of the abstraction concept-based frame processors. It is provided to see the differences in syntax. Most commands shown have additional parameters not covered here. For a full list of frame commands and their parameters, see the tool’s appropriate manuals.

<table>
<thead>
<tr>
<th>ANGIE syntax</th>
<th>XFramer syntax (Java version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Frame aFrame</td>
<td>frame aFrame {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.End Frame</td>
<td>}</td>
</tr>
<tr>
<td>&lt;!aSlot!&gt;</td>
<td>&lt;!aSlot!&gt;</td>
</tr>
<tr>
<td>&lt;? OptionalCodeBlock ?&gt;</td>
<td>&lt;!aVariable != null) ?</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>:&quot;&quot;!&gt;</td>
</tr>
<tr>
<td>.Dim aVariable as aType</td>
<td>private/public aType aVariable;</td>
</tr>
</tbody>
</table>

Figure 3.17 An example of a method using the abstraction concept
3 Frame Technology

<table>
<thead>
<tr>
<th>ANGIE syntax</th>
<th>XFramer syntax (Java version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Function aFunction as aType</td>
<td>private/public aType aMethod {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.End Function</td>
<td>}</td>
</tr>
<tr>
<td>.Export (aFrameInstance, aFilename, Filetype)</td>
<td>aFrameInstance.exportToFile(aFilename);</td>
</tr>
<tr>
<td>.aFrameInstance=CreateFrame(“aFrame”)</td>
<td>aFrameInstance=new aFrame;</td>
</tr>
<tr>
<td>.add( different Parameters here )</td>
<td>depends on the used data structure</td>
</tr>
<tr>
<td>=, +, -</td>
<td>=, +, -</td>
</tr>
<tr>
<td>.If aCondition Then</td>
<td>if (aCondition) {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.Else</td>
<td>else {</td>
</tr>
<tr>
<td>.End If</td>
<td>}</td>
</tr>
<tr>
<td>.For aVariable=aValue to anotherValue</td>
<td>for (anInitialization;aCondition;anIncrementExpression)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.Next</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 3.3 Examples of important commands of abstraction concept-based frame processors
The terms *frame* and *slot* were introduced by Marvin Minsky. Minsky, who is considered by many to be the father of Artificial Intelligence [Gen02], developed a concept for understanding human thinking in 1974. As special applications for his theory he named visual recognition and language analysis. He used a so called frame-system for capturing real life situations like understanding a sight (view or picture) or a sentence in natural language. These frame-systems allow programs to analyze such classes of problems [Min74].

Minsky's definition of frames states: “A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed.” [Min74]

Many other scientists based their theories on Minsky's work. Among others the most important scientists are Schank and Abelson. Their work is about computer analysis of natural language. Instead of *frames* they use the term *scripts* which describes a very similar concept [Pet]. Scripts are used for the recognition of real life situations, like Minsky's frames, but only in the context of natural language. A script is a sequence of knowledge statements describing a generic context, e.g. a restaurant-situation [SA77, p. 43], but not the specific details of the situation [SA77]. Schank and
Abelson also used other concepts for the analysis of natural language, known as plans and goals [SA77]. These are, however, not related to frames.

This theoretical work leads to the development of natural language analysis tools, dictionaries and common sense reasoning. Such tools and dictionaries are (among others)

- FrameNet (http://framenet.icsi.berkeley.edu/~framenet),
- ThoughtTreasure (http://www.signiform.com/tt/htm/tt.htm),
- WordNet (http://www.cogsci.princeton.edu/~wn) and
- Cyc (http://www.cyc.com).

Miriam R. L. Petruck, who is a member of the FrameNet development team, defined frames this way:

“A FRAME is any system of concepts related in such a way that to understand any one concept it is necessary to understand the entire system; introducing any one concept results in all of them becoming available.”

However, these tools can not be considered being frame processors for the purpose of this paper. Independent from each other Paul Basset and the company Delta Software Technology based their work on Minsky. Basset borrowed the terms frames and slots from Minsky, because of many existing analogies. Minsky's frames have terminals also called slots which must be filled by specific data. Slots can even specify conditions [Min74, p. 1]. Both is true for modern frame processors too. Another similarity is that slots can have default arguments [Min74, p. 2]. Frames can have sub-frames and are organized into frame-hierarchies [Net02a] which Minsky called frame-systems [Min74]. But there are also a lot of differences. Minsky used his theory in the environment of AI and never thought about the generation of programs (or at least he did not publish these ideas). The main difference, however, is their purpose. Minsky's frames were used for the analysis of real world situations like speech and images, Basset's for the synthesis of computer understandable code. Minsky's frames were intended to work at run time, Basset's frames work during construction time, before the compilation or the start of an actual program. [Bas97, p. 115]

Basset developed the frame technology from classical code generators. By the fall of 1979, he had to build a software-system for the city of Barrie, Ontario. The requirements of the software-system could not be easily handled by code generators. So Basset had the choice of changing the generated code manually or apply some major improvements to the generator. These improvements included the development of a new concept that used small customizable software parts, called chunks. When chunks eventually became frames, it was the genesis of frame technology [Bas97, p. XV]. Netron Fusion, developed in 1987, was the first frame processor based on Basset's work, with Basset being a co-founder of Netron [Net02b]. His ideas became adopted in 2001 by the National University of Singapore which developed the XVCL processor [Nus02]. Two more frame processors based on the same principle were developed in 2002: FrameProcessor and FPL. Each of these frame processors has its own specific features and application area.

Delta Software Technology is a company with a main focus on software generation. In the early seventies, they developed a tool called PDL/VM (Processor Description Language/Virtual Machine). It is a tool for the development of generators using a language similar to the programming language Pascal. Based on PDL, there are several more specialized tools for generation. The most important of these tools is DeltaMacro [DST93] which is a kind of universal preprocessor. It was mainly used for batch processing applications on mainframes. In 1988 Delta Software Technology became aware of Minskys work. ’If frames can even express natural language, then it must be easy to express synthetic language”, stated Schilling as the basic idea he had [Sch03a]. Therefore they developed a tool called ROOT [DST97], as an experimental prototype first and used it in many industrial projects later. It was, however, not the successor of DeltaMacro.
4 History of Frame Technology

The latter was still in parallel use. ROOT, which can already be considered a frame processor, is completely realized in Prolog. Prolog is a logical programming language with the ability to evaluate expressions in different directions. The expression $p_1 = e^{p_2}$, for example, has the standard evaluation direction which calculates the value of $p_1$, depending on the value of $p_2$. But Prolog also allows to calculate the value of $p_2$, for a given $p_1$, without the need to transpose the equation. The same technique is used by ROOT for the expression \texttt{transpose(program_source, frame_definitions, frame_instances)}. Therefore it is possible to transform frames to code or vice versa. ROOT supports analysis as well as synthesis of program sources. In 1999 Delta Software Technology decided to develop a successor to ROOT with the best features of Delta/Macro and ROOT. This successor called ANGIE dropped the ability to transform code to frames (analysis), but still uses this evaluation-approach [DST02a]. ANGIE also uses its own script language instead of Prolog. Three years later (2002), Max Schlee adopted the principle of ANGIE and created another tool called Cframer that can be used together with a C++-Compiler as a frame processor. Later he extended Cframer, so that it can be used also together with Java and C#. To express the ability of working with various frame code languages, he finally renamed his tool to XFramer.

Today, there are six frame processors and a constantly growing number of scientific papers in the area of frame technology available. Only the future will show if frames become accepted in the large.
Frames are a technology which can be used with the paradigm of generative programming to generate software components and even to construct whole software factories (see section 2.1.2 “What is a Software-System Family ?”). Such a software factory produces software-systems of a specific software-system family. However, to use frame technology with the paradigm of generative programming, a technology projection is needed which explains how the different aspects of the generative domain model is mapped to the specific elements of frame technology. Figure 5.1 shows a graphical overview of this technology projection.

![Figure 5.1 Technology projection for generative programming using frames](Based on [ES02])

### 5.1 Technology Projection for Generative Programming Using Frame Technology

In nearly every use of generative programming a generator is involved. This is also true for frame technology. In frame technology this generator is the frame processor or was created with the frame processor (see section 3.2.2.2 “What is the Generator in Context of the Abstraction Concept?”). The input of the frame processor consists of frames, functions and configuration data (e.g. XML-Files). The output are software components or whole software-system family variants. This means that the
frame processor and its environment (including graphical front-ends, XML configuration files, and so on) can be considered the software factory. In frame technology the whole model is covered by the frame processor, its frames, its functions, and its environment. This is the case because the frame processor is not a generator written for a specific problem, with specific configuration knowledge included, but a general processor which becomes specifically configured by the frames and functions which can also cover more than mere configuration knowledge. In detail, a projection to GP can be made the following way:

The **solution space** is covered by specific frames and functions. The frames of the solution space are low level and contain (small) solution parts in the target language. This can, for example, be a class definition or a single method. Both are of course customizable by the slots included. The functions (abstraction concept only) of the solution space are functions which generate specific frames for specific solutions in the target language. For an example see Listing 5.1. Lines 22-98 show a vector-frame, which contains a Java class.

The **configuration knowledge** is also covered by frames and functions. The frames and functions of the configuration knowledge, however, are of another kind. They are of mid or even high level and are mainly used for structure and organization purposes. They do not contain language specific code. Instead, they contain information about how the frames of the solution space and other frames of the configuration knowledge have to be assembled. This information includes the construction rules, dependencies, optimizations, illegal combinations, and default values. In fact, a whole software architecture and its design, which in most other paradigms is not kept in computer processable form, can be included here. In the case of illegal combinations which were specified in the problem space but detected here, the frame processor may stop and issue an error. The default values are defaults for slots, not specified in the problem space, but needed in the solution space. They can be filled in, depending on other configuration issues. Because of the dependencies and the construction rules there is often a complex logic involved. Therefore, the elements of the configuration knowledge contain a larger amount of frame commands. Examples for the configuration knowledge can also be found in Listing 5.1. In lines 102-118, the default values for vector classes are set, whereas the code of lines 133-146 is responsible for the difference in the assemblage of a vector class with a limited or an unlimited number of dimensions. Hereby the the attached frames (lines 6-20) belong to solution space, because they contain target language code (in this case Java), while the lines between 133 and 146, controlling the construction process, are actually code of the configuration knowledge.

The **problem space** is covered by many different elements that contain domain specific concepts and features [CE00]. Problem space elements are expressed in a *domain specific language* (DSL). A domain specific language in this context can mean as much as a fully specified domain language with its own grammar or as little as frame commands parametrized with domain specific terms, even a graphical description is possible. Elements of the problem space are also often referred to as specification. Specification in context of a frame processor has however a more general meaning, as explained in the sections 3.2.1.2 and 3.2.2.2. Therefore the problem space is not mapped to the specification (frame technology meaning), but to *configuration data* (see the sections 3.2.1.4 and 3.2.2.4). Problem space elements can be kept “inside” or “outside” of the frame code.

Inside the frame code, there are the SPCs, root frames, configuration frames and function parameters depending on the used frame technology concept. To provide an example, line 152 of Listing 5.1 shows the configuration of a specific variant of a vector class as function parameters. The set of all allowed parameter values can be considered a DSL. Therefore line 152 is part of the problem space. Additionally there can be other frames and functions (usually high level) that cover domain specific knowledge.
5 Frame Technology in the Context of Generative Programming

Outside of the frame code, there can be front-ends and external configuration files (see Figure 5.2 and Figure 5.3). A front-end can be graphical or text-based, therefore providing an easier interface that becomes translated to another form of configuration data that can be directly processed by the frame processor. In case of the adapt concept this is usually a SPC. Netron Fusion for example provides a library of templates that can be filled out for the generation of a SPC [NE01] (see also 3.2.1.4 “Configuration of an Adapt Concept-Based Software-System”). In case of the abstraction concept, it is usually a root frame or in turn an external file. External configuration files can be written in XML or another language that the frame processor is able to process (e.g. the ABA-Prototype, see [Sle02]).

As already stated (see 3.1.1 “Overview of Frame Technology”), frames can be used for intra-application variability (building single software-systems) as well as for inter-application variability (building software-system families). The following two chapters provide a detailed explanation for these applications of frame technology.

![Figure 5.2 Transformation of front-end configuration data in context of the adapt concept](image-url)
5.2 Using Frame Technology for Single Software-Systems

If frame technology is used for the construction of a single software-system, it is possible that the system is developed with traditional technologies while for some parts of the system, frames are used to generate customized components. These components become included in the system by writing glue-code which is, however, often less time consuming to write and also easier to maintain than traditional glue-code. In the best case, there is no glue-code at all. This is possible, because the components are already customized for the specific software-system, unlike traditional components (e.g. ActiveX-Controls, JavaBeans, just plain libraries, and so on) which are very different depending on the source.

It is possible that some frame hierarchies are written for a specific system alone, but more often, they will be designed for reuse in other projects too. A software-system may use generated code from many different frame hierarchies, e.g. a customized sound-library, a customized vector class and a customized user interface, and also different generated code from the same hierarchy, e.g. two types of vector classes. The latter example is provided here with more detail:

Considering a frame hierarchy for the generation of vector classes in Java, it should provide the following features:

- A variable number of dimensions, including unlimited dimensions during runtime.
- Different component types (int, float, double, and so on).
- Different kinds of memory usage and optimizations for special vector types, e.g. a unit vector.
- Different kinds of operations which can be included in a class or not, depending on the requirements in the context of the software-system.

Figure 5.4 shows a feature diagram for a specification supporting these requirements. In a real project, it would be more complex including more vector and component types and also more operations. However, to provide an easily understandable example it had been kept simple.

Figure 5.3 Transformation of front-end configuration data in context of the abstraction concept
A specific software-system (e.g. an application using a three dimensional graphical view) needs, for example, a standard three dimensional vector with constructors and only one operation that calculates the inner product. The component type for this example is float. Instead of writing the class (see Figure 5.5) and its methods manually from scratch, only these requirements have to be specified and the frame processor, using the example vector frame hierarchy, produces the desired output.

If the same software-system at a later point of time requires an additional vector which, for example, is two-dimensional, has integer component type and supports (aside from the constructors) component operations only (see Figure 5.6), it can also be generated using the same frame hierarchy. In this case the additional work for developing the frame hierarchy instead of writing the “Vector3D” class manually is perhaps already amortized by the generation of the “Vector2D”, partly due to the saved work for the creation of the second vector class and mostly due to the maintenance advantage. The frame hierarchy consists of 140 lines of code, the two generated classes comprise together about 100 lines of code. So there is a visible saving if a third vector class is needed and an enormous saving if ten or more vector classes are needed. However, “lines of code” (loc) are not a good metric and it could easily take more time developing the generic frame hierarchy consisting of 140 loc than writing three of four vector classes, with a total of 200 loc manually by the use of copy and paste. Despite this, it is still possible that even for only the two classes there is already a time saving in the long run, because of maintenance reasons. Most projects experience a lot of alterations during their life cycle. These maintenance alterations have only to be made to the frame hierarchy instead to each vector class. The result is that all vector classes benefit from it. It is furthermore possible that in the life cycle of the whole software-system another specialized vector class is needed later. It can be easily generated without having to deal with the inner functionality of such classes again. Whether the development of a frame hierarchy is preferable to the manual development depends on the life time and complexity of the system. In the majority of cases it is, however, worthwhile even if the direct savings are not immediately obvious.

Using the following ANGIE code (see Listing 5.1) the “Vector3D” class file can be created by calling the “createVector”-Function with the parameters ("Vector3D", EMPTY, "float", "yes", "yes"). The result is the “.java” file in Listing 5.2. These parameters are typical configuration data. The second parameter is kept empty, so the default value of 3 is used. For the “Vector2D” class file the parameters ("Vector2D", 2, "int", "yes", "no") produce the desired result (see Listing 5.3).
5.2 Using Frame Technology for Single Software-Systems

1. **NGEC
2. ' Project: Vector Example
3. ' Author: Marco Emrich, 16.07.2002
4. .Frame NullAssignment(DIMENSION_COUNT)
5. cValues[<?DIMENSION_COUNT<?>]]=0;
6. .End Frame
7. .Frame ConstructorAssignment(DIMENSION_COUNT)
9. .End Frame
10. .Frame ParameterlistElement(DIMENSION_COUNT, COMPONENT_TYPE)
11. <COMPONENT_TYPE> pValue<?DIMENSION_COUNT?>;
12. .End Frame
13. .Frame ParameterSeparator
14. ,
15. .End Frame
16. .Frame Vector
17. .Slots
18. .Dim CLASSNAME
19. .Dim DIMENSION
20. .Dim COMPONENT_TYPE
21. .Dim COMPONENT_OPERATIONS
22. .Dim INNER_PRODUCT
23. .Dim TYPE="Standard"
24. .Dim N .'is N-dimensional
25. .Dim NOT_N .'is Not N-dimensional
26. .Dim NULLASSIGNMENTS = createCollection()
27. .Dim CONSTRUCTOR_PARAMLIST = createCollection()
28. .Dim CONSTRUCTOR_ASSIGNMENTS = createCollection()
29. class <!CLASSNAME!>
30. {
31. private <!COMPONENT_TYPE!>[] cValues;
32. <?<?N?>private int n; //Number of Dimensions?
33. //Constructors
34. public <!CLASSNAME!>(<?<?N?>int pDimension>)
35. {
36. n=pDimension;
37. cValues = new <!COMPONENT_TYPE!>[<?DIMENSION?>];
38. for (int i=0;i<pDimension;+i)
39. {
40. ?
41. <!NULLASSIGNMENTS!>
5 Frame Technology in the Context of Generative Programming

58. "<?<!N!>"
59. }
60. }
61. }
62. }
63. }
64. <?<!NOT_N!>
65. public <!CLASSNAME!(<!CONSTRUCTOR_PARAMLIST!>)
66. {
67. <!CONSTRUCTOR_ASSIGNMENTS!>
68. }
69. ?>
70. }
71. // Get and Set Operations
72. public <!COMPONENT_TYPE!> get(int pPosition)
73. {
74. return cValues[pPosition];
75. }
76. 
77. public void set (int pPosition, <!COMPONENT_TYPE!> pValue)
78. {
79. cValues[pPosition]=pValue;
80. }
81. 
82. <?<!COMPONENT_OPERATIONS!>
83. // Component Operations could be generated here
84. ?>
85. 
86. <?<!INNER_PRODUCT!>
87. // Inner Product could be generated here
88. ?>
89. 
90. // Number of Dimensions
91. int numberOfDimensions()
92. {
93. return <!DIMENSION!>;
94. }
95. 
96. 
97. }
98. .End Frame
99. 
100. .Function createVector(CLASSNAME, DIMENSION, COMPONENT_TYPE, COMPONENT_OPERATIONS, INNER_PRODUCT)
101. 
102. ."---------------- Defaults -------------------"
103. .If CLASSNAME = EMPTY Then
104. .CLASSNAME = "Vector"
105. .End If
106. .If DIMENSION = EMPTY Then
107. .DIMENSION = 3
108. .End If
109. .If COMPONENT_TYPE = EMPTY Then
110. .COMPONENT_TYPE = "int"
111. .End If
112. .If COMPONENT_OPERATIONS = EMPTY Then
113. .COMPONENT_OPERATIONS = "yes"
114. .End If
115. .If INNER_PRODUCT = EMPTY Then
116. .INNER_PRODUCT = "yes"
117. .End If
118. ."---------------------------"
119. 
120. .Dim aVector=CreateFrame("Vector")
121. .Dim I=0
122. 
123. .aVector.CLASSNAME=CLASSNAME
124. .aVector.DIMENSION=DIMENSION
125. .aVector.COMPONENT_TYPE=COMPONENT_TYPE
126. 
127. .aVector.COMPONENT_OPERATIONS=Condition(COMPONENT_OPERATIONS="yes")
128. .aVector.INNER_PRODUCT=Condition(INNER_PRODUCT="yes")
129. 
130. .aVector.N=Condition(DIMENSION="n")
131. .aVector.NOT_N=Condition(DIMENSION<>"n")
132. 
133. ."Loop-Unrolling
5.2 Using Frame Technology for Single Software-Systems

Listing 5.1 ANGIE code module for the generation of vector classes in Java (Vector.cdm)

Listing 5.2 Output of the ANGIE vector code module for Vector3D (Vector3D.java)

```java
class Vector3D {
    private float[] cValues;

    // Constructors
    public Vector3D() {
        cValues = new float[3];
        cValues[0] = 0;
        cValues[1] = 0;
        cValues[2] = 0;
    }

    public Vector3D(float pValue0, float pValue1, float pValue2) {
        cValues[0] = pValue0;
        cValues[1] = pValue1;
        cValues[2] = pValue2;
    }

    // Get and Set Operations
    public float get(int pPosition) {
        return cValues[pPosition];
    }

    public void set(int pPosition, float pValue) {
        cValues[pPosition] = pValue;
    }

    // Component Operations could be generated here

    // Inner Product could be generated here

    // Number of Dimensions
    int numberOfDimensions() {
        return 3;
    }

    // Component Operations could be generated here

    // Number of Dimensions
    int numberOfDimensions() {
        return 3;
    }

    // Inner Product could be generated here
```
Frame Technology in the Context of Generative Programming

```java
class Vector2D {
    private int[] cValues;

    // Constructors
    public Vector2D() {
        cValues = new int[2];
        cValues[0] = 0;
        cValues[1] = 0;
    }
    public Vector2D(int pValue0, int pValue1) {
        cValues[0] = pValue0;
        cValues[1] = pValue1;
    }

    // Get and Set Operations
    public int get(int pPosition) {
        return cValues[pPosition];
    }
    public void set(int pPosition, int pValue) {
        cValues[pPosition] = pValue;
    }

    // Component Operations could be generated here

    // Number of Dimensions
    int numberOfDimensions() {
        return 2;
    }
}
```

*Listing 5.3 Output of the ANGIE vector code module for Vector2D (Vector2D.java)*

If the dimension parameter of the “createVector”-Function is set to “n”, it is even possible to get a vector with dynamic dimensions, as Listing 5.4 shows.

```java
class DynamicVector {
    private double[] cValues;
    private int n; // Number of Dimensions

    // Constructors
    public DynamicVector(int pDimension) {
        n = pDimension;
        cValues = new double[n];
        for (int i = 0; i < pDimension; ++i) {
            cValues[i] = 0;
        }
    }

    // Get and Set Operations
    public double get(int pPosition) {
        return cValues[pPosition];
    }
    public void set(int pPosition, double pValue) {
        cValues[pPosition] = pValue;
    }
}
```

*Listing 5.4 Output of the ANGIE vector code module for DynamicVector (DynamicVector.java)*
5.2 Using Frame Technology for Single Software-Systems

A software factory can generate different variants of a software-system which belong to a specific family. Such a software-system can be an application, a large web-based client/server-system, a library for some specific purpose, and so on.

For demonstration purposes, a software factory that generates vector library variants is chosen. The example in the last chapter illustrates the possibilities of frame technology for single software-systems. The following example belongs to the same domain and shows how frame technology can be used for the construction of software-factories of a specific software-system family, in this case the family of vector libraries. The whole family range is specified in the feature diagram of Figure 5.4.

For an appropriate understanding, one has to consider the development of an application which needs an exactly suited library for vector calculations. It should be as well-suited as possible, because high performance is needed that can not be supplied by all-purpose libraries. A well-suited library would also abstain from unnecessary features, thus reducing complexity and raising maintainability.

There are a few choices, how to get and use one. It could be written from scratch, with the advantage of being very well-suited, but also with the disadvantage of a large time effort. An existing library could be taken and encapsulated in a wrapper layer [GHJ+95], thus hiding the unnecessary features and saving some development time at the cost of performance. It still would take some time to write the glue code, and the performance disadvantage is even increased in comparison to the direct use of a multi purpose vector library. On the other hand, a frame based software factory could be used to build exactly the needed library. Of course, this library has to exist first, so it has to be developed. Once created however, it can be used for many projects, with different requirements for vector calculations.

<table>
<thead>
<tr>
<th>StandardVector</th>
<th>UnitVector</th>
</tr>
</thead>
<tbody>
<tr>
<td>- cDimension: int</td>
<td>- cDimension: int</td>
</tr>
<tr>
<td>- cValue: int</td>
<td>- cOneValue: int</td>
</tr>
<tr>
<td>- getPosition: int: int</td>
<td>- getPosition: int: int</td>
</tr>
<tr>
<td>+ numberOfDimensions: int</td>
<td>+ numberOfDimensions: int</td>
</tr>
</tbody>
</table>

The question that comes to mind is: Why not just generate the appropriate classes like in single system use (see chapter 5.2). The answer is simple. Libraries provide more than mere classes. Complex structures might be needed that could be different for each variant.

Page 68
5 Frame Technology in the Context of Generative Programming

For example a standard vector and a unit vector could be required (see Figure 5.7). To have the unit vector as a separate class is useful, because is has to store only the position of the one-value instead of all values. For example, the vector with cValues[0]=0, cValues[1]=1, cValues[2]=0 ... cValues[7]=0 becomes the unit vector with cDimension=8 and cOneValue=1. Therefore a lot of memory can be saved, if there are many unit vectors in use.

However, a common base class would be useful for interoperability. For example, the ability to add a unit vector and a standard vector up to a new standard vector could be achieved this way. Even the sum of the unit vectors like (0,1,0) and (1,0,0) produces a vector which is not a unit vector, in this case (1,1,0). Therefore, the add and subtract-methods of the unit vector have to return a standard or common base vector and it would also be useful to accept an other vector than a unit vector as parameter, e.g. add(pVector:Vector):Vector with Vector being a common interface or base class.

For increased maintainability, it is also preferable that common methods, like the component operations in the example, are kept in one place using an object oriented technique (e.g. a pattern [GHJ+95]). The result is a library with an object oriented design as shown in Figure 5.8.

![Diagram](image)

Figure 5.8 A small vector library in UML

If the inner product method is needed in both vectors, the software factory should also generate the appropriate interface and command class. If it is only needed in one class, it could be produced in-line, without an interface. If there are specialized versions of the standard or unit vector, they should be derived from the matching class and so on. It is important that architecture and design exactly match the provided functionality and features for achieving the best performance (due to the abandonment of overhead and superfluous calls) and maintainability (due to the very right level of complexity).

To make a conclusion, frame technology is well fitted for the construction of software factories and therefore the context of software-system families, because of its construction-controlling, reuse and structuring capabilities.

---

20 Intra-variability use could of course be realized by adding a DERIVED_FROM or IMPLEMENTS variable (slot) holding an interface name or a class name, thus controlling the placement of the generated vector class in the class hierarchy. Therefore the unit vector could also be used. In a library however, the design thoughts are already integrated and should be reusable too.
Frames are a technology not widely known yet (see chapter 4 “History of Frame Technology”). Therefore it is obvious that there are not many tools available which already support this new technology. But those that are available are very promising. Each tool has its advantages and disadvantages. It is up to the reader and his requirements, to select the tool fitting best for his projects. The six tools (frame processors) examined here are (in alphabetical order) ANGIE, FrameProcessor, FPL, Netron Fusion, XFramer and the XVCL Frame Processor.

6.1 Frame Processors in Detail

6.1.1 ANGIE

The name ANGIE is a wordplay. There is the abbreviation NGE which means New Generator Engine (referring to its antecedent generator engines). If the letters are pronounced contiguously, they sound like the name ANGIE, and so it was called. [Gie02a]

ANGIE was developed by Delta Software Technology as a successor of the Prolog-based analysis and generation tool ROOT (see chapter 4 “History of Frame Technology”) [DST02a]. It was used as an internal tool and as back-end for other Delta Software Technology products (e.g. SCORE Integration Suite, PBE) [DST02b], but is now also available as a separate application (see www.d-s-t-g.com/GP).

The syntax of its frame commands is based on the ECMA-262 standard for script languages and therefore similar to Visual Basic [Gie02a]. The embedding of slots can be done with user-defined flags. This allows to use ANGIE with every target language without interfering with target language specific expressions, while keeping the code readable (for example <!aSlot!>). It has also a very similar technique for conditional statements. Beside the .if command which can be used for the control flow in scripts, it also has the user-defined optional code block flags (standard is <? and ?>>) for the usage inside frames. These flags bind the insertion of a code block on the state of definition of one or more frame variables (for an example see lines 64-69 of Listing 5.1). It is a very declarative way for specifying existence conditions.

In ANGIE, the alteration of slots within frames is also done in an state full way. In a script-function there may be expressions and assignments like in any procedural programming language. To address the slot of a specific frame, a syntax in the form frameInstanceName.slotName is used. Therefore the value of a slot can be changed like a public attribute of an object [DST02d].

ANGIE furthermore features a frame based type concept. Variables can be without a specific type (keyword: variant) or they can have the type of a specific frame, just like a class in a typed language. In the first case, the variable can hold everything, in the latter case it can hold only instances of the specific frame [DST02d]. In addition to these types there are collections (keyword collection). Collections are multi-set variables which can contain different elements. The contained elements can even be indexed by keys [DST02d] like a map (see [Str97]) in the STL.

The frame commands that are available in script-functions have everything a full featured programming language should provide. Additionally, there are library functions [DST2b] for a wide range of tasks, such as retrieving the current date or working with a database (during construction time). For an overview of ANGIE commands see Table 6.1.
6.1 Frame Processors in Detail

<table>
<thead>
<tr>
<th>ANGIE Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Frame aFrame</td>
<td>Represents a frame.</td>
</tr>
<tr>
<td>... .End Frame</td>
<td></td>
</tr>
<tr>
<td>&lt;!aSlot!&gt;</td>
<td>Embedding of slots. Please notice that &lt;! and !&gt; are user defined.</td>
</tr>
<tr>
<td>&lt;? OptionalCodeBlock ?&gt;</td>
<td>Optional code block, becomes inserted depending on the state the variables. Please notice that &lt;? and ?&gt; are user defined.</td>
</tr>
<tr>
<td>.Dim aVariable as aType</td>
<td>Variable definition</td>
</tr>
<tr>
<td>.Function aFunction as aType</td>
<td>Represents a script function.</td>
</tr>
<tr>
<td>... .End Function</td>
<td></td>
</tr>
<tr>
<td>.Export (aFrameInstance, aFilename, Filetype)</td>
<td>Exports a frame instance to a file of a given type (e.g. Java, HTML, Text, and so on).</td>
</tr>
<tr>
<td>.aFrameInstance=CreateFrame(“aFrame”)</td>
<td>Creates an instance of the frame aFrame and assigns it to the variable aFrameInstance.</td>
</tr>
<tr>
<td>.add( different Parameters here )</td>
<td>Add an element to a Collection (type of slot)</td>
</tr>
<tr>
<td>=, +, -</td>
<td>Typical arithmetic operators</td>
</tr>
<tr>
<td>.If aCondition Then</td>
<td>Conditional control structure</td>
</tr>
<tr>
<td>... .Else</td>
<td></td>
</tr>
<tr>
<td>... .End If</td>
<td></td>
</tr>
<tr>
<td>.For aVariable=aValue to anotherValue</td>
<td>Iterative control structure</td>
</tr>
<tr>
<td>... .Next</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1 Important commands of ANGIE

ANGIE furthermore provides an external interface for integration with other software-systems. A repository can be used for data-storing and configuration purposes during construction processes [DST02b, p. 8]. These repositories can also be XML-based. ANGIE has even a control interface for direct linking with other software-systems working as front-ends [DST02b, p. 9].

![Figure 6.1 The ANGIE architecture](Adapted from [DST02b])
6 Available Frame Processors

**ANGIE** is based on a two layer architecture. One layer is responsible for compilation, the other for interpretation. The compiler takes the **ANGIE** source code and produces an intermediate code which will be interpreted subsequently. The following interpretation processes can be controlled by the before mentioned interface. The **ANGIE** Compiler is the actual frame processor while the **ANGIE** Interpreter is part of the custom generator. Figure 6.1 shows a graphical overview of the concept [DST02b] (see also 3.2.2.2 “What is the Generator in Context of the Abstraction Concept?”).

**Advantages**
+ frame commands provide a full featured programming language
+ support for optional code blocks
+ user-definable slot embedding
+ support for library functions

**Disadvantages**
- only available for the windows operating system

**Projects that apply ANGIE**
- **HUTN-Parser Project** by M. Schönwald [Gie02b; Soe02]
- **HyperSenses** by Delta Software GmbH [Gie02b; Sch03b]
- **Pattern By Example** [DST02c]
- **Project GP-WEB**, a web-based e-learning software-system family [BEH+02]
- Project-specific generators in a **SCORE**-project at **SUVA** company [Gie02b]
- **Root** (**ANGIE**’s predecessor) was already used for year 2000 problems [DST02a]
- **SCORE** [DST02b]

**Further Reading**
For more information about **ANGIE** see [BEH+02], [DST01], [DST02a], [DST02b], [DST02c], [DST02d], [ES02], [Sch01a], [Sch01b] and [Sch02].

---

21 Human-Usable Textual Notation (HUTN) is an upcoming standard of the OMG [OMG].
6.1 Frame Processors in Detail

6.1.2 FPL

The FPL (Frame Processing Language) was developed by Frank Sauer. FPL is the name of the language as well as the name of the frame processor that interprets this language. FPL is available as plugin for Eclipse, which is a well known modular IDE for Java and other programming languages (see Figure 6.2). Eclipse was originally developed by IBM and is now available as a free open source product [Ecl]. As an Eclipse-plugin, FPL is written in Java while the language it interprets is XML-based. Most of the frame commands are therefore XML-tags. Table 6.2 shows the important commands (for more details see [Sau02]).

<table>
<thead>
<tr>
<th>FPL Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;frame name=&quot;aFrameName&quot; language=&quot;aLanguage&quot;&gt; ... &lt;frame&gt;</td>
<td>Represents the frame aFrameName and specifies the language aLanguage as the target language.</td>
</tr>
<tr>
<td>&lt;break name=&quot;aSlot&quot;&gt; ... &lt;/break&gt;</td>
<td>Marks a point or a block of default code in the actual frame, where custom code can be inserted, default code overwritten, and so on. It creates the slot with the name aSlot.</td>
</tr>
<tr>
<td>&lt;adapt frame=&quot;aFileName1&quot; outfile=&quot;aFilename2&quot;&gt; ... &lt;/adapt&gt;</td>
<td>Copies (adapts) the frame with the file name aFileName1 to the actual frame. This means the frames become connected. In the enclosed block customization commands (inserts) are allowed. Also triggers the export of the actual frame to the file aFilename2.</td>
</tr>
<tr>
<td>&lt;replace break=&quot;aSlot&quot;&gt; ... &lt;/replace&gt;</td>
<td>This customization command replaces the code of the matching breakpoint with custom code.</td>
</tr>
<tr>
<td>&lt;remove break=&quot;aSlot&quot;&gt; ... &lt;/remove&gt;</td>
<td>This customization command removes the code, enclosed by the matching breakpoint.</td>
</tr>
<tr>
<td>&lt;add-after break=&quot;aSlot&quot;&gt; ... &lt;/add-after&gt;</td>
<td>This customization command inserts custom code after the matching breakpoint.</td>
</tr>
</tbody>
</table>
6 Available Frame Processors

<table>
<thead>
<tr>
<th>FPL Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;add-before break=&quot;aSlot&quot;&gt;</code>...<code>&lt;/add-before&gt;</code></td>
<td>This customization command inserts custom code before the matching breakpoint.</td>
</tr>
<tr>
<td><code>&lt;around break=&quot;aSlot&quot;&gt;</code>...<code>&lt;/around&gt;</code></td>
<td>Combines the functionality of <code>add-before</code> and <code>add-after</code>.</td>
</tr>
<tr>
<td><code>&lt;set var=&quot;aVariable&quot; value=&quot;anExpression&quot;/&gt;</code></td>
<td>Sets the variable <code>aVariable</code> to the value of <code>anExpression</code>.</td>
</tr>
<tr>
<td><code>&lt;set var=&quot;aMultiVariable&quot; list=&quot;element1, element2, ...&quot;&gt;</code></td>
<td>Defines a multi-set variable.</td>
</tr>
<tr>
<td><code>${aVariable}</code></td>
<td>Inserts the value of the variable <code>aVariable</code> in the target language code or in the frame code.</td>
</tr>
<tr>
<td><code>&lt;select var=&quot;aVariable&quot;&gt;</code>...<code>&lt;/select&gt;</code></td>
<td>Conditional control structure. Code between the when-tags will be inserted, if the the variable <code>aVariable</code> matches the value <code>aValue</code>, according to the used comparator. An when-defined, when-undefined, and otherwise tag is also available.</td>
</tr>
<tr>
<td><code>&lt;while listvars=&quot;multivariable1, multivariable2, ...&quot;&gt;</code>...<code>&lt;/while&gt;</code></td>
<td>Iterative control structure. The enclosed Code becomes inserted for each value in one of the multi-set variables.</td>
</tr>
</tbody>
</table>

Table 6.2 Important commands of FPL

**Advantages**
+ free & open source
+ XML-based

**Disadvantages**
- only available as Eclipse-Plugin

**Projects that apply FPL**
There are none known yet.

**Further Reading**
More information about FPL can be found in [Sau02].

6.1.3 FrameProcessor

FrameProcessor was developed by Thomas Patzke with the intention of creating a very small, light-weight, easy to handle and easy to learn frame processor, supporting a minimal set of frame commands of the adapt concept. He published the first version (0.6) in June 2002 on Sourceforge.net ([www.sourceforge.net](http://www.sourceforge.net)), where it can be downloaded for free as an open source application. Therefore, FrameProcessor can be extended by anybody, if the need for additional functionality arises. The actual version 0.7 (in July 2002) includes an experimental GUI and a tool for checking legacy applications for compatibility with the frame commands. FrameProcessor is written in Phyton while the GUI uses Tk. Therefore FrameProcessor is independent from a specific operating system. Table 6.3 shows the commands of FrameProcessor [Pat02]. The Listings 6.1 to 6.3 show examples of FrameProcessor code.
6.1 Frame Processors in Detail

### FrameProcessor Command Description

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTFILE filename</td>
<td>Specifies the file name for the code output of a frame.</td>
</tr>
<tr>
<td>VP aSlot</td>
<td>Marks a point or a block of default code in the actual frame, where custom code can be inserted, default code overwritten, and so on. It creates the slot (variation point) with the name aSlot.</td>
</tr>
<tr>
<td>ADAPT aFileName</td>
<td>Copies the frame with the file name aFileName to the actual frame. This means the frames become connected. In the enclosed block customization commands (inserts) are allowed.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>This customization command replaces the code of the matching variation point with custom code.</td>
</tr>
<tr>
<td>INSERT_AFTER</td>
<td>This customization command inserts custom code after the matching variation point.</td>
</tr>
<tr>
<td>INSERT_BEFORE</td>
<td>This customization command inserts custom code before the matching variation point.</td>
</tr>
</tbody>
</table>

#### Table 6.3 Commands of FrameProcessor

#### Listing 6.1 Example for a FrameProcessor frame with a variation point (main.frame)
```c
void main() {
    VP exampleVp
    printf("This is the common part\n");
    VP_END
}
```

#### Listing 6.2 Example for a FrameProcessor frame, adapting "main.frame" (main.spc)
```c
ADAPT main.frame
INSERT_BEFORE exampleVp
    printf("This code is executed first\n");
   }
```

#### Listing 6.3 Output from frame "main.spc" of Listing 6.2 (main.c)
```c
void main()
{
    printf("This code is executed first\n");
    printf("This is the common part\n");
}
```

#### Advantages
- independent of a specific operating system
- free & open source
- easy to learn

#### Disadvantages
- no control structures
- no slots for variables

#### Projects that apply FrameProcessor
There are none known yet.

#### Further Reading
For further reading, see the README-file [Pat02] enclosed in the package.
6 Available Frame Processors

6.1.4 Netron Fusion

Netron Fusion is commercially available and sold by Netron Inc. [Net02b]. It was developed by Netron Inc. in 1986 [Net97a] for use in COBOL environments. Like other frame processors, it is able to produce code for any target language. It is however tightly coupled with COBOL [WJS+01]. It comes with a tool suite [Net02b] which is strongly COBOL-centric [Net97a; WJM+01] and generated systems might need to link COBOL DLLs for execution [Net97b, p. 7]. Most of the tools work only with COBOL [WJM+01, p. 2(165)] and some of Netron Fusion’s commands provide special parameters for COBOL only [Net97a].

It supports the adapt concept (see 3.2.1 “The Adapt Concept”), using commands with a COBOL-like syntax. Table 6.4 summarizes the most important commands of Netron Fusion [Net97a;WJM+01]. The other Commands are .KEYWORD, .LANGUAGE, .QUOTE, .COMMENTS, .GROUP, .REPLACEON, .REPLACEOFF and .ERROR [Net97a, p. TR-449]. A detailed explanation of all these commands however, would go beyond the scope of this survey. For a code example see Listing 6.4.

<table>
<thead>
<tr>
<th>Netron Fusion Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame</td>
<td>Represents a frame.</td>
</tr>
<tr>
<td>.BREAK aSlot</td>
<td>Marks a point or a block of default code in the actual frame, where custom code can be inserted, default code overwritten, and so on. It creates the slot with the name aSlot.</td>
</tr>
<tr>
<td>.COPY aFileName</td>
<td>Copies the frame with the file name aFileName to the actual frame. This means the frames become connected. In the enclosed block customization commands (inserts) are allowed.</td>
</tr>
<tr>
<td>.INSERT</td>
<td>This customization command replaces the code of the matching breakpoint with custom code.</td>
</tr>
<tr>
<td>.INSERT-AFTER</td>
<td>This customization command inserts custom code after the matching breakpoint.</td>
</tr>
<tr>
<td>.INSERT-BEFORE</td>
<td>This customization command inserts custom code before the matching breakpoint.</td>
</tr>
<tr>
<td>.REPLACE aVariable BY anExpression</td>
<td>Sets the variable aVariable to the value of anExpression.</td>
</tr>
<tr>
<td>;aVariable</td>
<td>Inserts the value of the variable aVariable in the frame.</td>
</tr>
<tr>
<td>.SELECT ;aVariable</td>
<td>Conditional control structure. Code after .WHEN becomes inserted, if aVariable operator aValue is true, and so on. If no .WHEN or .ORWHEN is executed, the code after .OTHERWISE becomes inserted. The default operator is =.</td>
</tr>
<tr>
<td>.WHEN [operator] aValue</td>
<td></td>
</tr>
<tr>
<td>.ORWHEN [operator] aValue2</td>
<td></td>
</tr>
<tr>
<td>.OTHERWISE</td>
<td></td>
</tr>
<tr>
<td>.END-SELECT ;aVariable</td>
<td></td>
</tr>
<tr>
<td>.WHILE ;anExpression</td>
<td>Iterative control structure. The enclosed code becomes inserted as long as anExpression is true.</td>
</tr>
<tr>
<td>.END-WHILE</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4 Important commands of Netron Fusion

(Based on [WJS+01])
6.1 Frame Processors in Detail

Listing 6.4 Example of a Netron Fusion frame

```java
QUERY_LOANS_UI.F // user interface frame to display loan query methods
class QueryLoans extends LibPanels implements ActionListener {
    QueryLoans (LibClient client){ // constructs a panel to show loan query methods
        queryLoansMethodLst = new List ();
        queryLoansMethodLst.addMethod ( by Item ); //QUERY_LOANS_BY_ITEM – mandatory
        queryLoansMethodLst.addMethod ( by Member ); //QUERY_LOANS_BY_MEMBER – mandatory
        // anticipated variant loan query methods are defined below:
        . WHILE ;QUERY_METHOD_CHOICE; N
            . SELECT ;QUERY_METHOD_CHOICE;N
            . WHEN query_loans_by_overdue //QUERY_LOANS_BY_OVERDUE-OR1
                queryLoansMethodLst.addMethod( by Overdue Items );
                . WHEN query_loans_by_reserved //QUERY_LOANS_BY_RESERVED-OR2
                    queryLoansMethodLst.addMethod( by Reserved Items );
            . END-SELECT ;QUERY_METHOD_CHOICE; N
            R N BY (\$N + 1)
        END-WHILE
    }BREAK QUERY_LOANS_METHODS // this breakpoint caters for unexpected new variants
    // new unexpected loan query methods can be defined here
    .END-BREAK QUERY_LOANS_METHODS
}
```

Netron Fusion is shipped along with a large suite of tools supporting frame-based development. It features the tools Workplace, Wizard, Object Manager, Data Manager, Designer, Screen Editor, Report Editor, FAUST, Build Utility, Frame Tree Utility, Processor and Graphical Debugger. Workplace, a graphical IDE, is the main application for creating and editing frames. The Wizard can be used to build frameworks based on predefined templates. The Object Manager's task is to organize reusable objects (entities) which may be frames, subroutines or other components. External data definitions can be manged with the Data Manager. The Designer and Screen Editor are for designing GUI-Elements and layouts, while the Report Editor can be used for the layout of reports. FAUST, which means Frame Analysis and Usage Statistics Tool, is used to evaluate the quality of the reuse level. The Build Utility can be compared to a classical make tool, however specialized for frames. The Frame Tree Utility displays the frame hierarchy of a system. The Processor is the actual frame processor itself, generally producing COBOL code, and the Graphical Debugger can be used to trace the construction process. Besides these tools Netron Fusion comes along with a library of ready made frames for different tasks in COBOL and a library containing templates of application frameworks [Net02b; Net97a, p. TR-4].

Advantages
+ large suite of tools for different tasks

Disadvantages
- tightly coupled with the COBOL language

Projects that apply Netron Fusion
- Billing System for Motorola Telco [Net02b]
- Complete Solution for the Barclay Bank (Barclay Merchant Services - BMS) [Net02b]
- CICS on-line mainframe system [Bas97, p. 67]
- many more (mostly in COBOL) [Bas97, p. 5]
6 Available Frame Processors

Further Reading
For more information about Netron Fusion see [Bas97], [Jar01], [JOZ01], [JS00], [Net97a], [Net02b] and [WJS+01].

6.1.5 XFramer
As already mentioned, XFramer is inspired by ANGIE and is therefore the second realization of an abstraction concept-based frame processor [Sle]. It was developed by Max Schlee and is available as freeware for the operating systems Linux and Windows [Sle]. XFramer can not be considered a frame processor itself. It is a preprocessor that becomes a frame processor when used in combination with a compiler [ES03]. The “X” means the language XFramer is used with. If XFramer is used with Java, it can be called JFramer. In the case of C++, it is called CFramer, and so on. The languages supported by the version 1.45 of XFramer are Java, C++ and C#. Further languages might be added later. The language used with XFramer is called the frame code language. The target code language on the other hand is not limited to a supported set of programming languages. It can be any programming language, markup language, natural language, and so on. This is of course standard for all frame processors. If the same language is used for frame code and target code at the same time, it has the advantage that the same functions can be used for the generator as well as the generated code [ES03].

The advantage of using an existing object-oriented programming language as frame code language is that no new language has to be learned and various software engineering methods (e.g. OOA and OOD) can be applied not only to the generated source code, but to the generator as well. It is possibly a disadvantage that object-oriented languages are not optimized for generation purposes. There are features in object-oriented languages which might be not useful for generation. Therefore the complexity could be unnecessarily increased. On the other hand, it also could lack features which are beneficial for generation purposes. However, which features of an object-oriented language are also useful during generation time and which are not is still subject to research. It could as well be possible that an object-oriented language is already very well-suited for generation purposes.

Figure 6.3 shows the generation process with XFramer. First, the specification (see Listing 6.5 for an example), consisting of frame code and target language code, is processed by XFramer. The result (see Listing 6.6 for an example) is a file that contains code in the frame code language (Java, C++ or C#). The code blocks of the target language have been converted to inlined string literals. Thereafter the file is processed by the appropriate compiler, resulting in an executable binary file. This file is a generator. If executed, it can process configuration data for further customization and finally outputs the source code in the target language (see Listing 6.7 for the appropriate example).
In XFramer, the target language code blocks are embedded within the frame code. For embedding there are two specific frame commands. `BEGIN_FRAME_TEXT()` marks the beginning of a target language block, while `END_FRAME_TEXT()` marks the end of it. The slots embedded inside the target language block are variables of the frame code language (i.e. attributes, local variables or parameters). Figure 6.4 shows the complete anatomy of a frame in XFramer.

The principle of XFramer’s generation process has the unique advantage that the generation process itself is fully debuggable by a debugger of the used frame code language. This principle further allows to use any library - that is available for the frame code language - during the generation process. For example, XML files can be processed with the Xerces library [ASF]. Multi-set variables can also be realized this way. The standard template library (STL) of C++, for example, provides the generic multi-set templates Vector and Map [Str97]. Another special feature of XFramer is the embedding of binaries (see Table 6.5 for the syntax). This allows to also generate non-text artifacts like media files or special protocol data. Please notice that all embedding flags (e.g. the slot-flags <! !>) can be changed by the user.

```java
frame FrSwap {
    FrSwap(String T) {
        BEGIN_FRAME_TEXT()
        void swap(<! T!>& a, <! T!>& b) {
            <! T!> c = a;
            a = b;
            b = c;
        }
        END_FRAME_TEXT()
    }
    // ======================================================================
    public class SwapGenerator {
        public static void main(String argv[]) {
            FrSwap frSwap = new FrSwap("int");
            frSwap.exportToFile("swap.cpp");
        }
    }
```

Listing 6.5 C++ Swap-Generator in XFramer using Java as frame code language

(Based on [ES03])
6 Available Frame Processors

```java
FrSwap(String T) {
    setFrameText("166"
        + MString.toString(T) + "46"
        + MString.toString(T) + "40"
        + MString.toString(T) + "40"
        + MString.toString(T) + "40"
        + MString.toString(T) + "40"
        + "143" + "40" + "40" + "143" + "141"
    );
}

// ======================================================================
public class SwapGenerator {
    public static void main(String argv[]) {
        FrSwap frSwap = new FrSwap("int");
        frSwap.exportToFile("swap.cpp");
    }
}
```

Listing 6.6 Swap-Generator already processed by XFramer (SwapGenerator.java)

```c
void swap(int & a, int & b) {
    int c = a;
    a = b;
    b = c;
}
```

Listing 6.7 A function for swapping integer values in C++ (swap.cpp)

<table>
<thead>
<tr>
<th>XFramer syntax (Java version)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame aFrame { ... }</td>
<td>Represents a frame.</td>
</tr>
<tr>
<td><code>&lt;aSlot!&gt;</code></td>
<td>Embedding of slots. Please notice that <code>&lt;!</code> and <code>!&gt;</code> are user redefineable.</td>
</tr>
<tr>
<td>private/public aType aVariable;</td>
<td>Variable definition</td>
</tr>
<tr>
<td>private/public aType aMethod { ... }</td>
<td>Represents a method.</td>
</tr>
<tr>
<td>aFrameInstance.exportToFile(aFilename );</td>
<td>Exports a frame instance to a file of a given type (e.g. Java, HTML, plain text, and so on).</td>
</tr>
<tr>
<td>aFrameInstance = new aFrame;</td>
<td>Creates an instance of the frame aFrame and assigns it to the variable aFrameInstance.</td>
</tr>
<tr>
<td>if (aCondition) { ... } else { ... }</td>
<td>Conditional control structure</td>
</tr>
</tbody>
</table>
### 6.1 Frame Processors in Detail

#### XFramer syntax (Java version)

<table>
<thead>
<tr>
<th>XFramer syntax (Java version)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (anInitialization; aCondition; anIncrementExpression) { ... }</td>
<td>Iterative control structure</td>
</tr>
<tr>
<td><code>&lt;# aFile #&gt;</code></td>
<td>Embedding of external files (including binaries)</td>
</tr>
</tbody>
</table>

*Table 6.5 Important commands of XFramer*

#### Advantages

- frame commands are derived from a full-featured programming language
- user-definable slot embedding
- support for library functions of the frame code language
- debugging of the generator is possible
- support for binary components

#### Disadvantages

- no optional code blocks
- used frame code language not optimized for generation purposes

#### Projects that apply XFramer

- *Mini-ABA* [EEE+03b]

#### Further Reading

To start working with XFramer, read the details in [ES03].
6 Available Frame Processors

6.1.6 XVCL processor

XVCL means “XML-based Variant Configuration Language” [NN01, p. 4] and is the pure language specification, while the frame processor using this language is called the XVCL processor [NN01, p. 4], and sometimes referred as the x-frame processor [WJS+01]. Frames in XVCL are respectively called x-frames (or XVCL-frame). It was developed by the National University of Singapore, and it is available to download for free from the XVCL web-page on Sourceforge.net [Nus02]. It supports the adapt concept of frame technology based on [Bas97] (according to [Nus02l]). XVCL is XML-based [NN01] and therefore comes with a fully specified DTD containing the description of the frame commands and the composition of its documents. Therefore developers familiar with XML should learn the XVCL syntax fast. A further advantage is that an editor capable of parsing a DTD, e.g. jEdit (see [Jed]) can provide editing enhancements such as syntax-highlighting, syntax-checking, tag-completion, and so on (see Figure 6.5).

The disadvantage is that the code may become cluttered with long tags, therefore making it harder to read. Especially the tag for inserting variables is very long. For example the line

<value-of expr=”?@VECTOR_CLASSNAME?” />

could be written in ANGIE and XFramer as

<!VECTOR_CLASSNAME>

and in Netron Fusion as

VECTOR_CLASSNAME.
6.1 Frame Processors in Detail

Luckily, the cluttering can be largely avoided - therefore keeping the code readable - by syntax-highlighting and a disciplined programming style.

The **XVCL processor** was developed in Java and uses the open source library **JAXP**, which is a fast and cost efficient solution for parsing the XML-based code [WJS+01]. Because of being a Java-application, the **XVCL processor** is also independent from specific operating systems.

An overview of important frame commands is provided by Table 6.6, structuring the information from [NN01].

<table>
<thead>
<tr>
<th><strong>XVCL Command</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;x-frame name=“aFrameName”</code>&lt;br&gt;<code>outfile=“aFilename”&gt;</code></td>
<td>Represents the frame <code>aFrameName</code> and specifies the file name <code>aFilename</code> for the code output of the frame.</td>
</tr>
<tr>
<td><code>&lt;/x-frame&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;break name=“aSlot”&gt;</code></td>
<td>Marks a point or a block of default code in the actual frame, where custom code can be inserted, default code overwritten, and so on. It creates the slot with the name <code>aSlot</code>.</td>
</tr>
<tr>
<td><code>&lt;/break&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;adapt x-frame=“aFilename”&gt;</code></td>
<td>Copies (adapts) the frame with the file name <code>aFilename</code> to the actual frame. This means the frames become connected. In the enclosed block customization commands (inserts) are allowed.</td>
</tr>
<tr>
<td><code>&lt;/adapt&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;insert break=“aSlot”&gt;</code></td>
<td>This customization command replaces the code of the matching breakpoint with custom code.</td>
</tr>
<tr>
<td><code>&lt;/insert&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;insert-after break=“aSlot”&gt;</code></td>
<td>This customization command inserts custom code after the matching breakpoint.</td>
</tr>
<tr>
<td><code>&lt;/insert-after&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;insert-before break=“aSlot”&gt;</code></td>
<td>This customization command inserts custom code before the matching breakpoint.</td>
</tr>
<tr>
<td><code>&lt;/insert-before&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;set-var=“aVariable” value=“anExpression”&gt;</code></td>
<td>Sets the variable <code>aVariable</code> to the value of <code>anExpression</code>.</td>
</tr>
<tr>
<td><code>&lt;/set-var&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;value-of expr=“@?aVariable?”&gt;</code></td>
<td>Inserts the value of the variable <code>aVariable</code> in the frame.</td>
</tr>
<tr>
<td><code>&lt;/value-of&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;select option=“anExpression”&gt;</code></td>
<td>Conditional control structure. Code between the option-tags will be inserted if the two Expressions are equal. There is support for complex expressions and different operators (instead of “=”) by the condition-attribute. An option-defined, option-undefined, and otherwise tag is also available.</td>
</tr>
<tr>
<td><code>&lt;option value=“anotherExpression”&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;/option&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;while using-items-in=“@?aMultiVariable?”&gt;</code></td>
<td>Iterative control structure. The enclosed Code becomes inserted for each value in the multi-variable <code>aMultiVariable</code>.</td>
</tr>
<tr>
<td><code>&lt;/while&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

In addition to the the commands **XVCL** provides means of indirection. This can be done by assigning a variable name as value to another variable. With the @-operator, variables can be dereferenced [NN01].

Another special feature is the capability to use multi-variables. They behave like a multi-set, because they can contain multiple values. This is useful in conjunction with the while command, which executes the enclosed frame commands (or inserts target code) for each value contained in the specified multi variable (like a for each in some programming languages). On the other hand, standard iterations by supplying a range are not possible (like for(i=0;i<=10;++i) in C++ or Java). There is however a way to emulate this behavior by the means of multi-variables. Listing 5.1 (**ANGIE**) uses an iteration of this form to do a loop unrolling for the creation of vector

---

22 **FrameProcessor** does not support variables.
6 Available Frame Processors

classes in the lines 137 and 141. If the dimension of the vector class to be created is already known, there is no need for a time consuming runtime iteration. The appropriate lines could be created by the frame processor during construction time. Instead of

```java
for (int i=0; i<=n; ++i)
    values[i]=0;
```

with n=3, the direct code

```java
values[0]=0;
values[1]=0;
values[2]=0;
```

would save some time in large software-systems. The same could be done in XVCL. Listing 6.8 is the XVCL version of the vector class generating ANGIE frame hierarchy from Listing 5.1. The loop unrolling is done by assigning all values of the range (in this case 1,2 and 3) to a multi variable (see line 13) and then doing an iteration over this multi variable (see line 42).

```
<xml version="1.0"/>
<!DOCTYPE x-frame SYSTEM "file:///c:\xvcl_1.0_beta2\dtd\xvcl_1_0.dtd">
<!--
Version:  01.00.00
Category: Frame
Project:  Vector Example
Author:   Marco Emrich
-->
<x-frame name="Vector" language="java">
  <!-- Defaults -->
  <set var="VECTOR_CLASSNAME" value="Vector"/>
  <set var="DIMENSION" value="3"/>
  <set-multi var="DIMENSION_LIST" value="0,1,2"/>
  <set var="COMPONENT_TYPE" value="int"/>
  <set var="TYPE" value="Standard"/>

  <break name="VECTOR_PARAMETERS" />

  <set var="DIMENSION_MINUSONE" value=?@DIMENSION?-1/>
  class <value-of expr=?@VECTOR_CLASSNAME?/>
  {
    private <value-of expr=?@COMPONENT_TYPE? />[] cValues;

    private int n; //Number of Dimensions

    <select option=?@DIMENSION?><option value="n" condition="=">
    for (int i=0;i<=<value-of expr=?@DIMENSION? >;++i)
        {
            <option>
               cValues[<value-of expr=?@DIMENSION_LIST? >]=0;
            </option>
        }
    <select option=?@DIMENSION?><option value="n" condition="!=">}</select>
  </select>

  public <value-of expr=?@VECTOR_CLASSNAME? />
  {
  ...}
</x-frame>
```
6.1 Frame Processors in Detail

Listing 6.8 A XVCL-frame for the generation of vector classes in Java (Vector.XVCL)

Advantages
+ independent of a specific operating system
+ free & open source
+ XML-based

Disadvantages
- While operates only on multi variables
- code might be cluttered with tags

Projects that apply XVCL
- Computer Aided Dispatch System product line [Won00] [Jar02]
- Reimplementation of the Java Buffer Library [Jar02]
- An exemplary notepad [Nus02]

Further Reading
More information can be found in the XVCL specification [NN01] and its web-site [Nus02]. The papers [CJ99], [JS00], [JZ01], [WJS+01] and [ZJS01] can also be found online.
6 Available Frame Processors

6.2 Comparison of Available Frame Processors

Each frame processor has its advantages and disadvantages. Which one is best suited, depends on the project for which the frame processor should be used and the project's environment. Table 6.7 Quick comparison of available frame processors provides an overview to support the decision.

<table>
<thead>
<tr>
<th>Name</th>
<th>ANGIE</th>
<th>FPL</th>
<th>FrameProcessor</th>
<th>Netron Fusion</th>
<th>XFramer</th>
<th>XVCL processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor/developer</td>
<td>Delta Software Technology</td>
<td>Frank Sauer</td>
<td>Thomas Patzke</td>
<td>Netron Inc.</td>
<td>Max Schlee</td>
<td>National University of Singapore</td>
</tr>
<tr>
<td>available as</td>
<td>commercial and free</td>
<td>free &amp; open source</td>
<td>free &amp; open source</td>
<td>commercial</td>
<td>free</td>
<td>free &amp; open source</td>
</tr>
<tr>
<td>tested version</td>
<td>V 0.85 free and V 2.3 commercial</td>
<td>V 0.9.1</td>
<td>V 0.7</td>
<td>V 3.2</td>
<td>V 1.45</td>
<td>V 2.0 beta</td>
</tr>
<tr>
<td>OS required</td>
<td>Windows</td>
<td>OS independent</td>
<td>OS independent</td>
<td>Windows</td>
<td>Linux or Windows</td>
<td>OS independent</td>
</tr>
<tr>
<td>target language</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any/COBOL *</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>target OS</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>Windows NT, 9x, 3.x, OS/2, CICS, IMS, OS/400, UNIX, Open VMS</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>syntax</td>
<td>similar to Visual Basic</td>
<td>XML-based</td>
<td>own</td>
<td>similar to COBOL</td>
<td>extended C++, C# or Java</td>
<td>XML-based</td>
</tr>
<tr>
<td>implementation technology</td>
<td>ANSI C</td>
<td>Java</td>
<td>Python</td>
<td>COBOL</td>
<td>C++, C# and Java</td>
<td>Java</td>
</tr>
<tr>
<td>supported concept</td>
<td>abstraction</td>
<td>adapt</td>
<td>adapt</td>
<td>adapt</td>
<td>abstraction</td>
<td>adapt</td>
</tr>
<tr>
<td>control structures</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>frame variables</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>multi-set variables</td>
<td>yes: collections</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes: depends on frame code language</td>
<td>yes: multi-variables</td>
</tr>
<tr>
<td>is shipped with</td>
<td>libraries with many external functions (including XML processing)</td>
<td>-</td>
<td>IDE, testing tool for compatibility</td>
<td>IDE, various tools</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

*any language is possible, but it is optimized for COBOL*
Software-systems can be developed with the aid of object-oriented programming or frame technology or both of them. For a direct comparison of these possibilities, the example of a vector class library from chapter 5 “Frame Technology in the Context of Generative Programming” is used. As stated above, the goal is to create a vector library which provides all functionality, stated in the appropriate feature diagram (see Figure 5.4). It is important that all legal configurations of vector classes are possible and some kind of interoperability between them is guaranteed.

In fact, this goal can be reached by applying inclusion and overloading polymorphism [Eis99]. However, with object-oriented programming alone, it is not possible to automatically obtain different vector library variants. This means, instead of creating a family of software-systems for the different configurations, taken from the feature diagram, one system has to be built which includes all possible configurations at once. Therefore, the library is not specialized for one kind of application, but instead provides the whole bunch of functionality. This functionality becomes selected during runtime instead of construction time. An implementation of such a vector library in Java is shown in the UML-diagram of Figure 7.1. Please notice that the exchange ability of the component type is missing in this implementation. Only float is used here as a component type.

In a language with support for templates (e.g. C++) or a template-similar concept (generic polymorphism [Eis99]), the exchange ability can be easily added using a generic data-type. Java does currently not have such a concept, but the capability of different component types can be achieved by the application of some special object-oriented techniques. For example, the bridge pattern [GHJ+95] could be applied to encapsulate the component type. It has, however, been left out to avoid further cluttering of the diagram.

Interoperability between the different vector classes is ensured by the implementation of the vector interface. The user of the library has the ability to choose which functions are supported by selecting the appropriate vector class. The prototype of the different methods is guaranteed to be compatible due to the implementation of the support-interfaces. The implementation itself, however, has been moved to aggregated utility classes. Therefore, there is no need for the vector classes to implement the same algorithms themselves over and over. They just delegate the calls to the responsible utility-classes and provide their encapsulated data. The redundancy is kept as low as possible. Additionally, the dimension of the different vector classes has been made variable as demanded by the feature diagram. Please notice that it is possible to further optimize the performance of the library by providing specialized algorithms for unit vectors in additional utility classes.

As can be seen, the object-oriented techniques apply well to the design of the vector library. Despite this, there are also some disadvantages in comparison to the frame solution of chapter 5.

In an all object-oriented solution, the decisions which vector classes have to be used are made during runtime. This is, however, not necessary. Before the start of a software-system using the vector library it is already known which specific vectors and functionalities are needed. Therefore, the all object-oriented solution contains software components which are dispensable. This leads to the disadvantage of increased memory usage, unnecessary complexity and a longer command chain. As everyone knows, overly complex systems are also difficult to maintain. The command chain is longer due to the unnecessary classes, methods and general runtime decisions in the hierarchy. The
Figure 7.1 An all object-oriented implementation of a vector library as UML-classdiagram
result is a decreased performance. By the application of frame technology on the other hand, decisions that can be made before runtime are moved to construction time. This results in a customized library containing only the required vector components (see section 5.3 “Using Frame Technology for Software-System Families”). Another disadvantage of not using frame technology (in this case generator techniques in general) is that no specialized optimizations can be applied. In the vector library, this is exemplified by loop unrolling (see section 6.1.6 “XVCL processor”). In Listing 5.1, lines 137-140 perform loop unrolling for the null-assignments in the vector class constructor of standard vectors, but it could also be applied in other appropriate places.

The code for the addition of vectors in the utility-class for example looks like this:

```java
for (int lComponentNr=0; lComponentNr<pDimension; ++lComponentNr)
  lResultVector[lComponentNr]=pVector1.get(lComponentNr)+pVector2.get(lComponentNr);
```

If the vector used by the software-system is, for example, three dimensional, then the unrolled code would look like the next lines:

```java
lResultVector[0]=pVector1.get(0)+pVector2.get(0);
lResultVector[1]=pVector1.get(1)+pVector2.get(1);
lResultVector[2]=pVector1.get(2)+pVector2.get(2);
```

This piece of code becomes automatically generated by a frame processor and achieves a little higher performance, because there is no count variable to increase and to evaluate. In the all object-oriented solution, this could eventually be done automatically by a good compiler. It is, however, not standard and the developer has (mostly) no control about such optimizations. More complicated optimizations are not possible at all (for a detailed listing of generator optimizations see [CE00, p. 345+]).

Also not possible without the aid of a frame processor is the generation of specialized constructors in Java. If for example the three dimensional vector (4.3, 5.3, 6.21) has to be created, in the object-oriented solution just writing

```java
Vector3D aVector = new Vector3D(4.3, 5.3, 6.21);
```

, like in the frame technology implementation (see Listing 5.2 lines 17-22 for the implementation of the constructor) is not possible, because the vector classes do not know their dimensions at construction time. Instead the more inconvenient lines

```java
StandardVector aVector = new StandardVector(3); // 3 is the dimension
aVector.set(0, 4.3);
aVector.set(1, 5.3);
aVector.set(2, 6.21);
```

would do the same. Please notice that in C++ the convenient solution could be achieved by the use of the ellipsis “...” for passing an arbitrary number of arguments, but would lead to difficult readable code inside the vector library.

The only workaround to these disadvantages is to write a customized object-oriented vector library manually from scratch or adjust an existing one and include exactly the needed vector classes with their dimensions and supported functionality. Manually rewriting as well as manually adjusting, however, are long and time consuming processes. Luckily, this is exactly what a frame processor does automatically in almost no time.

Another idea could be to develop the vector library without using object-oriented programming and inclusion polymorphism. Instead frame technology and a procedural language will be used. A proper target language for this could be C. With C and frames, it could be done the following way: Frame hierarchies have to be developed that take the specification of the used vectors and their customized configuration as input for the generation of the specific variant of the vector library. The vectors have to been built as structures (C keyword: struct), holding the appropriate data (the
7 Frames in the Context of Object-Oriented Programming

dimension, the components according to the dimension, and so on) in the appropriate form (unit or standard vector). For each required operation, the frame processor must generate free functions with different prototypes and different prefixes. Each function bears the name of the operation and takes the specific types of structures, for which this function is specialized, as parameters. To make it possible that the appropriate algorithms are used, each operation has to be provided multiple times. For example, if the add-operation is specified for four-dimensional unit and standard vectors, there have to be an addFourDimensionalVectors-function, taking two four-dimensional standard vectors, another add-function with the same name, taking two four-dimensional unit vectors, and two functions, also with the same name, taking a four-dimensional unit and a four-dimensional standard vector in both different orders. All these functions return a four-dimensional standard vector. Because C has no support for overloading polymorphism [Eis99], a way is needed to distinguish these functions. This can be done with prefixes. The name of each function has to be extended with characters, specifying the input parameters of the function. This would allow to use the function later with different vector structures and get the desired result. The frame code would be clean and without any redundancies, but this method still bears many disadvantages.

If someone tries to understand the algorithms of a specific variant, it is generally easier to read the generated code than the frame source. The generated C source code is however procedural and therefore potentially more difficult to read than object-oriented Java (or C++) code.

Sometimes the desire might arise to change the generated code directly, instead of the frame source. This is not recommend, but it may be useful in some situations. Often object-oriented code is much easier to maintain than procedural code. In the example a change to the algorithm of the add-function would have to be applied on four different code parts (in each function) instead of just one (the utility-class).

A major disadvantage comes from the fact that the frame processor is not target language aware and can therefore be no replacement for inclusion polymorphism. For example there is no replacement for inheritance, and therefore no such thing like a common base class or interface for interoperability can be provided. The above solution just uses the standard vector as general container, because it can also hold a unit vector. This is however only a provisional solution.

If it is necessary to work with frame technology alone, without the aid of object-oriented programming techniques, because the target platform is a language lacking object-oriented elements, it might be better to use frame technology than to rely on procedural programming only. It is, however, not advisable to abandon available abilities of object-oriented programming.

Finally, it could be said that the best results can be achieved neither by the application of object-oriented programming alone, nor by the sole usage of frame technology (in conjunction with a non-object-oriented target language). The combination of both technologies (see the examples of chapter 5 “Frame Technology in the Context of Generative Programming“) can be easily arranged and provides the best of both worlds.

As Basset reports, frames are a good addition to object-oriented programming, because frames are a reuse technology, while objects are centered on use. Use means in this context that objects are excellent for modeling dynamic behaviors at runtime. Object-oriented programming provides also support for reuse in the form of class inheritance. Class inheritance has however some weaknesses, if applied for reuse purposes. An example for such a weakness is that classes inherit everything from its ancestor classes and it is not possible to remove inherited properties (“You get the whole gorilla even if you only need a banana” [Gut89, cited from Bas97]). Frames in contrast work during construction time and are meant for reuse purposes that could not be established by OO-techniques alone. Basset believes that frames will let the promises come true that OOP had made [Bas97]. For

---

23 It is faster, if the appropriate place in the frame source is difficult to identify. There might also be the need for small changes for a specific generated piece of code which would require a rather complicated checking in the frame code.
7 Frames in the Context of Object-Oriented Programming

a more detailed comparison between objects and frames it is recommend to read chapter 5 of [Bas97].
8 Techniques for Frame Processors

8.1 What are Techniques for Frame Processors?

Whenever someone starts to work with a technology that is new to him, he needs time to become familiar with it. This is mostly because there are many solutions which are not elementary parts of the technology but are usually known to the more experienced users of this technology. Because this techniques are not an essential part of the technology, they are not found in basic manuals. It would, however, be very useful (especially for a beginner) to benefit from the experience of advanced users of the technology that already made up solutions for common problems.

This is, of course, especially true for frame technology which allows many ways to solve a specific problem. Therefore, this document tries to counter the lack of written experiences by providing solutions to common problems in an easy and understandable way. These reusable solutions could be called patterns according to Gamma et al., because they state that patterns are reusable solutions in a specific context [GHJ+95]. By reading the term patterns, however, it is usually expected that the presented solutions have reached a high maturity level that was provided by a considerable time and amount of experience. Because frame technology is a rather unknown technology which not until recently became more interesting to a larger audience, there are not that many projects that are able to provide mature solutions. Another problem is that the solutions that have been found for frame technology problems appear on different abstraction levels. Some of these reusable solutions solve very general problems while others depend on the frame processor and work at code level. The latter ones could have been called idioms. There are solutions which depend more or less on the frame processors in use. They vary also in the provided abstraction level. Some solutions are quite obvious while others are not easy to understand at first. There are no hard borders which allow an easy categorization of the different types of solutions. Therefore, the decision was to use a general term and call the reusable solutions for frame technology problems simply what they are – techniques. A classification is, however, necessary for orientation. Therefore, several classifications are provided here concerning different aspects. The most important aspect is the placement of a technique in the generative domain model, so that working with the GP paradigm is supported.

8.2 Anatomy of a technique

The techniques of the catalog (see chapter 9) are described in a consistent form with fixed subsections. This is useful for quickly finding the necessary information. It also makes techniques easier to read and to understand. The names and organization of the subsections are loosely based on the template that was first used in [GHJ+95]. There are, however, some differences because frame techniques can not be exactly presented in the same way as design patterns. For example, the subsection Also Known As is not included for the frame techniques due to the lack of additional names besides the first one. On the other hand, there are new subsections that apply specifically for frame techniques (e.g. Placement in the GDM). All subsections are explained in the following.

Placement in the GDM

It is important to know to which part of the generative domain model the specific technique refers to, problem space, configuration knowledge or solution space. This helps developers who work with the generative programming paradigm to find the best suited techniques faster.

Type

Like Placement in the GDM, type is another categorization that assists in finding and selecting the best suited technique. The following types are currently present:
8.2 Anatomy of a technique

- Emulation Technique
  Emulation techniques are techniques that recreate the behavior of another technology (including another specific realization of frame technology) with a specific frame technology based solution.

- Optimization Technique
  Optimization techniques provide solutions to improve performance or memory behavior of the target language code due to the application of frame technology. Most optimization techniques rely on moving variability or functionality from runtime to generation-time. *Inlining* moves the function calls, *constant folding* the evaluation of expressions or functions and *loop unrolling* the loop-constructs.

- Other
  This type is stated in the case that no other type applies.

Requirements
The requirements describe which frame processors and programming languages are necessary to make use of the technique. Some techniques require frame processors with specific features. If this is the case, it is listed too.

Target Language
Some techniques are only usable for specific target languages. This subsection explains which languages are suited.

Examples from
Code examples are available for most techniques. In this subsection the projects are listed from which the code has been taken. In some cases, however, the example code has been specifically created. In such a case, it is also stated here.

Examples in
This subsection lists the frame processors and target languages in which the example code was written.

Intent
The *intent* is a short statement that outlines what the technique does and what problem it solves.

Description
The *description* is a more detailed explanation than the intent of what the technique does and what problem it solves. It provides an overview of the technique's possibilities and also places the technique in an understandable context.

Motivation
The motivation provides a scenario that presents a typical problem for which the current technique is an appropriate solution. This scenario also assists in understanding the following rather abstract realization of the technique [GHJ+95].

Applicability
The applicability provides a check list that describes the necessities a situation must have, so that the technique can be successfully applied. It also describes what specific properties of a situation prevent the technique from being useful in that situation.
8 Techniques for Frame Processors

Abstract Realization
The abstract realization explains how the technique can be realized without talking about the specifics of the various frame processors and target languages.

Realization in [specific Frame Processor]
This section provides a detailed explanation how the technique can be realized with a specific frame processor (the one that is named in the heading) and also provides appropriate sample code.

Advantages/Disadvantages
This sections lists the consequences a technique can have, divided in positive and negative ones.

Known Uses
Known uses names real projects that make use of the technique. Not in all cases such projects are known. Some techniques have been created from scratch and have not been used in real projects yet.

Related Techniques
In this subsection other techniques are mentioned which are related with the current one. They may be similar to or can be used in combination with the current technique.

8.3 Examined Projects
In order to find techniques and benefit from experienced frame developers, several projects have been examined for reusable solutions. These projects are listed in the following.
ABA

The **ANGIE-Based GUI-generAtor (ABA)** is an experimental prototype developed by Max Schlee. It is a generator that builds a graphical user interface (GUI) for an image viewing and manipulating application along with its functionality. It has been developed using the GP paradigm as method and **ANGIE** in combination with C++ as realization technology. The desired application is specified using a graphical frontend that produces a XML-file. Figure 8.1 provides an overview of its principle. For a detailed explanation on how it works see [Sle02].
GP-WEB

GP-WEB is a prototypical generator for e-learning systems that was developed by the GP-WEB team of the University of Applied Sciences Kaiserslautern, Zweibrücken, Germany. It allows to order a customized e-learning system through a web-based frontend (see Figure 8.2). After the ordering, the generated system is sent by email to the customer. It was developed using the GP paradigm as method. For its technical realization it uses ANGIE and web-services. It also features an XML-based language for specification of the learning contents. You can try it yourself at [BEH+02]. For a detailed explanation how it works see [WH02a] and [WH02b].

HUTN-Parser Project

Matthias Schönwald developed a parser-generator for HUTN-based configuration files on behalf of Delta Software Technology. HUTN (Human-Usable Textual Notation) is an upcoming standard of the OMG [OMG]. This notation allows to store data from a MOF-model (Meta Object Facility, see [OMG]) in a clearly arranged and easily readable way [Soe02]. The parser-generator reads a XML-description of a feature diagram and generates a parser that is specific for this description. The generated parser reads HUTN-based files and checks if these files are valid configurations of the feature diagram. The parser is ANGIE-based and can be used as part in another ANGIE-based generator. ANGIE is also used as the implementation technology for the parser-generator. Therefore, ANGIE generates ANGIE code.
HyperSenses

HyperSenses is a commercial product from Delta Software Technology [Sch03; DST]. It implements the principle of Intentional Programming (IP) (see [ISC; Sim97a]). Intentional Programming is a new paradigm that focuses on the intention of code instead of its actual presentation. The aim is to preserve the design information that is usually not identifiable in the code [ISC]. In order to reach this goal, HyperSenses stores the code not as ASCII-text. Instead XML- and MOF-based (see [OMG]) repositories are used to store the “intention” of the code. Various views can be rendered that show a specific representation of the code which can be also edited through these views. For example, there can be a representation showing C++-code while another representation shows design issues (including design patterns) and dependencies and a third representation renders the code in a domain specific language.

HyperSenses consists of three components: Active Intent, Pattern Composer (see Figure 8.3), and Meta Composer (see Figure 8.4). Active Intent is a programming editor that provides features like changing a representation or context sensitive checks and operations. The two other tools allow to extend the rendering patterns on different abstraction levels. HyperSenses uses ANGIE as an underlying technology for various generation tasks and for the manipulation of repository data [DST].

Figure 8.3 Pattern Composer
(Courtesy of Delta Software Technology)
Mini-ABA by Max Schlee is an experimental GUI-generator like ABA. However, unlike ABA it features the generation of calculator applications. It also integrates some additional technologies, like for example, the ability to work with resource-files. The realization technologies are XFramer and C++. For more information see [EEE+03b].

PBE

Pattern By Example (PBE) is a product from Delta Software Technology which can be obtained from [DST] free of charge. It allows to capture the rather abstract descriptions of design patterns (see [GHJ+95]) in a machine processable form, therefore allowing to store patterns as so-called
pattern definitions. These pattern definitions are independent of the used programming language. PBE is shipped with several patterns already included, but also enables the user to create his own patterns. A new pattern can be created from an existing implemented pattern in a specific context. This feature is the reason for its name – Pattern By Example. The Pattern Composer of HyperSenses is a successor of PBE. ANGIE is used by PBE as backend to render the various embodiments of a pattern. For more information see [DST].

![Diagram](image)

Figure 8.6 Principle of the SCORE Integration Suite
(Adapted from [DST])

SCORE Integration Suite

SCORE Integration Suite is a commercial product from Delta Software Technology. Its main purpose is to integrate legacy applications with modern technologies like Java, XML and modern web-technologies [DST]. Other technologies it can interact with include embedded systems, component technologies like CORBA (see [OMG]) or EJB (see [SUN]), and many more.

The method used by the SCORE Integration Suite is called Model Driven Legacy Integration. This method is based on the MDA approach (Model Driven Architecture) of the OMG (see [OMG]). SCORE Integration Suite provides a GUI as front-end that allows the user to map interface definitions to each other in graphical/semi-graphical ways. The actual integration is done by generating custom middleware and proxies which allow the different technologies to communicate with each other. The technology used for the generation processes is the ANGIE frame processor. In addition to the graphical specifications, custom frame code can be embedded in the system [DST]. Figure 8.6 shows the principle of the SCORE Integration Suite and its relation to common MDA terms.
Adapt Emulation

Placement in the GDM: Solution Space
Type: Emulation Technique
Requirements: abstraction concept-based frame processor (i.e. ANGIE or XFrmer)
Target language: any
Examples from: specifically created
Examples in: plain text with ANGIE and XVCL

Intent
Emulate the adapt concept with the abstraction concept.

Description
The abstraction concept can emulate the adapt concept's break- and insert-commands if the emulating abstraction concept-based frame processor supports some kind of multi-set variables (e.g. collections in ANGIE).

Motivation
There are many reasons why adapt concept-based frame technology can be valuable to a project. It is out of the scope of the description this technique to discuss all possible applications of adapt concept-based frame technology. Simply consider the case that for an actual project this technology should be applied. There may, however, be reasons why an adapt concept-based frame processor can not be used. One problem could be that an abstraction based frame processor is already in use and a new tool would cause an additional administration effort. Another reason could be that adapt as well as abstraction functionality is needed at the same time. This could lead to an often changing use of different frame processors which can be very inconvenient. Whatever the reasons are (technical or economical reasons), this technique allows to gain the advantages of the adapt concept-based frame technology without actually using such a frame processor.

Applicability
Use the adapt emulation technique if
☑ you need adapt functionality
but
☑ you have only an abstraction concept-based frame processor available.

Abstract Realization
A set of functions has to be implemented that enables the functionality of the adapt concept commands insert, insert-before and insert-after. Instead on frames of the adapt concept the new commands work on frame instances of the abstraction concept.
Realization with ANGIE (Generated Code: plain text)
The following example (see Listing 9.1 and Listing 9.2) is a simple application of the adapt-concept.

```xml
<x-frame name="EmulationFrame" language="java">
some fixed content
  <break name="TEST">
some variable content
  </break>
more fixed content
</x-frame>
```

Listing 9.1 A XVCL-frame generating some simple text (Emulation.XVCL)

```xml
<x-frame name="EmulationSPC" outfile="Emulation.txt">
<adapt x-frame="EmulationFrame.XVCL">
  <insert-after break="TEST">
some other variable content
  </insert-after>
</adapt>
</x-frame>
```

Listing 9.2 An adapting XVCL-frame generating some simple text (Emulation.S)

Without an insertion the generated output would look like in Listing 9.3.

```
1. some fixed content
2. some variable content
3. more fixed content
```

Listing 9.3 Textual output of the emulation example (Emulation.txt)

Line 2 some variable content is the one which is enclosed with breaks and therefore marked as a variation point. By applying the insert-after of Listing 9.2 in line 3, the generated text is extended with the line some other variable content (see Listing 9.4).

```
1. some fixed content
2. some other variable content
3. some variable content
4. more fixed content
```

Listing 9.4 Textual output of the emulation example using insert-after (Emulation.txt)

If instead of the insert-after command insert-before is used, the line some other variable content is respectively inserted before the line some variable content (see Listing 9.5):

```
1. some fixed content
2. some other variable content
3. some variable content
4. more fixed content
```

Listing 9.5 Textual output of the emulation example using insert-before (Emulation.txt)

Finally in the case of an insert command the new line replaces the old one (Listing 9.6)

```
1. some fixed content
2. some other variable content
3. more fixed content
```

Listing 9.6 Textual output of the emulation example using insert (Emulation.txt)

To emulate this behavior in ANGIE instead of a break (see Listing 9.1) an ANGIE-slot is used (see the slot TEST in the frame EmulationFrame of Listing 9.7). The slot is a variable of the type Collection and is initialized with the line some variable content. In order to do this, a language feature of ANGIE is used. The curly brackets ({} and {}) allow to initialize a collection with various elements separated by commas. In this case only one element is inserted - the frame instance that contains the default value some variable content. The creation process with the createFrame-function can be realized directly in the initialization.

The TEST slot equals the appropriate break. Changes can be applied with script-functions analogous to the different insert-commands. Each script-function changes the value of the collection

Page 104
9 Catalog of Techniques

type slot by assigning the old value and the new one in the appropriate order. The insert-function just replaces the old value with the new one. The used insert-function is finally called by the main function which is equivalent to the SPC in XVCL (Listing 9.2). Calling the frame's creation-script can be compared with the adapt-command of the XVCL-frame (see Listing 9.2). The generated output is identical (Listing 9.4, Listing 9.5, and Listing 9.6).

```
**NGEC
.Frame CodeLine(line)
  <!line!>
.End Frame

.Frame EmulationFrame
  .Dim TEST = {createFrame("CodeLine","some variable content")}
  some fixed content
  <!TEST!>
  more fixed content
.End Frame

.Function insertAfter(FRAME,VALUE)
  .FRAME.TEST = {FRAME.TEST, createFrame("CodeLine",VALUE)}
.End Function

.Function insert(FRAME,VALUE)
  .FRAME.TEST = VALUE
.End Function

.Function insertBefore(FRAME,VALUE)
  .FRAME.TEST = {createFrame("CodeLine",VALUE), FRAME.TEST}
.End Function

.Function main()
  .Dim anEmulationFrame=createFrame("EmulationFrame")
  .insertBefore(anEmulationFrame,"some more variable content")
  .Export(anEmulationFrame,"Emulation", "text")
.End Function
```

Listing 9.7 ANGIE-realization of the adapt emulation technique

Disadvantages

- Readability decreases

  The emulated frame code in the abstraction concept is, of course, harder to read than native commands of the adapt concept.

Known Uses

There are no known uses yet.

Related Techniques

none
Constant Folding

**Placement in the GDM:** Solution Space  
**Type:** Optimization Technique  
**Requirements:** frame processor with the ability to perform calculations (i.e. ANGIE, FPL, Netron Fusion, XFrmer or XVCL)  
**Target language:** any  
**Examples from:** specifically created  
**Examples in:** Java with XFrmer

**Intent**
Expressions with known operands are evaluated during the generation process to increase runtime performance [CE00, p. 345].

**Description**
This technique is the frame technology version of the `constant folding` generator optimization from [CE00, p. 345] which is a specific variant of `partial evaluation` (see [CE00; JGS93; Jon96]). It is a useful technique with possible benefits on both memory and performance improvements. To some extend, constant folding could be achieved with a preprocessor or macro language. With a frame processor, however, constant folding can be used even on complicated problems. It is furthermore useful on systems where no aids like macros or preprocessors are available.

**Motivation**
Many software-systems are fragmented with small (or sometimes even medium or large) sized helper functions which have parameters that are already known (or could be calculated) at compile-time. These function are often called throughout the system. Consider, for example, a function for computing the factorial (e.g. `double fac(double x)`). Is the factorial of 11 (with 11 being a at compile time known value) needed, the programmer would rather write `a = fac(11)` than using a calculator and filling in `a = 3.99168e+07`. The latter could however be a large improvement in performance, especially if the function becomes called often. In even more complicated cases, where a calculator isn’t sufficient to determine the return value of an expression, the programmer would more than ever use the runtime function (or write out the expression) than face the inconvenience of manual evaluation. This is not a bad decision at all. The manual evaluation of such expressions would waste important development time. Tracking and changing such known values in complicated algorithms of existing code wastes even more time. It is very difficult to maintain code with already evaluated expressions, if the intention can no longer be identified. It is even worse if a (at compile time) known value is used in a function and the return-value of that function is used by another function, and so on. Therefore a technique is needed which allows to transfer the evaluation of expressions with known operands from runtime to compile-time (or before), without the loss of development time and intention.

**Applicability**
Use the `constant folding` technique, when

- Expressions with known (or at compile-time evaluable) operands are used often or in time critical environment.
expressions with known (or at compile-time evaluable) operands are used which are in such a complex manner that a noticeable performance loss can be experienced.

**Abstract Realization**

The algorithm which evaluates the expression must be provided in frame code. In the target language code, the expressions with at compile time known operands can be replaced by a slot that calls the frame function. This has the effect that in the generated source code the expression is replaced by its return value.

To gain the ability to switch constant folding on and off, the functions have to be provided twice – in frame code and target language code. This has the disadvantage that changes to the algorithm must be made in two places, therefore decreasing maintainability. However, there are luckily solutions to this problem. In *XFramer*, the same language can be used for frame code and target language code. Thus the function has only to be implemented once in an external file or library. This file can then be included in both code types. In case of using a library, the function can be called from frame code as well as from target language code. Using another frame processor, there is also a way to increase the maintainability. The solution is to implement a generator-generator which generates the frame code as well as the target language code (see the generator-generator technique and the alternative rendering technique). If the additional effort is worth the maintainability advantage is, however, questionable. It depends on the complexity of the function.

**Realization Example in XFramer (Generated Code: Java, Frame Code: C++)**

The following example (see Listing 9.8) shows a Java program which solves a typical combinatorial problem. It calculates the numbers of possibilities for k-combinations of a n-set [CSN02].

```java
import java.util.*;

public class CFExample //Lowest Common Multiple {
    static long fac(int x) {
        long fac=1;
        for (int i=2;i<=x;++i) {
            fac*=i;
        }
        return fac;
    }

    static long nOverK(int n,int k) {
        return fac(n) / (fac(k)*fac(n-k));
    }

    public static void main(String[] args) {
        System.out.println(nOverK(12,3));
    }
}

Listing 9.8 Constant Folding Example – NoverK (CFExample.java)
```

In this example the operands are known. There is no need to do the calculation at runtime, decreasing performance. Using *XFramer*, the functions have to be implemented in the generator's code (see lines 6-17 of Listing 9.9), and additionally to the implementation in the target language code (see lines 27-39 of Listing 9.9). If both implementations are present, constant folding can be turned on and off on demand.

```c++
// Constant Folding with XFramer

frame (MathAid) {
    public:
```

Page 107
static long fac(int x)
{
    long fac=1;
    for (int i=2;i<=x;++i)
        fac*=i;
    return fac;
}
static int nOverK(int n, int k)
{
    return fac(n) / (fac(k)*fac(n-k));
}
}

frame (Main)
{
    public:
    Main()
    {
BEGIN_FRAME_TEXT()
    class CFExample //Lowest Common Multiple
    {
        static long fac(int x)
        {
            long fac=1;
            for (int i=2;i<=x;++i)
                fac*=i;
            return fac;
        }
        static long nOverK(int n,int k)
        {
            return fac(n) / (fac(k)*fac(n-k));
        }
        public static void main(String[] args)
        {
            System.out.println(<!MathAid::nOverK(12,3)!>);
        }
    }
END_FRAME_TEXT()
};

int main()
{
    Main x;
    x.exportToFile("CFExample.java");
}

Listing 9.9 Constant Folding Example – nOverK – Specification code in XFramer (CFExample.cpp)

The generated Java code is shown in the following listing (Listing 9.10). In line 18 of the generated code, the function call nOverK(12,3) has been replaced by its return value 220.

class CFExample //Lowest Common Multiple
{
    static long fac(int x)
    {
        long fac=1;
        for (int i=2;i<=x;++i)
            fac*=i;
        return fac;
    }
    static long nOverK(int n,int k)
    {
        return fac(n) / (fac(k)*fac(n-k));
    }
    public static void main(String[] args)
    {
        System.out.println(220);
    }

Listing 9.10 Constant Folding Example – nOverK – generated code (CFExample.java)
To switch back to runtime evaluation, the slot in line 42 of Listing 9.9 \( \text{MathAid::nOverK(12,3)!} \) just has to be replaced by the original expression \( \text{nOverK(12,3)} \).

**Advantages**

- **Increase in performance**
  
  The time the expressions needed to be evaluated is saved, because their return value is already embedded in the generated source code.

- **Less memory usage**
  
  Due to the fact that the expression is already calculated, all memory which has been originally be allocated by the original expression is saved. If an expression includes the use of a function or method and all occurrences of that function can be replaced by the frame processor, the function itself is also no longer needed. It can be left out, reducing the source code in size and therefore saving additional memory.

**Disadvantages**

There are no known disadvantages besides the usual cluttering of the specification code with frame commands.

**Known Uses**

There are no known uses yet.

**Related Techniques**

For expressions consisting of functions only (i.e. without operators in symbolic form) this technique could be combined with inlining for operands (arguments) not known at compile time.

The *generator-generator* technique and the *alternative rendering* technique might be useful, if the ability to switch constant folding on and off is needed.
**Generator-Generator (Meta-Specification)**

**Placement in the GDM:** Problem Space, Configuration Knowledge and Solution Space  
**Type:** other  
**Requirements:** any frame processor that is able to produce its own frame code (i.e. ANGIE, XFramer and XVCL)  
**Target language:** any frame code language  
**Examples from:** specifically created  
**Examples in:** ANGIE

**Intent**  
Use a high level specification for the frame processor that contains frames, slots and functions to generate another specification for the same frame processor which is in turn used to generate the target language code.

**Description**  
A generator-generator (also called meta-generator) is a generator (in terms of the abstraction concept) implemented in a meta-specification that is used to produce in turn a specification as its generated target language code (see Figure 9.1 and Figure 9.2). The realization of a generator-generator is, however, not limited to the abstraction concept. Any frame processor which is able to produce its own frame code as generated code is usable. In case of the adapt concept a generator-generator can be called meta-specification. It is a high level specification for the frame processor which is used to produce another specification which in turn is used to generate the target language source code. The specification of the generator-generator therefore contains three abstraction levels: The generator-generator frame code, the generator frame code and the actual target language code.

**Motivation**  
In some projects additional variability is needed at frame code level which can, however, not be provided by the frame code language. A simple example is the inlining technique where names of frames and frame variables contain redundant parts which can not be handled on the same abstraction level by the frame code language. A generator-generator provides this type of abstraction. For example, the frames calcVolumeCount, calcVolumeImpl, calcVolumeCall from the inline-example all include calcVolume which is the name of the function that becomes inlined. The suffixes Count, Impl and Call are the same for each function which will become inlined. Instead of doing the error-prone work of choosing this names along with other redundant code manually, a generator-generator could produce them automatically based on the function which will become inlined.
Another typical problem appears if the product of a company is a generator (in opposition to an application) for a specific task and there are various such generators – this means there is a software-system family of generators. A meta-generator can be used to produce a specific member of the generator-family. A similar case is that of a company which sells a very powerful and complicated generator as its product. There might be a customer who likes to buy a generator that has not the full functionality in order to allow an easier handling. A meta-generator can be used in this case to produce a specialized generator that has exactly the functionality desired by the customer.

Generator-generators can also be used to build a generator for a specific problem, which can be used as a component for other generators. One such case is the understanding of a syntax description. In this case the generator-generator is a parser-generator. A specification of a language syntax is used to parametrize the meta-generator in order to produce a specific generator that can parse the language which syntax is described in the specification. The generated specific generator is a parser for the given language and can be used as a part of larger generator which needs to parse documents of this language, because it uses the language as its DSL. Parser-generators already have a long tradition independent of frame technology. For an example see Yacc [Joh79]. An overview of parser-generators is provided by [GF03]. For a frame-technology version of a parser-generator and a similar application see known uses below.

**Applicability**

Write a generator-generator when

![Figure 9.1 Principle of a Generator-Generator](image1)

![Figure 9.2 Principle of a Generator-Generator within the Abstraction concept](image2)
an additional abstraction level is needed.

or

☑ a variability issue is too complicated to be expressed on the same (meta)-level of frame code.

or

☑ your product is a software-system family of generators.

Abstract Realization

The generator-generator specifies the variability of a software-system family of generators. Each generated generator contains this variability in static form while it holds its own variability for the target language code. This means that in the generator-generator two levels of variability must be distinguished. This is done with different implementation techniques depending on the frame processor.

Another problem that must be solved is that the generator-generator code contains frame code that is not intended to be interpreted by the frame processor because it is the target language code on the current level. The frame code of the target generator (the frame code for the target specification in the terms of the adapt concept) must be masked in a way that it can not be detected by the frame processor. How this is done also depends on the frame processor in use.

Realization in ANGIE (Generated Code: C++)

Consider the following example. A generator-generator should produce generators which produce C++ header or implementation files for a specific project. Therefore, a generated generator could be either a header or an implementation generator. It further generates files for a specific project. The project name is hard-coded into the generator. It is, however, a variability of the meta-generator. Furthermore, associated with each project it is possible to include a function which should appear in each class and is therefore also hard-coded in the generator while it is a variability of the meta-generator. The variability of each generated generator includes the name of the class for which an implementation or header should be produced as well as an additional function which may vary for each class. Figure 9.3 shows a feature diagram for the meta-generator while Figure 9.4 shows a feature diagram for a generated generator which has been parametrized to produce C++-headers.

Figure 9.3 Feature Diagram of a Meta-Generator

* This feature is optional, because a general function which will be included in every class of the project is possible, but seldom needed.

** The variable name of the project is used for the inclusion of a specific header file.
9 Catalog of Techniques

Please notice that the example is not a really useful application of a generator-generator. The same result can be easily achieved with a standard generator. The example, however, demonstrates the necessary programming techniques in a simple and understandable manner.

Listing 9.11 shows the **ANGIE** code module of the generator-generator specification. It contains two frames. One frame consists of the generic description of the header generator while the other contains the generic description of the implementation (cpp) generator. These two frames can be parametrized with the name of the project and a function. The project name is used for an include file while the function is an example for a function which should be inserted in each class later. This parameters are variabilities of the meta-generator only. Once a generator is generated, it is no longer a variable but a static part. For example, if the project is called *MyProject* all generated generators always build class files (header or implementation) that include the *MyProject.h*-file. This is no longer a variability of the generated generator. Please notice that in a real application, instead of the *MyProject.h*-file a set of header-files specific to the project would have been included.

![Feature Diagram of a Generated C++ Header Generator](image)

In order to parameterize the meta-generator and thereby select a specific generator to be generated, the arguments are passed through the command line (see the parameters of the main function of Listing 9.11 and the second call in Listing 9.13). This kind of parametrisation which results in specific generated generators is also called *partial specialization of generators*. In this example (Listing 9.13) a header generator is built for the project *MyProject* with the function *projectSpecificFct*. The result can be seen in Listing 9.15. For an example of an implementation generator see Listing 9.14. Both types of generated generators take a class name and function name as their input parameters in order to build the header or implementation skeleton which also includes the skeleton for the function. The other function specified by the meta-generator is also included but can not be changed because it is a static/immutable part of the generated generator. Exemplary command line calls from a batch file can be found in Listing 9.16 and Listing 9.18. The proper generated source code in C++ is shown in the Listings 9.17 and 9.19.

The frame code of the target generator (the frame code for the target specification in the terms of the adapt concept) must be masked in a way that it can not be detected by the frame processor. This also depends on the frame processor in use.

In order to mask the frame code of the generator-generator in a way that it can not be detected by the frame processor the **ANGIE** syntax for the frame code must be examined in detail. One obvious feature of an **ANGIE** frame command is that it starts with a dot (except for flags). Therefore one way to mask a frame command is to replace the dot with a slot containing the dot (see {@dot@} in Listing 9.11). For the **ANGIE** compiler the masked frame commands seem to be target language code because of the missing dot. When the slot thereafter is filled with a dot by the **ANGIE** interpreter, the result is regular **ANGIE** code.
To solve the problem of distinguishing between two levels of slots, a special directive for the ANGIE compiler can be used. ANGIE allows not only to redefine the flags but it also supports to have several flag definitions in the same configuration file at once. In the configuration file the alternative definitions can be named by having a section with the custom name in square brackets. The name Flags serves as default name (see Listing 9.12). For the example the name meta was chosen for the flags of the meta-slots (the slots of the generator-generator) (see also Listing 9.12). Within the code module, the directive \*.NGEFLAGS followed by the chosen name (in this case meta) specifies which flag-definition is used (see the beginning of Listing 9.11). The generated code does not contain such a directive which means that the definition of the default section named Flags will be applied for the generated ANGIE code. The meta-flags are defined as {@ and @} while the flags in the generated ANGIE are <! and !>. This way, both levels of flags can be easily distinguished.

```angie
/**NGEC
 .*NGEFLAGS meta
 .Frame HeaderGen(ProjectName, FunctionName)
  .Dim dot="."
 **NGEC

{@dot@}Frame Header(ClassName, FunctionName)
#include "{@ProjectName@}.h"

class <!ClassName!>
{
  void <!FunctionName!>();
  <void {@FunctionName@}()>;
}
{@dot@}End Frame

{@dot@}Function main(ClassName, FunctionName)
  {@dot@}Dim aHeader=createFrom("Header",ClassName,FunctionName)
  {@dot@}export(aHeader, ClassName,"Header")
{@dot@}End Function

.Frame CppGen(ProjectName, FunctionName)
  .Dim dot="."
/**NGEC

{@dot@}Frame CppFile(ClassName, FunctionName)
#include "<!ClassName!>.h"
#include "{@ProjectName@}.h"

void <!ClassName!>::<!FunctionName!>()
{}

<?void <!ClassName!>::{@FunctionName@}()
{ //some code
}
<?
{@dot@}End Frame

{@dot@}Function main(ClassName, FunctionName)
  {@dot@}Dim aCppFile=createFrom("CppFile",ClassName,FunctionName)
  {@dot@}export(aCppFile, ClassName,"C++")
{@dot@}End Function

.End Frame

.Function main(GeneratorType, ProjectName, FunctionName)
  .Dim aGenerator
  .If GeneratorType="header" then
    .aGenerator=createFrom("HeaderGen", ProjectName, FunctionName)
  .End If
```

Page 114
9 Catalog of Techniques

```plaintext
.. If GeneratorType="cpp" then
  .aGenerator=createFrame("CppGen", ProjectName, FunctionName)
. End If
  .export(aGenerator, ProjectName & "." & GeneratorType & ","Generator"="ANGIE")
.. End Function
Listing 9.11 Generator-generator in ANGIE
```

```plaintext
[Common]
TypeFile=.	ypes.xml

[Compiler]
MessageFile=c:\programme\delta\angie\sys\dltnngcsg.msg
DeclListPath=..\cmp\DeclListFileExt=.DLST
DeclOutputPath=..\cmp\ListPath=..
ListFileExt=.CLST
OutputPath=..

[Interpreter]
ListFile=..\cmp\list.ilst
ModuleIndexFile=..\cmp\moduleIndex.xml
MessageFile=c:\programme\delta\angie\sys\dltngisc.msg

[TestDriver]
ListFile=..\cmp\driver.lst

[Container]
System=c:\programme\delta\angie\sys\dltngsyct

[meta]
OpenEmbeddedCode ="{"
CloseEmbeddedCode ="@}"

[Flags]
OpenEmbeddedCode ="<!"
CloseEmbeddedCode ="!>"

[Export]
Column=Off
Line=Off
BackReference=Off
Listing 9.12 ANGIE configuration file for the generator generator
```

call c:\programme\delta\angie\bin\dltngcc.exe .\GeneratorGenerator.cdm .\angie.ini
call c:\programme\delta\angie\bin\dltnge.exe GeneratorGenerator main .\angie.ini header
MyProject projectSpecificFct

Listing 9.13 Batch file calling ANGIE with the generator-generator code module

```plaintext
**NGEC
..Frame CppFile(ClassName, FunctionName)
#include "<!ClassName!>.h"
#include "MyProject.h"

void <!ClassName!>::<!FunctionName!>()
{
}

void <!ClassName!>::projectSpecificFct()
{
    //some code
}
..End Frame

..Function main(ClassName, FunctionName)
  .Dim aCppFile=createFrame("CppFile",ClassName,FunctionName)
  .export(aCppFile, ClassName,"C++")
..End Function
Listing 9.14 A generated implementation generator
```

**NGEC

Page 115
Advantages

- High Level of Abstraction

  The high abstraction level this technique provides aims for an also high level of automation. Complex tasks can be realized with a (in relation to usual solutions) small amount of code. Maintainability is also considerably increased, because changes which apply to many places in the result can be made by changing only a small amount of the meta-generator's code. It is the same advantage that applies to any generator because of its higher abstraction level. For meta-generators, however, this advantage is even increased.

Disadvantages

- High Complexity
9 Catalog of Techniques

The high level of abstraction comes at the cost of an also high complexity. Meta-generators are not easy to handle and demand careful planning from the developer. The maintainability advantage due to the high abstraction level is also decreased on the other hand, because of the difficulty to track and understand the impact of changes in the code of a meta-generator.

Known Uses

- **HyperSenses**

  *HyperSenses* uses a meta-generator to build generators which act as parsers and are also responsible for specific renderings (see *HyperSenses* in chapter 8.3 Examined Projects). The meta-generator as well as the generated generators are implemented with *ANGIE*. The meta-generator reads a XML-file (from pattern repository) that describes a specific rendering pattern and produces a generator that can parse input (from model repository) for this pattern and renders it in a specific form. This form can also include different programming languages. The rendering is the output of the generator. Therefore, the target language code makes up the rendering. There are several patterns described in the pattern repository which use the same input (from model repository) but allow different renderings. Therefore, *HyperSenses* uses a meta-generator that can be considered a parser-generator as well as a generator for alternative renderings.

- **HUTN-Parser Project**

  The parser-generator is implemented in *ANGIE* and generates *ANGIE*-based parsers that read HUTN files (see the description of the HUTN-Parser Project above).

Related Techniques

A generator-generator is useful to overcome disadvantages of the inlining and constant folding technique (see there). It can also be used in conjunction with the alternative rendering technique. If one alternative uses frame code as the target rendering language, a generator-generator is needed.
Inlining

<table>
<thead>
<tr>
<th>Placement in the GDM: Solution Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: Optimization Technique</td>
</tr>
<tr>
<td>Requirements: any frame processor</td>
</tr>
<tr>
<td>Target language: any language supporting some kind of procedures or functions</td>
</tr>
<tr>
<td>Examples from: specifically created</td>
</tr>
<tr>
<td>Examples in: C with ANGIE Generation Now! and XVCL</td>
</tr>
</tbody>
</table>

**Intent**
Replace functions, procedures or methods with their implementation in order to avoid the calling overhead [CE00, p. 345].

**Description**
This technique is the frame technology version of the inlining generator optimization from [CE00, p. 345]. It is a rather obvious technique which does not make use of advanced frame processor concepts. Thus, it can also be implemented with other tools supporting plain text replacement. Some compilers even interpret an inline-direction. This technique can, however, become very useful in combination with other techniques. The use of a frame processor additionally provides the ability to trigger the inlining of specific functions depending on other frame variables, thus giving the control completely to the developer. This distinguishes the frame processor from less powerful tools.

**Motivation**
Invoking methods, procedures and functions incurs some overhead with only a small impact in most implementations. In large projects with a deep calling hierarchy, however, the performance loss due to calling overhead may be noticeable. This is especially the case if many small functions are called regularly. An example of this is the consequential application of the Law of Demeter [HLR88] that leads to many small getter and setter methods. This overhead can be avoided by inlining function implementations. It is, however, also desirable that the inlining can be turned on and off easily or can be controlled on a per call or per function basis. Turning the inlining off is sometimes necessary for checking, reviewing and debugging the generated code or for generation of a version, with the requirement to keep the generated source code (and as a consequence, the compiled binary file) as small as possible.

**Applicability**
The application of the inlining technique will obfuscate the target language code (not the generated code!) to some extent. Therefore, it is advisable to use inlining only, if one of the following conditions apply:

- The inlined function is called in a time critical context.
- The software-system suffers from noticeable performance loss because of many calls of small functions.
- The software-system has a deep calling hierarchy.

24 C++ compilers, for example, interpret the keyword `inline`. They can, however, not guarantee that the function will definitely become inlined [Str97].
and one of the following conditions is also true:

- The target language/compiler provides no means for inlining.
- An efficient and maintainable switching between inline code and calling code needs to be available.

Abstract Realization

In order to make the explanation easier, only the term function will be used, but the statements also apply to procedures, methods and other equivalent concepts as well. Three frames are needed. One frame must keep the implementation of the function, one the header of the function and one the calling command. The implementation frame is adapted/aggregated by the header frame. The client frames which use the function can either adapt/aggregate the implementation frame or the command frame (see Figure 9.5). If the implementation frame is used, no calling overhead is needed. If the command frame is used, the call will be made at runtime (see Listing 9.22).

Local Variables

One problem associated with inlining is that local variables (as well as parameters) of the inlined function or method can come into conflict with the surrounding variables of the calling context. A possible solution is to use a convention that demands to attach a prefix, containing the name of the function to each local variable of the function. If, for example, the function calcVolume has the local variable sideLength, the variable could be named _calcVolume_sideLength. (see Listing 9.23, lines 9-12 for an example in ANGIE and Listing 9.25, lines 6-9 for an example in XVCL).

![Diagram of frame hierarchy of inlining](image)

If the same function is inlined more than once in the same context (in most languages called a block), this solution is insufficient because the initialization of a variable appears more than once. In most programming languages this is treated as an error. In this case, it is necessary to attach a frame variable to the name of the local variable which counts each use in order to generate names like...
Parametrization

If the inlined function uses parameters, the automatic copy mechanism (call by value) of the target language is bypassed. In case such a copy mechanism is needed, the function has to copy the specific parameters to local variables itself (see Listing 9.23, line 9). Parameters that are not copied act as in/out-arguments.

Return value

There is no easy solution for the inlining of a function with a return value. If the return value and the implementation of the function can be executed during the generation process, it is advisable to use the constant folding technique instead.

Variations

Depending on the implementation concept, it can be varied whether or not to use inlining selectively for each call, for context dependent groups of calls or for all calls of a specific function.

Realization Example in ANGIE (Generated Code: C)

The following example (see Listing 9.22) shows the inlining of an exemplary C function without parameters, using the ANGIE frame processor.

```plaintext
1. **NGEC Inline Example
2. .Frame myFunctionImpl
3.  
4.  //The Implementation Code of myFunction
5.  //Do something
6. 
7. .End Frame
8. .Frame myFunctionWithHeader
9.  
10. Dim lImpl=createFrame("myFunctionImpl")
11.  
12. void myFunction()
13. {
14.   <!lImpl!>
15. }
16. .End Frame
17. .Frame myFunctionCall
18.  
19. myFunction();
20. .End Frame
21. .Frame mainProgram
22.  
23. int main()
24. {
25.   //some code context
26.   <!myFunction!>
27.   //some code context
28. }
29. .End Frame
```
9 Catalog of Techniques

Listing 9.20 Inline Example 1 (Inline Example.cdm)

The generated output uses the function in a normal way (see Listing 9.21).

Listing 9.21 Inline Example 1 – calcVolume – generated with inlining off (INLINE.C)

By changing line 30 of Listing 9.22 to

 void myFunction()
 {
   //The Implementation Code of myFunction
   //Do something
 }

 int main()
 {
   //some code context
   myFunction();
   //some code context
 }

Listing 9.22 Inline Example 1 – calcVolume – generated with inlining on (INLINE.C)

Variations

In the last example the change of the frame variable myFunction switched all uses of the target function at the same time. This is the most desired application of inlining because it allows to change large portions of the target code depending on a global optimization flag which is typically also checked for other kinds of optimizations. In some cases however, other realizations of inlining might be required. By providing two frame variables, the application of inlining can be determined per use. The following changes to Listing 9.22 starting with line 30 illustrate this application:
The first two lines should not be changed anymore. With the inclusion of `<!myFunctionC!>` and `<!myFunctionI!>` in the target code, inlining is turned on and off. This variation can also be extended to a per group decision, as the next example illustrates:

In the last variation, the function is subdivided in three logical groups (the characteristics of each group have to be determined first). Changing line one from call to implementation (inlining), for example, results in changing the whole group (myFunction1).

**Realization Example with Parameters in ANGIE (Generated Code: C)**

The following example demonstrates inlining with a function which calculates the volume of a cube based on the length of one side. In this example a parameter is used. The individualization of the local variables is realized with a static variable (a so-called module variable\(^{25}\) in *ANGIE* terms), while the counting is done using a script function (see Listing 9.23, lines 72-77).

\(^{25}\) Module variables can be accessed throughout the whole module.
1. **NGEC Inline Example 2 - VolumeCalc**
2. 
3. .Static CalcVolumeCounter=0
4. 
5. .Frame calcVolumeImpl(p1)
6. 
7. .dim count=calcVolumeCount()
8. 
9. float _calcVolume<count>_sideLength=<!p1!>;
10. _calcVolume<count>_result=_calcVolume<count>_sideLength^3;
11. printf("%s", _calcVolume<count>_result);
12. 
13. .End Frame
14. 
15. 
16. .Frame calcVolumeWithHeader
17. 
18. .Dim lImpl=createFrame("calcVolumeImpl","pSideLength")
19. 
20. void calcVolume(float pSideLength)
21. {
22.   <!lImpl!>
23. }
24. 
25. .End Frame
26. 
27. .Frame calcVolumeCall(p1)
28. 
29. calcVolume(<!p1!>);
30. 
31. .End Frame
32. 
33. .Frame mainProgram
34. 
35. .Dim calcVolumeC=CreateFrame("calcVolumeCall","5")
36. .Dim calcVolumeI=CreateFrame("calcVolumeImpl","8")
37. .Dim calcVolumeI2=CreateFrame("calcVolumeImpl","3")
38. 
39. int main()
40. {
41.   <!calcVolumeC!>
42.   <!calcVolumeI!>
43.   <!calcVolumeI2!>
44. }
45. 
46. .End Frame
47. 
48. .Frame programFile
49. 
50. .Dim fvComment As Variant
51. 
52. #include <stdio.h>
53. 
54. .Dim lFunctions=createFrame("calcVolumeWithHeader")
55. .Dim lMain=createFrame("mainProgram")
56. 
57. .End Frame
58. 
59. Function calcVolumeCount()
60. 
61. .calcVolumeCounter=CalcVolumeCounter+1
62. .calcVolumeCount=CalcVolumeCounter
63. 
64. .End Function
Listing 9.23 Inline Example 2 – calcVolume (Inline Example 2 - VolumeCalc.cdm)

Line 47 to 49 shows the code that makes use of the function. It is called once in a normal way and
inlined twice. The generated output shows how the parametrization and the generated names of the
local variables work (see Listing 9.24).

Listing 9.24 Inline Example 2 – calcVolume – generated code (INLINE.C)

Realization Example with Parameters in XVCL (Generated Code: C)

This is an example of the realization of the calcVolume Function with XVCL. The XVCL realization
is very similar to the ANGIE realization. Each frame has its own file, but the frame hierarchy is the
same. There is a small difference in the individualization of the local variables with the count frame
variable. There are no functions in XVCL. The increase of the count variable is instead included in
the implementation frame (see Listing 9.25, line 10).

Listing 9.25 Inline Example 2 – calcVolume – Implementation Frame (calcVolumeImpl.xvcl)
Advantages

- Performance increase

  Depending on the number of uses and the complexity of the function, a noticeable increase in performance can be achieved due to the reduction of the calling overhead.

Disadvantages

- Increased complexity

  The application of this technique results in an increased complexity of target code and frame code which can also reduce the maintainability of the software-system and especially of the specification. One reason is the redundant use of names (e.g. calcVolumeCount, calcVolumeImpl, calcVolumeCall, and so on). This redundancy could be further reduced by the application of a generator-generator which produces the redundant frame code. This however, would make the system even harder to handle.

- Production time increase

  Additional effort is required to make the functions inlineable. The question is whether this effort is worth the increased performance of the software-system.

- No return value

  This technique works only for functions without return value.
Known Uses
There are no known uses yet.

Related Techniques
If the data with which a function operates is already known at compile-time, it is advisable to use the constant folding technique. It eliminates not only the calling time overhead but also the complete implementation runtime.
Intent
Loops with a (at compile-time) known number of iterations can be replaced by $n$ copies of its body [CE00, p. 346].

Description
This technique is the frame technology version of the loop unrolling generator optimization from [CE00, p. 345]. Loop unrolling usually increases performance at the cost of memory. With a frame processor, however, this optimization can be switched on and off depending on the actual demand.

Motivation
Program code often contains loops which have been designed for flexibility. This is particularly true for libraries. The application of such a piece of code (e.g. the use of a library function), however, does not necessarily need this flexibility at runtime. Therefore, the number of iterations can be determined before runtime which allows a static implementation. An example for this is a vector library which has been designed for vectors of $n$ dimensions. A software-system which uses the library may need only a constant number of dimension (two or three is usual). In this case, loops depending on the vector size are no longer needed and could be replaced by $n$ copies of the loop body, with $n$ being the number of dimensions of the vector. This would apply to all functions with element access (i.e. add, sub, initialization, and so on). But instead of writing specialized code manually, a frame processor can be used to generate the required versions. This is also true for versions which keep the loops to reduce the code size. This may be needed on specific hardware (i.e. embedded or legacy systems).

By and large, it can be said that processor architectures with a large cache tend to execute standard loops faster while unrolled loops are preferable in systems with a small cache or in parallel execution environments.

Besides this, there are loops with an already static number of iterations in the original code. They have not been replaced, however, to keep the source small and maintainable. If the loop had been replaced manually, changes to the original body of the loop would mean to change all copies of the body. For a high number of iterations, this could be a massive decrease in productivity and maintainability. On the other hand, keeping the loop can be a noticeable decrease in performance, especially if this code runs in a time-critical part of the software-system. Luckily, with the application of a frame processor it is no longer necessary to decide between productivity/maintainability and performance.

Applicability
Use the loop unrolling technique, when the number of iterations is known at compile-time and
the specification code must keep the loops, because of different variations of the code (e.g. members of a software-system family) or for maintainability reasons.

switching between unrolled and normal loops is required for different versions of the code (e.g. use on different platforms or architectures).

Abstract Realization

The loop which should be unrolled has to be provided in frame code. A loop typically consists of head and body. The head contains different elements, depending on loop type and programming language. The head of a do- or while-loop usually contains an initialization and a condition to terminate the loop. The head of a for-loop additionally provides an update command. The body consists of the code which will be iterated at runtime. To make the loop-implementation accessible at generation time (i.e. in the frame code) the body must be inserted in a frame while the head contents become part of the head of a loop at frame level. A slot is inserted at the place inside the code where the loop is needed. A function of the frame code contains the new loop (parametrized with the original values) that fills the slot with n copies of the original loop body (now contained in another frame) at generation time. Thereby, a frame variable is used as loop parameter.

The option to switch between the original and the unrolled loop is useful to customize code to the requirements of the platform (memory or performance). There are other reasons as well (see known uses below). To implement this option, the original code must also be provided and a parameter of the configuration (in the DSL – problem space) has to be mapped to allow the switching.

In order to increase maintainability, it should be avoided to do code changes to both, the original loop and the unrolled loop. To achieve this, the original loop must also be generated. The loop body can be easily reused while for the head the parametrization must be separated.

Realization in ANGIE (Generated Code: C++)

Listing 9.26 shows a simple C++-program (could also be just C) that initializes an array of integers with the value zero. In order to unroll the for-loop with ANGIE, the loop body is captured in the frame LoopBody while the frame MainProgram holds the surrounding code of the example (see Listing 9.27). The slot loop (from the type collection) is the insertion point where n copies of the loop body will be inserted in the generated code. It has to be of the type collection, so that instances of the frame that contain the loop body (LoopBody) can be inserted multiple times. The script-function insertUnrolledLoop contains the loop at generation time that does the unrolling by adding instances of the loop body to the insertion point. The insertion point is specified by the argument target. In a more general version of the function the instantiated and inserted loop body frame would also be passed as an argument. A version with variable parameters for the meta loop is shown later. The main function finally instantiates the Frame MainProgram and exports it after invoking the loop unrolling. Listing 9.28 shows the generated program that contains the unrolled loop.

```c
void main()
{
    int value[10];
    for (int i=0;i<=9;++i)
        value[i] = 0;
}
```

Listing 9.29 Array initialization using a loop

**NGEC**

.Frame LoopBody(fieldNo)
    value[<!fieldNo!>] = 0;
. End Frame
9 Catalog of Techniques

```plaintext
void main()
{
  int value[10];
  <!loop!>
}

Listing 9.30 A generator featuring the Loop Unrolling Technique

```
In order to reduce redundancy for maintenance reasons, the loop body and the parameters of the
loop head should be only provided once. This has been done in Listing 9.30. The original loop
reuses the loop body while the parameters of the loop head have become function arguments of
insertOriginalLoop and insertUnrolledLoop. To gain more generality, the loop body
has also be made variable. The Listings 9.31 and 9.32 shows the generated code.
### Catalog of Techniques

```cpp
.End Function
.Function main()
    .createCPPExample("loopExample – original", "no", 10, 50, 2)
    .createCPPExample("loopExample – unrolled","yes", 10, 50, 2)
.End Function
```

**Listing 9.33** The generator from Listing 9.29 with a better maintainability

```cpp
void main()
{
    int value[100];
    for (int i=10; i<=50; i+=2)
        value[i] = 0;
}
```

**Listing 9.34** Generated code of the Loop Unrolling technique – not unrolled

```cpp
void main()
{
    int value[100];
    value[10] = 0;
    value[12] = 0;
    value[14] = 0;
    value[16] = 0;
    value[18] = 0;
    value[20] = 0;
    value[22] = 0;
    value[24] = 0;
    value[26] = 0;
    value[28] = 0;
    value[30] = 0;
    value[32] = 0;
    value[34] = 0;
    value[36] = 0;
    value[38] = 0;
    value[40] = 0;
    value[42] = 0;
    value[44] = 0;
    value[46] = 0;
    value[48] = 0;
    value[50] = 0;
}
```

**Listing 9.35** Generated code of the Loop Unrolling technique – unrolled

### Realization in XFramer (Generated Code: C++, Frame Code: C++)

There are several possibilities to implement loop unrolling with *XFramer*. One option is to use slots being an instantiation of the template `vector` (from the Standard Template Library [Str97]) and fill them line by line with copies of the loop body. This solution is very similar to the *ANGIE* realization in Listing 9.27, due to the substitution of the `collection` type slots for `vector` type slots. Another solution is presented here, in Listing 9.33.

```cpp
#include <iostream.h>
frame FrLoopBody {
    public:
        FrLoopBody(int fieldNo)
        {
            BEGIN_FRAME_TEXT()
            nValue[<!fieldNo!] = 0;
            END_FRAME_TEXT()
        }
};
frame FrMainProgram {
    public:
        FrMainProgram()
        {
            MString loop;
            int i;
            for (i = 0; i < 10; ++i) {
```
In this solution no multi-set or container variable (like a collection or a vector) is used. The individual lines of the loop become instead connected as plain strings. At the end of the target language block of frLoopBody, there is a newline character to ensure that each initialization starts in a new line. This solution is very elegant but carries a maintenance problem. A programmer who reads this code sees the loop slot in the target language block of FrMainProgram. The type of the loop slot is MString. Therefore, the programmer does not know if the slot will be used to insert one frame or multiple frames. Using a vector of MStrings instead solves this problem.

The function which does the unrolling is the constructor of FrMainProgram. This is a good solution because it can be easily recognized that the function is used on the appropriate frame. It also saves an additional function call. The constructor is executed when the frame frMain is instantiated. The generated code is equal to the code of Listing 9.27.

**Advantages**
- Option to switch between increase in performance or less memory usage

Depending on the requirements this technique provides either an increase in performance (unrolling) or less memory usage. The latter advantage is in fact the default which had been also available without the application of this technique. The actual advantage is, however, that the run-time behavior can be easily switched between increase in performance or less memory usage in order to fit the actual architecture.

**Disadvantages**
There are no known disadvantages besides the usual cluttering of the specification code with frame commands.

**Known Uses**
This technique is used in the vector generator from [EEE+03a].

**Related Techniques**
one
### Template Emulation

<table>
<thead>
<tr>
<th>Placement in the GDM:</th>
<th>Solution Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Emulation Technique</td>
</tr>
<tr>
<td>Requirements:</td>
<td>any frame processor</td>
</tr>
<tr>
<td>Target language:</td>
<td>any</td>
</tr>
<tr>
<td>Examples from:</td>
<td>specifically created</td>
</tr>
<tr>
<td>Examples in:</td>
<td>C++ with XFramer</td>
</tr>
</tbody>
</table>

#### Intent
Emulate parametric polymorphism using frame technology.

#### Description
This simple technique allows to instantiate type independent templates with specific types. It is an emulation of a template mechanism, like that of the C++-compiler and can be used with languages which do not support parametric polymorphism.

#### Motivation
Templates allow to represent concepts independent of data types (see [Str97]). This is useful for different fragments of code. A concept can be expressed as class, function, method, and so on. If the same concept applies to a different type, it is not necessary to rewrite the whole code. With templates, an implementation can be made independent of the types it operates on. Consider, for example, a sort-function which was originally written to work on a numeric type. Without parametric polymorphism it has to be rewritten from scratch if it is also needed for strings. By using templates, the function is written only once and becomes instantiated by parameterizing it with the appropriate data type. This works, of course, on any data type. There are only a few languages that have a mechanism for parametric polymorphism like C++. For Java, it is planned (see Java generics [BCK+01]). With frame technology it is, however, possible to provide such a mechanism for all languages lacking it.

#### Applicability
Use template emulation if
- parametric polymorphism is needed in your project
and
- the used language does not provide parametric polymorphism.

#### Abstract Realization
The code which should be reused has to be encapsulated in frames (this is called “the code is framed”). The types of which the code shall become independent of are the variation points of the frame and must be exchanged by slots. If a type-specialized version of the frame is needed, the frame has to be either adapted (adapt concept) or instantiated (abstraction concept) while the slots must be filled with the appropriate data type.
Catalog of Techniques

Realization in XFramer (Generated Code: C++, Frame Code: C++)

Listing 9.34 shows a C++-template that realizes a function which swaps the values of two variables (arguments of the function `swap`). The template parameter `T` is used as data type for the variables, thus making them type independent.

```c++
template<class T>
void swap(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
}
```

Listing 9.37 C++ Function Template Swap

The frame implementation in XFramer is very similar. The content of the template becomes the target language block of a frame (it is framed). The types of which the code shall become independent are the types of the arguments of the function `swap`. Therefore, they have to be replaced by the slot `<!T!>`. In order to instantiate the frame, a constructor is provided that takes the template parameter as argument.

```c
frame FrSwap {
    FrSwap(char* T)
    {
        BEGIN_FRAME_TEXT()
        void swap(<!T!>& a, <!T!>& b)
        {
            <!T!> c = a;
            a = b;
            b = c;
        }
        END_FRAME_TEXT()
    }
}
```

Listing 9.38 XFramer Swap

Advantages

- Debugging of the code generation process is possible
  
  It is not possible to debug the instantiation process of C++-templates which happens during the compilation process. The resulting generated code can be debugged, but not the instantiation process itself. By using frame technology based templates, however, it is possible to debug the instantiation process of these templates which happens during the run-time of the frame processor. In order to debug XFramer code, a debugger which can handle the frame code language (i.e. C++. Java or C#) can be used. In order to debug ANGIE code, however, a specific ANGIE debugger is needed. Such a debugger is not available at present, but is planned by Delta Software Technologies.

Disadvantages

- No checking
  
  The C++-template mechanism provides additional checking capabilities, which are not available for frame processors. The frame processor is not aware of the target language and its syntax, but it is possible to implement checking capabilities manually (see also Type Library in chapter 10).

Known Uses

This technique is used in the vector generator from [EEE+03a].
There are many more possible techniques that have been identified in the examined projects and other techniques which are not more than mere ideas yet. It would exceed the size of this work to describe all of them. Therefore, this chapter outlines additional techniques.

**Alternative Rendering**
Frame technology can be used to provide alternative renderings for different platforms initiated by a high level specification processed by frame commands or expressed in a frame language. This technique was identified in the *GP-WEB* project as well as in *HyperSenses*.

**ANGIE Singleton**
This technique describes a concept of access control which is used to create a Singleton in ANGIE (see [GHJ+95]) on frame code level.

**AOP with Frames**
Frame Technology can also be used as implementation technology for Aspect Oriented Programming (AOP). This technique explains how it is possible. Please notice that the requirement of AOP for aspect unawareness, however, can not be realized with frame technology.

**Back-Tracing**
This technique shows various ways, how the origin of generated code can be traced back to the specification.

**Configuration Storage**
There are many different possibilities how configuration data can be stored during the processing of frames. This technique points out a kind of storage which is most useful in regard to memory usage and ease of access.

**Escapes**
Escapes are used in a configuration for a generator to specify features that can not be expressed in the DSL of the configuration [Cle01]. This technique provides the means of implementing escapes with frame technology.

**Generator Framework**
The term framework is meant here in the sense of the object-oriented programming. Therefore, a generator framework is a an extensible framework that allows new components to be added by third parties. It can be considered an extensible generator that has predefined interfaces for the integration of generator components. These components are realized as frames like the generator framework itself. A generator framework is part of the *SCORE Integration Suite* from *Delta Software Technology*. It allows the customers to make complex adaptations which are too complicated to be made with the graphical user interface.

**Mapping the Configuration of a Feature Diagram to XML**
This technique explains how a configuration of a feature diagram can be mapped to XML-files. The XML-files are used by the frame processors to control the generation process. Applications of this technique can be found in *ABA*, *Mini-ABA* and *GP-WEB*.
Pattern Implementation
Object-oriented patterns like the ones described in [GHJ+95] provide abstract design information which can not be expressed directly in source code. With frame technology, however, it is possible to encapsulate this kind of knowledge. This technique is an essential part of the applications PBE and HyperSenses.

Test Case Generator
A test case generator automatically builds custom test cases for a specific member of a software-system family. In addition, the tests are run automatically as soon as a system is generated. Such a test case generator can be integrated in existing frame hierarchies which are responsible for creating the software-system.

Type Library
A type library that is implemented in specification code checks type requirements of target language code in frame variables. It contains specific knowledge about the types (and casting behavior) of the target language and has therefore to be specifically implemented for each target language.

Using Frame Technology in the World Wide Web
In the GP-WEB Project, ANGIE based generators are closely coupled with web-services. This technique outlines various possibilities of how to use frame technology in the world wide web. This includes how a front-end can transfer data to a generator as well as how generators can communicate with each other using web-technologies.

XML processing
This technique discusses, how XML-files can be processed by or for frame technology, in order to take the specializations which are specified in XML-files into account by the various frame processors.
11 Conclusion and Further Perspectives

As you did see, frame technology in conjunction with GP has a lot of potential to fulfill the requirements of today's software market, especially by the application of software-system families and software factories. There are, however, many more application domains for frame technology, as the following two examples illustrate.

Aspemw_vivec1024.bmpct Oriented Programming (AOP)

AOP is also a new technology which can be used with and without the generative paradigm [Mac01]. The goal of AOP is to unitize aspects of programming which are scattered across the source code (so-called crosscutting concerns). AOP can be applied by specific generators (so-called aspect weavers) like, for example, AspectJ for Java. Frame technology is also suited for the application of AOP [Gie02a], even independent of the target language.

Model Driven Architecture (MDA)

Model Driven Architecture is an upcoming standard of the OMG for separating the abstract software model from the actual implementation on different target languages, platforms, middleware architectures, and so on. Many specialized case-tools are already available with support for this new concept, while more are still under development [OMG02]. MDA can be applied with frame technology as well [Gie02a].

These are just a few additional promising fields frame technology can be applied. Frames are still an emerging technology with the number of application domains and successful projects steadily growing.

In addition to examinations about interaction possibilities between frames and other technologies, there is still a large amount of research needed to bring frame technology to the general public. Further work based on this diploma thesis will therefore include the whole process of generative programming with frame technology explained by a detailed exemplary project. It is also necessary to provide “learning-scripts” for the specific frame processors which feature a more didactical approach and aim to be a reliable guideline for developers. A more detailed discussion is necessary about other involved technologies like XML and HUTN or the impact of and to specific programming languages. Various kinds of support tools have to be evaluated about their applicability to frame technology while the different frame processors have to be examined for their applicability to specific project types. And last, but not least there are, of course, many ideas for new techniques for frame processors as already outlined in chapter 10.
Abstraction concept
The *abstraction concept* is a specific method for the connection of frames and the handling of slots. It refers to the frame technology invented by *Delta Software Technology* and first used by the frame processor *ANGIE* [DST01]. See also section 3.2.2 “The Abstraction Concept”.

Adapter Frame
See Ancestor

Adapt concept
The *adapt concept* is a specific method for the connection of frames and the handling of slots. It refers to the frame technology invented by Basset [Bas97] and was first used by the frame processor *Netron Fusion*. See also section 3.2.1 “The Adapt Concept”.

Ancestor
1. *Ancestor* is a term used in the context of graph theory. Any other node on the path of a node to the root is called an ancestor of that node [CLR90]. That node is called descendant of the other nodes.
2. *Ancestors* (or *ancestor frames*) of a specific frame are all frames that are located above this specific frame in the frame hierarchy along its path to the root, including the root frames. An ancestor frame can also be called *adapter frame*, because it adapts other frames [Bas97, p. 99; Net02]. This term was adopted from graph theory.

Breakpoint
A *breakpoint* or *variation point* (Basset calls it just *break* [Bas97]) is a special kind of a slot, which can be created with the BREAK command of *XVCL* or *Netron Fusion*. It contains frame-text (may also be empty), which can be modified by ancestors of the current frame [Bas97, p. 340; JS00; CJ99].

Chunk
Basset's former term for “Frame” (see 4 History of Frame Technology )[Bas97].

Child
1. *Child* is a term used in the context of graph theory. A node is called the child of another node if they are connected by an edge and the path to root of that node contains the other node. This means the child node is more distant from the root than the other [CLR90]. The other node is called its *parent*.
2. A *child* (or *child frame*) is the next descendant frame of a specific frame. This means the next frame below the actual one in the frame hierarchy [Bas97, p. 99; NE02]. This term was adopted from graph theory.

Configuration
"A partially or fully specialized feature model (see specialization) can be declared to be a configuration. That means that a configuration can still contain some variability. Either it must be specified in the configuration knowledge how to treat that variability or the variability must be annotated so that a generator can decide whether this variability has to be removed during the
generation process or whether a system has to be generated which contains this variability as features that can be selected during run-time. The difference of a configuration and a feature model (specialized or not) is rather of technical nature." [Eis03]

**Configuration Data**
The data that makes up a configuration.

**Configuration Knowledge**
"The knowledge necessary to automate the assembly of system family members out of implementation components. It specifies illegal combinations of system features, default settings, default dependencies, optimizations and construction rules stating which configurations of components satisfy which configurations of features. Configuration knowledge can be implemented using generators.” [CE00]

**DAG**
DAG is the abbreviation for directed acyclic graph. All frame hierarchies are structured as DAG's.

**Descendant**
1. Descendant is a term used in the context of graph theory. A descendant of a node is a node with the other node on its path to the root [CLR90]. The other node is called its ancestor.
2. Descendants (or Descendant frames) of a specific frame are all frames that are located below this specific frame in the frame hierarchy. [Bas97, p. 99; Net02a]. This term was adopted from graph theory.

**Edge**
See graph.

**Frame**
“A component in any programming language that can be reused, not only as-is, but as adapted by other frames to fit a new application. A frame contains (1) program commands and variables and (2) frame commands and variables. The frame commands can add to or subtract from other frames' capabilities as the application requires.” [Bas97]
“Frames are code modules as objects.” [DST]

**Framing**
“Converting use-as-is information into frames” [Bas97]
Frame Code

The frame code is one of two parts of the specification for a frame processor. The other part is the target language code. Frame code consists of frame commands and their parameters only and is free of target language code (with the possible exception of small target language code containing strings used as arguments of functions within the abstraction concept). Therefore the frame code is the sum of all frame commands in a specification. **ANGIE** separates frame code and target language code. Target language code is contained in frames while most frame code is contained in functions. In frames not all frame commands are allowed (e.g. `if`, `while`). **XFramer** also allows more types of frame commands inside a target language code block. Adapt concept-based frame processors (i.e. **FPL, FrameProcessor, Netron Fusion and XVCL**) lack the concept of functions and therefore mix frame commands and target language code inside the frames. Please notice that this is a difference that can have advantages and disadvantages as well.

Frame Command

A frame command is a command for the frame processor. It allows the modification of frames [WJS+01]. Frame commands are part of frames or functions. Typical frame commands are `insert` (add), `break` (see `breakpoint`), `replace` and `remove`. Depending on the frame processor, these commands have different syntax. In **XVCL** for example, they are XML tags (e.g. `<BREAK...> some frame-text</BREAK>`) [NN01]. Many **ANGIE** and **Netron Fusion** commands start with a dot, e.g. `.add`, `.INSERT`. The abstraction concept further distinguishes between different types of frame commands. Frame commands can be divided in flags, procedural commands and function calls. Three different types of functions can be defined: intrinsic functions, library functions and user-defined functions.

Frame-Definition

Frame-definition is a synonym for frame-text used in context of **ANGIE**.

Frame Hierarchy

1. Meaning in context of the adapt concept

According to [Bas97] a frame hierarchy consists of “Frames organized into assemblies and subassemblies for the purpose of constructing software modules”. An exact definition in context of this document is provided here: A frame hierarchy forms a DAG with frames being the nodes of the graph. Frames are organized into frame hierarchies due to the application of the adapt command.

The root of a frame hierarchy is a SPC. Each frame of the hierarchy forms its own sub-hierarchy, which is a subassembly with itself as the root [Jar01]. Sub-hierarchies can also be called frame layers. Frame hierarchies are separated in layers to form logical groups of software parts representing different computational aspects. There can be for example a layer for a graphics-engine, a layer for a database, a layer for the user interface, a layer for the business logic and so on [WJS+01]. One or more frame hierarchies that form a system are called framework [CJ99].

2. Meaning in context of the abstraction concept

Frame instances are organized into the structure of a DAG because frame instances aggregate other frame instances. Such a structure is called frame hierarchy. Therefore, a frame hierarchy forms a DAG with frame instances being the nodes of the graph.

The root of a frame hierarchy which will be exported is called root frame. The term root frame applies to frames and frame instances as well. Each frame instance of the hierarchy forms its own sub-hierarchy, which is a subassembly with itself as the root [Jar01]. Sub-hierarchies can also be called frame layers. Frame hierarchies are separated in layers to form logical groups of software
parts representing different computational aspects. There can be for example a layer for a graphics-engine, a layer for a database, a layer for the user interface, a layer for the business logic and so on [WJS+01]. One or more frame hierarchies that form a system are called framework [CJ99].

**Frame Instance**
1. Meaning in context of the adapt concept
   A *frame instance* (instantiated frame) is a frame, which is already processed. The slots of the frame are filled with concrete values. Therefore, it is free of previous frame commands. The contents consists of the target language code only. The frame lost its generic character and is now specific [Bas97, p. 88].
2. Meaning in context of the abstraction concept
   A *frame instance* is the actual realization of a frame while a frame expresses an abstract concept of a specific code building block.
   A frame in the abstraction concept corresponds with a class in object-oriented programming while a frame instance corresponds with an object in object-oriented programming. A frame can be instantiated statically by declaration (*XFramer* with C++ as frame language only) or dynamically by the appropriate frame command (*CreateFrame()* in ANGIE, *new* in *XFramer*) from within a function or an adapting frame [DST02d]. Each frame can have multiple instances at once.
   A frame instance can be customized by inserting different values in its slots. Functions can even change already instantiated frames by altering their slots by inserting new values to them. In order to do this a slot can be changed like an attribute of an object in an object-oriented language. A frame instance has the same frame variables as its frame. Only slot-contents can be changed from outside the frame. The actual values of the frame variables are thereby considered the state of the frame.

**Frame Variable**
A *frame variable* is a variable used in a specification to be processed by a frame processor. A frame variable can be controlled by frame commands like a variable in any turing complete language. Most frame variables are used for embedding in slots, while usually only a few are used for calculations only.

Within the abstraction concept all slots embed frame variables while in context of the adapt concept there also other types of slots. Within the abstraction concept, frame variables can contain values of elementary data types (i.e. numerical types and strings) and frame instances. Adapt concept-based frame variables can only contain values of elementary data types, because there are specific slots for frames.

**Frame Layer**
A *frame layer* is a sub-hierarchy of a frame hierarchy. See frame hierarchy for a detailed explanation.

**Frame Library**
The term *frame library* can mean different things, depending on the specific frame system. A *frame library* can be a collection of frames for a specific purpose. In this case it is similar to a classical library which is a collection of classes and functions. However, the organization of the library content is very different. Example: There could be a library of frames for building a high-level sound-system fitting specialized application requirements. Now the library could be used once to

---

26 The term *framework* should not be mistaken as *framework* in context of the object-oriented programming. It has a totally different meaning there.
generate a high-level sound-system the application frames rely on. But it is also possible that the
generation of the sound library is coupled with the generation of the application by customizing the
SPC (see below) of the library by an application frame. Therefore the sound-system fits always the
actual application requirements.

Another additional meaning of frame library exists within the context of the adapt concept. Here a
frame library holds templates or prototypes of SPCs. With SPC templates it is possible to create an
SPC just by setting a few parameters of the SPC template [Net02a].

**Frame Processor**

A *frame processor* is a tool that takes frames and functions (functions are only available in the
abstraction concept) as input and processes them with the goal of source code as output. This source
code can be used by an appropriate compiler [Bas97, JS00]. Adapt concept-based frame processors
directly produce the source code. Abstraction based frame processors, however, produce a (custom)
generator first which in turn produces the source code. This custom generator is specialized to a
specific domain and can take further configuration data into account. The available adapt concept-
based frame processors are FPL, *FrameProcessor*, Netron Fusion and the XVCL processor. The
available abstraction concept-based frame processors are *XFramer* and ANGIE. Both names mean a
product rather than exactly match the definition of a frame processor. *XFramer* is only part of a
frame processor. The combination of *XFramer* and an appropriate compiler for the frame code
language can be considered a frame processor. ANGIE consists of two parts, a compiler and an
interpreter. The ANGIE compiler is the actual frame processor while the ANGIE interpreter is part
of the custom generator. Frame processors can be programming language dependent or
independent. It is also possible to produce other text-oriented output instead of source code of a
programming language, like documentation or help files.

**Frame Repository (Adapt Concept only)**

The term *frame repository* is only defined in context of the adapt concept. A *frame repository* is a
collection of frame libraries [Bas97]. It can also consist of only one library. Therefore the terms
frame repository and frame libraries are sometimes used synonymously. Additionally a frame
repository can hold meta-information like design templates or other data the system can provide or
has use for.

**Frame-Text**

Each frame consists of a frame-name and a *frame-text* and may also contain other components, e.g.
a human-readable description for aiding a developer to understand the frame's purpose. *Frame-text*
denotes the lines of text that hold the functionality of the frame. This functionality consists of
frame-commands and plain text in the target language (for example Java Code, C++ Code, plain
English as part of an applications user manual, and so on) or just of one of these [Bas97, p. 99 + p.
345]. Please notice, that in *XFramer* BEGIN_FRAME_TEXT only means the beginning of the target
language block. A synonym for frame-text is *frame body*. Frame-definition is another synonym for
frame-text, used in context of ANGIE.

**Framework**

For the purpose of this paper the term *framework* is not used in the common meaning of the object-
oriented programming. In frame technology, a *framework* consists of one or more frame hierarchies
[CJ99] and represents a software-system or a software-system family and its generic architecture.
Function
A function in context of frame technology is a specific type of a frame command within the abstraction concept. Functions are further divided into intrinsic functions, library functions and user-defined functions.

Generative Domain Model
“A model of a system family that allows the automatic generation of family members from abstract specifications. It consists of a means of specifying family members, the implementation components from which each member can be assembled, and the configuration knowledge.” [CE00]

Generative Programming
“Generative Programming (GP) is 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 automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge. The generated products may also contain nonsoftware artifacts, such as test plans, manuals, tutorials, maintenance and troubleshooting guidelines, and so on.” [CE00]

Frames are a technology, that can be used for Generative Programming.

Generator
“A generator is a program that takes a higher-level specification of a piece of software and produces its implementation. The piece of software could be a large software-system, a component, a class, a procedure, and so on.” [CE00]

A frame processor is a generator that works on frames (see Frame Processor above)27. In the context of the abstraction concept the sum of all frame hierarchies with their proper functions making up a specific system are often called custom generator or just the generator.

Inter-application Variability
Reuse can be applied on different software-systems of a software-system family. One member can have a feature that varies from that of another member of the family. Often these features have common parts. The decision which option(s) of the variability has/have to be chosen is made during the construction time of a system. Such a variability that appears between members of a software-system family is called inter-application variability [CE00]. Frame technology can help to implement the various options of the variability based on common parts and therefore realizes reuse.

Intra-application Variability
Reuse is a concept that works not only for software-system families, but also for single software-systems. Intra-application variability [CE00] is the variability that appears within a single software-system. Specific features of such a variability can be chosen during run-time. Frame technology can help to implement various options of the variability based on common parts and therefore realizes reuse.

Leaf
Leaf is a term used in the context of graph theory. A leaf is a node with no children [CLR90].

---

27 The definition used here differs from Bassets [Bas97], who distinguishes between generators, compilers, and frame processors. Applying the definition used here, compilers and frames processors are specific generators.
12 Glossary

Node
Node is a term used in the context of graph theory. A node is a vertex of a rooted tree [CLR90].

Parent
1. Parent is a term used in the context of graph theory. A node is called the parent of another node, if they are connected by an edge and the path to root of that node does not contain the other node. This means the parent node is closer to the root than the other [CLR90]. The other node is called its child.
2. A parent (or parent frame) is the next ancestor frame of a specific frame. This means the next frame above the actual one in the frame hierarchy along its path to the root [Bas97, p. 99; NE02]. This term was adopted from the graph theory.

Problem Space
“The terminology used to specify the members of a system family.” [CE00]

Product Line
“A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market.” [Wit96, cited from CE00]

Product Family
“A product family is a group of products that can be built from a common set of assets.” [Wit96, cited from CE00]

Root
Root is a term used in the context of graph theory. A root is a distinguished vertex of a rooted tree. It is the only node without a parent [CLR90].

Rooted Tree
Rooted tree is a term used in the context of graph theory. “A rooted tree is a (free) tree in which one of the vertices is distinguished from the others.” [CLR90]

Sibling
Frames called siblings, are child frames of the same parent frame [Bas97, p. 99]. The mathematical definition of siblings is analog. Siblings are any two nodes with the same parent [CLR90].

Slot
A slot is an insertion or variation point (that is any point at which frame can be changed) located in a frame, which can become a numerical value, a string, or a sub-frame (instance) [Sch01b]. The principle is similar to templates used in other programming languages. However, the term slot is of more abstract nature, representing a general concept rather than a specific technique. There are different types of slots depending on the applied concept and frame processor. Examples for slots are breakpoints and embedded frame variables [JS00; CJ99].

Solution Space
“The implementation components of a system family together with their possible configurations.” [CE00]
Specialization
1. *Specialization* is the process of selecting variants of a software-system family. It is usually done by successively choosing values for the parameters that express the different variabilities. That is, specialization is removing variability. By the application of generative programming, specialization works on feature models.
2. A *full specialization* does no longer contain any variability. It is a specific variant of a software-system family while a *partial specialization* refers to a set of members of a software-system family.
See also configuration.

Specification
A generator takes a high level *specification* and processes it in order to generate source code [Bas97; CE00]. A specification is therefore always the input of a generator. However, depending on which generator is viewed the term specification has a different meaning. This is, because every generator expects another type of input.

If the point of view is on the frame processor (which is indeed a generator), the specification is the sum of all frames, functions (functions are only available in the abstraction concept) and internal configuration data (e.g. default values, hard-coded parametrization) to be processed by the frame processor. Such a specification can also be divided in *frame code* and *target language code*. Within the abstraction concept the custom generator is implemented with this specification.

Within the abstraction concept, however, the output of a frame processor is also a generator. It is a custom generator specialized to a specific domain and not a general one like the frame processor. If this generator is the point of view, its specification consists of configuration data only (see *configuration*). This configuration data is not included in the specification of the frame processor and is therefore called *external configuration data*.

If the point of view is a generator that is the combination of a frame processor and an abstract based custom generator, the specification consists of both - frames, functions and internal configuration data as well as external configuration data.

Sub-frame
When a frame adapts or aggregates other frames, it is possible to say, that the frame consists of the other frames. The frames another frame consists of are called *sub-frames* of this frame. In fact, a sub-frame is the same as a child frame, but the term sub-frame is used in another context.

Subcomponent Frame
See Sub-frame.

Subtree
A *Subtree* is a tree, which is part of a larger tree, one with more vertices. In case of a rooted subtree it has its own root, which is different from the root of main *tree* [CLR90].

System Family
See Product Family.

Target Language Code
The *target language code* is one of two parts of the specification to be processed by a frame processor. The other part is the *frame code*. The target language code consists, as the name implies, of code of the target language only. Depending on the frame processors and frame technology
concept, there may be a separation of concerns that dictates which frame commands are allowed to stand in-between target language code (see also frame code).

Target language code can virtually consist of anything expressible in strings. This includes programming languages (like C++, Java, COBOL, and so on) and markup languages (like HTML or XML) as well as even natural languages (like English or German). Some frame processors, however, provide special features for specific programming languages (e.g. Netron Fusion for COBOL).

**Target Language Code Block**

A *target language code block* is a cohesive block containing target language code. A target language code block in an adapt based frame processor is the whole frame-text. There is no limitation on allowed frame commands inside the block. Therefore the term target language code block is rarely used in context of the adapt concept where the (in this context synonym) term frame-text is more usual. Within the abstraction concept, however, inside a target language code block usually not every frame command is allowed. In XFramer a target language code block is called frame-text (which is, however, not frame-text in the general meaning) and is enclosed in the commands `BEGIN_FRAME_TEXT()` and `END_FRAME_TEXT()`. In ANGIE a target language block is a whole frame body (also frame text) without frame variable declarations. A frame is the only place in ANGIE where a target language code block is allowed.

**Tree**

*Tree* is a term used in the context of graph theory. A (free) tree is a unidirected graph, which is connected (there is a path between each two vertices) and acyclic [CLR90].

**Variation Point**

See Breakpoint
13 References


[But02] G. Butler. Object-Oriented Application Frameworks. Talk at Concordia University, Montreal, 2002


[Cla94] B. K. Clark. Domain Specific Software Architectures (DSSA). Presentation at University of Southern California, Center for Software Engineering, June 1994, see http://sunset.usc.edu/TechRpts/Presentations/Brad_ArchWkshp.ps

[Cle01] C. Cleaveland. Program Generators with XML and Java. Prentice-Hall, 2001, see


[CSN02] Canada’s SchoolNet. Information on Combinations of a Set. See http://www.schoolnet.ca/vp-pv/amof/e_combI.htm

13 References


[DST02a] Delta Software Technology GmbH. Das ANGIE Frame-System. internal paper, 2002


[DST02c] Delta Software Technology GmbH. Pattern By Example. April 2002, see http://www.d-s-t-g.com/


[EEE+03b] U. W. Eisenecker, M. Emrich, C. Endler, M. Schlee. Generative Implementation Techniques for C++, Frames, and Graphical User Interfaces. University of Applied Sciences Kaiserslautern, Zweibrücken, Germany, as part of the PoLiTe-Project from Fraunhofer Institute for Experimental Software Engineering (IESE), 2003, see http://www.polite-project.org

13 References

[Eis02b] U. W. Eisenecker. Generative Programmierung und Software-Ökonomie. GP-WEB Workshop at University of Applied Sciences Kaiserslautern, Zweibrücken, Germany, June 2002


[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reuseable Object-Oriented Software. Addison-Wesley, 1995.


[Gie02c] C. Giese. ANGIE Kurs. internal workshop, 2002


[Hen99] S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. Department of Computer Science & Engineering, University of
13 References

Nebraska-Lincoln, November 1999, see
http://www.cse.unl.edu/~scotth/papers/TOSEM-henninger97.pdf


References


D. Liu and H. Mei. Mapping requirements to software architecture by feature-orientation. Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Proc. of the Second International Workshop “From SoftWare Requirements to Architectures” (STRAW'03) as part of International Conference on Software Engineering 2003 (ICSE'03), Portland, Oregon, 2003, see http://se.uwaterloo.ca/~straw03/


MetaCase Consulting. The MetaCase Consulting Internet Presence. See http://www.metacase.com

J. McLeod. The Card Games Web Site. 2002, see http://www.pagat.com


Netron Inc. The Netron Internet Presence. 2002, see http://www.netron.com/


Object Management Group Inc. The OMG Internet presence. See http://www.omg.org
13 References


[SBa] Y. Smaragdakis and D. Batory. Scoping Constructs for Program Generators. Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, see http://www.cs.utexas.edu/users/smaragd/research.html


References


[Sle] M. Schlee. XFramer web site, see http://www.geocities.com/mslerm/xframer


[ZJS01] H. Zang, S. Jarzabek and S. M. Swe. *XVCL Approach to Separating Concerns in Product Family Assets*. Department of Computer Science, National University of
13 References

Singapore, 2001, see
Index of Figures

Figure 2.1 Elements of the generative domain model 10
Figure 2.2 Domain Implementation Technologies for Generative Programming 15
Figure 2.3 Software Development based on Domain Engineering 17
Figure 2.4 Feature Diagram from FODA 18
Figure 2.5 Feature diagram with mandatory features 21
Figure 2.6 Feature diagram with optional features 21
Figure 2.7 Feature diagram with alternative features 22
Figure 2.8 Feature diagram with or-features 22
Figure 2.9 Exemplary feature diagram of a car 23
Figure 2.10 Figure diagram of a car in GME2000 24
Figure 2.11 Figure diagram of a car in AmiEddi 25
Figure 3.1 Basic principle of a frame processor 28
Figure 3.2 Organization of frames 31
Figure 3.3 Undirected graph with cycle 32
Figure 3.4 Directed graph without cycle 32
Figure 3.5 Directed graph with cycle 32
Figure 3.6 How to use a SPC 37
Figure 3.7 Feature diagram for vector classes extended by unit vectors 38
Figure 3.8 Export of adapt concept-based frame hierarchies 41
Figure 3.9 One possible implementation of the feature diagram in Figure 3.7 42
Figure 3.10 Instantiation of frames 44
Figure 3.11 Principle of an abstraction based frame processor 46
Figure 3.12 Single-phased generation 48
Figure 3.13 Multi-phased generation 49
Figure 3.14 XML file for external configuration, specified in the graphical generator of ABA 50
Figure 3.15 Graphical configuration of ABA 50
Figure 3.16 UML-classdiagram of the result of a "vector"-generator using the abstraction concept-based method 51
Figure 3.17 An example of a method using the abstraction concept 52
Figure 4.1 History of frame technology 55
Figure 5.1 Technology projection for generative programming using frames 59
Figure 5.2 Transformation of front-end configuration data in context of the adapt concept 61
Figure 5.3 Transformation of front-end configuration data in context of the abstraction concept 62
Figure 5.4 Feature diagram for simple vector classes 63
Figure 5.5 The "Vector3D" class in UML 64
Figure 5.6 The "Vector2D" class in UML 64
Figure 5.7 The “StandardVector” and “UnitVector” class in UML 69
Figure 5.8 A small vector library in UML 70
Figure 6.1 The ANGIE architecture 72
Figure 6.2 FPL in Eclipse 74
Figure 6.3 Code generation with XFramer 79
Figure 6.4 Anatomy of a Frame in XFramer 80
Figure 6.5 XVCL code from Listing 3.6 in jEdit 82
Figure 7.1 An all object-oriented implementation of a vector library as UML-classdiagram 90
Figure 8.1 Principle of the ABA prototype 98
Figure 8.2 GP-WEB Frontend 99
Figure 8.3 Pattern Composer 100
Figure 8.4 Meta Composer 101
Figure 8.5 HyperSenses components 101
Figure 8.6 Principle of the SCORE Integration Suite 102
Figure 9.1 Principle of a Generator-Generator 111
Figure 9.2 Principle of a Generator-Generator within the Abstraction concept 111
Figure 9.3 Feature Diagram of a Meta-Generator 112
Figure 9.4 Feature Diagram of a Generated C++ Header Generator 113
Figure 9.5 Frame hierarchy of inlining 119

Index of Tables
Table 3.1 Corresponding terms of graph theory and frame technology 35
Table 3.2 Examples of important commands of adapt concept-based frame processors 43
Table 3.3 Examples of important commands of abstraction concept-based frame processors 53
Table 6.1 Important commands of ANGIE 72
Table 6.2 Important commands of FPL 75
Table 6.3 Commands of FrameProcessor 76
Table 6.4 Important commands of Netron Fusion 77
Table 6.5 Important commands of XFramer 81
Table 6.6 Important commands of XVCL 84
Table 6.7 Quick comparison of available frame processors 87

Index of Listings
Listing 3.1 A XVCL frame with Java code 30
Listing 3.2 An ANGIE frame with C++ code 30
Listing 3.3 A SPC in XVCL 33
Listing 3.4 An ANGIE root frame with scoped variables (decorator.cdm) 33
Listing 3.5 A frame for generating vectors (Vector.XVCL ) 39
Listing 3.6 A frame for generating unit vectors (UnitVector.XVCL ) 40
Listing 3.7 SPC of the Vector-Interface (VectorInterfaceSPC.S) 41
Listing 3.8 SPC of the standard vector (UnitVectorSPC.S) 42
Listing 3.9 SPC of the unit vector (UnitVectorSPC.S) 42
Listing 5.1 ANGIE code module for the generation of vector classes in Java (Vector.cdm) 66
Listing 5.2 Output of the ANGIE vector code module for Vector3D (Vector3D.java) 66
Listing 5.3 Output of the ANGIE vector code module for Vector2D (Vector2D.java) 67
Listing 5.4 Output of the ANGIE vector code module for a dynamic Vector(DynamicVector.java) 68
Listing 6.1 Example for a FrameProcessor frame with a variation point (main.frame) 76
Listing 6.2 Example for a FrameProcessor frame, adapting “main.frame” (main.spc) 76
Listing 6.3 Output from frame “main.spc” of Listing 6.2 (main.c) 76
Listing 6.4 Example of a Netron Fusion frame 78
Listing 6.5 C++ Swap-Generator in XFrmer using Java as frame code language 80
Listing 6.6 Swap-Generator already processed by XFrmer (SwapGenerator.java) 81
Listing 6.7 A function for swapping integer values in C++ (swap.cpp) 81
Listing 6.8 A XVCL-frame for the generation of vector classes in Java (Vector.XVCL) 85
Listing 9.1 A XVCL-frame generating some simple text (Emulation.XVCL) 104
Listing 9.2 An adapting XVCL-frame generating some simple text (Emulation.S) 104
14 Indices

Listing 9.3 Textual output of the emulation example (Emulation.txt) 104
Listing 9.4 Textual output of the emulation example using insert-after (Emulation.txt) 104
Listing 9.5 Textual output of the emulation example using insert-before (Emulation.txt) 104
Listing 9.6 Textual output of the emulation example using insert (Emulation.txt) 104
Listing 9.7 ANGIE-realization of the adapt emulation technique 105
Listing 9.8 Constant Folding Example – NoverK (CFExample.java) 107
Listing 9.9 Constant Folding Example – NoverK – Specification code in XFramer (CFExample.cpp) 108
Listing 9.10 Constant Folding Example – NoverK – generated code (CFExample.java) 108
Listing 9.11 Generator-generator in ANGIE 114
Listing 9.12 ANGIE configuration file for the generator generator 115
Listing 9.13 Batch file calling ANGIE with the generator-generator code module 115
Listing 9.14 A generated implementation generator 115
Listing 9.15 A generated header generator 116
Listing 9.16 Batch-file calling ANGIE with the header generator code module 116
Listing 9.17 A generated header file 116
Listing 9.18 Batch-file calling ANGIE with the implementation generator code module 116
Listing 9.19 A generated implementation file 116
Listing 9.20 Inline Example 1 (Inline Example.cdm) 121
Listing 9.21 Inline Example 1 – calcVolume – generated with inlining off (INLINE.C) 121
Listing 9.22 Inline Example 1 – calcVolume – generated with inlining on (INLINE.C) 121
Listing 9.23 Inline Example 2 – calcVolume (Inline Example 2 - VolumeCalc.cdm) 124
Listing 9.24 Inline Example 2 – calcVolume – generated code (INLINE.C) 124
Listing 9.25 Inline Example 2 – calcVolume – Implementation Frame (calcVolumeImpl.xvcl) 124
Listing 9.26 Inline Example 2 – calcVolume – Function Header Frame (calcVolumeWithHeader.xvcl) 125
Listing 9.27 Inline Example 2 – calcVolume – Calling Frame (calcVolumeCall.xvcl) 125
Listing 9.28 Inline Example 2 – calcVolume – Main Frame (main.xvcl) 125
Listing 9.29 Array initialization using a loop 128
Listing 9.30 A generator featuring the Loop Unrolling Technique 129
Listing 9.31 Generated code of the Loop Unrolling from Listing 9.27 129
Listing 9.32 A generator featuring the Loop Unrolling Technique with the switch ability 130
Listing 9.33 The generator from Listing 9.29 with a better maintainability 131
Listing 9.34 Generated code of the Loop Unrolling technique – not unrolled 131
Listing 9.35 Generated code of the Loop Unrolling technique – unrolled 131
Listing 9.36 Loop Unrolling with XFramer 132
Listing 9.37 C++ Function Template Swap 134
Listing 9.38 XFramer Swap 134