FAST product-line architecture process

Maarit Harsu
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, 33101 Tampere
e-mail: `firstname.lastname at tut.fi`

Contents

1 Introduction

This report gives an overview of FAST (Family-Oriented Abstraction, Specification, and Translation) process. It has been introduced at AT&T by David Weiss and further developed at Lucent Technologies Bell Laboratories. The report is mainly based on the book cornerning FAST [WL99]. In addition, there are other publications about FAST [ADD$^+$00, ADH$^+$00, CHW98].

FAST applies product-line architecture principles into software engineering process. Thus, a common platform is specified to a family of software products. The platform is based on the similiraties between several products close to each other. The variabilities among the members of a product family can be implemented with different variation techniques such as parametrization or conditional compilation. (A more precise representation about variability in product-line architectures can be found, for example, in [Myl02].) The purpose of FAST is to make software engineering process more efficient by reducing multiple work, by decreasing production costs, and by shortening time-to-market.

FAST process can be applied in a consistent and disciplined way. This is called PASTA (Process and Artifact State Transition Abstraction) model. PASTA model provides a path to follow during FAST process. It determines a set of steps that can succeed the current step. Thus, it gives precise instructions to follow, but still supports individual choises to make during the process. The purpose of PASTA is to make the software engineering process easy to iterate and reuse in future processes.

The purpose of this report is to introduce FAST process and to show how FAST can be applied in industrial companies. As an example of applying FAST, mobile terminals are considered. The example presents how FAST suits in developing different terminals that have variant properties and restrictions.

This report proceeds as follows. Section 2 gives an overview about FAST process, and Section 3 considers the principles behing FAST. Next three sections describe the activities of FAST: Section 4 concentrates on domain qualification, Section 5 on domain engineering, and Section 6 on application engineering. Section 7 introduces PASTA model, and Section 8 depicts an example of applying FAST process. Finally, Section 9 concludes the topic by comparing FAST to other software engineering processes.

2 Overview of FAST

FAST (Family-Oriented Abstraction, Specification, and Translation) is a development process for producing software in a family-oriented way. It separates product-line engineering process into two main parts. One step concentrates on providing the core assets including the environment for implementing each product. The other step utilizes the environment in the production of different software products belonging to the family.

Software engineering in an efficient way is usually difficult. On one hand, careful engineering in software process is essential to be able to meet the customer requirements. Thus, software products should be reliable and easy to use and maintain. On the other hand, markets and competitors require rapid production of software. These two requirements, rapid production and careful engineering are difficult to achieve at the same time. However, product-line engineering via FAST tries to resolve the dilemma and to achieve both the goals.

FAST process can be divided into the following subprocesses (shown in Figure 1):

- domain qualification to identify families worthy of investment,
- domain engineering to invest in facilities for producing family members,
- application engineering to use those facilities to produce family members rapidly.

These subprocesses are considered in more detail in Sections 4, 5, and 6.

Besides the division into subprocesses, FAST distinguishes between activities, artifacts, and roles. Each subprocess (shown in Figure 1) consists of different activities, produces different artifacts, and are performed by people acting in different roles. However, these two divisions are not totally separated. FAST activities actually comprises the aforementioned subprocesses.

This section considers the division into roles, artifacts, and activities. Each item of the division is described in an own subsection.

2.1 Roles

When adopting FAST, the organization should have two different groups of engineers: one for domain engineering and the other for application engineering.

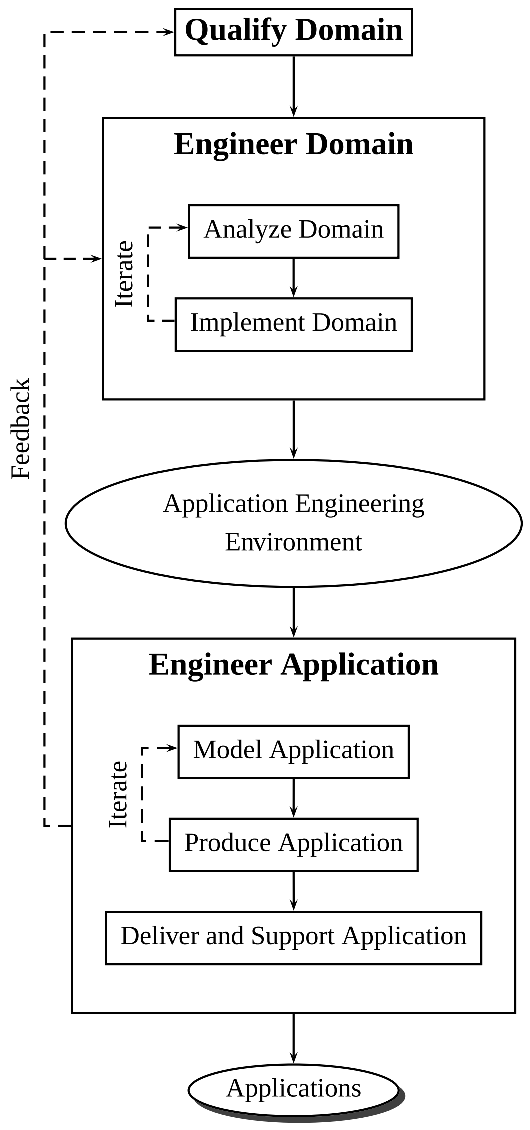

Figure 1: FAST process pattern [WL99]

Domain engineers take care of the evolution of the family and control that the investment in the family stays paying. Application engineers produce family members. They are in contact with customers to be able to satisfy their requirements.

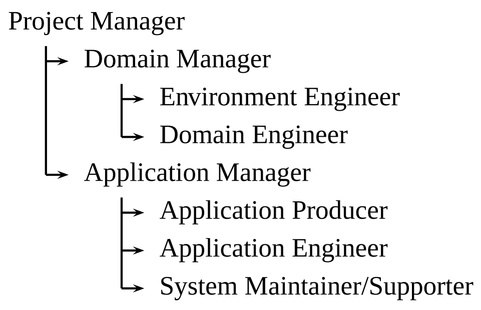

Figure 2: FAST role hierarchy [WL99]

FAST roles are shown in Figure 2. The roles are arranged into a hierarchy that can be a basis for an organizational hierarchy.

2.2 Artifacts

Like roles, also artifacts are presented as a hierarchy (see Figure 3). Each artifact consists of the subartifacts below it. For example, an application consists of an application model, application documentation, and application code. Note that there are a connection between the Figures 2 and 3. Persons responsible of the domain part produce environment artifacts, while application persons take care of artifacts concerning application.

Entry and exit conditions used in PASTA model (considered in Section 7) are defined in terms of artifacts. As an example of such a condition, the design of an application modeling language is not possible until commonality analysis report has been reviewed. However, these conditions are not shown in Figure 3.

2.3 Activities

Like roles and artifacts, also activities are shown as a hierarchy in Figure 4. Note that many activities and artifacts that are part of FAST process can also belong to

Family Artifact

- Environment
  - Domain Model
    - Economic Model
    - Commonality Analysis
    - Decision Model
    - Family Design
    - Composition Mapping
    - Application Modeling Language
    - Toolset Design
    - Application Engineering Process
  - Domain Implementation
    - Library
    - Generation Tools
    - Analysis Tools
    - Documentation
- Application
  - Application Model
  - Application Documentation
  - Application Code

Figure 3: FAST artifact hierarchy [WL99]

other software development processes. However, FAST also contains parts that are unique to this particular process. One of such parts is the design of the family.

FAST
- Qualify Domain
- Engineer Domain
  - Analyze Domain
    - Define Decision Model
    - Analyze Commonality
    - Design Domain
    - Design Application Modeling Language
    - Create Standard Application Engineering Process
    - Design Application Engineering Environment
  - Implement Domain
    - Implement Application Engineering Environment
    - Document Application Engineering Environment
- Engineer Application
  - Model Application
  - Produce Application
  - Provide Delivery and Operation Support
- Manage Project
- Change Family

Figure 4: FAST activity hierarchy [WL99]

Some activities may proceed concurrently while some others may require a particular order. For example, domain engineering and application engineering can be performed in parallel. Domain engineers are able to refine parts of application engineering environment at the same time when application engineers are using other parts of it. PASTA model (described in Section 7) specifies the order of FAST activities more precisely.

Figure 1 depicts the three main activities that will be considered in the sequel of this report in more detail. However, Figure 4 shows also two additional activities: managing a project and changing a family. Project management means managing the work concerning identification and satisfaction of customer requirements. Management is based on application engineering environment. It includes normal management tasks such as scheduling, allocating, and monitoring. Changing a family means controlling and managing the evolution of the family. It includes both allowing changes to the family members and making the generation of the members easier and more efficient. For both of these additional activities, PASTA model provides no detailed guides. Instead, each organization can apply their own policies in these activities.

Together the three hierarchies (see Figures 2, 3, and 4) clarify what people acting in different roles must do, what they must produce, and when they must produce it. There is an evident relationship between the hierarchies. For example, a domain engineer is responsible for domain engineering and produces a domain model.

Note that the figures show some levels of details for each item. However, each item could be viewed in more or less detail when necessary.

3 Principles of FAST

This section introduces the assumptions behind FAST and the ideas and methods FAST relies on. Finally, the most important ideas are considered in more detail.

3.1 Assumptions

FAST is based on three assumptions or hypotheses listed below:

- redevelopment hypothesis,
- oracle hypothesis,
- organizational hypothesis.

According to the redevelopment hypothesis, software development is very often redevelopment. In many cases, it consists of creating new variations on existing software systems. Usually, there are more similarities than differences between variations.

According to the oracle hypothesis, it is possible to predict the changes that are likely to concern a software system. Future changes can be derived from earlier changes. For example, software engineers familiar with telephone switches can have the experience that different customers will want to use different billing algorithms.

The organizational hypothesis concerns both software and software development organization. Each of them can be organized to take into account of predicted changes. Predicted changes can be made independently of other type of changes. The purpose is to make predicted changes refer only to a minimal subset of system modules. For example, software for telephone switches can be designed such that billing algorithms can be changed independently of the other system.

Together these assumptions suggest that it is important and useful to find a family among similar programs. The commonalities between several programs reveal a family, while their variabilities show the boundaries of the family.

3.2 Ideas and methods

There is a variety of ideas and methods concerning software engineering. FAST is based on some of the most succesfull ideas that are listed below:

- predecting expected changes to a system over its lifetime,
- separating concerns,
- designing for change using abstraction and information hiding,
- formally specifying and formally modeling systems,
- creating application modeling languages for specifying family members,
- composing software from adaptable, reusable components,
- designing process and product concurrently,
- building compiler compilers,
- using template-based reuse,
- restricting variability to gain efficiency.

FAST integrates the above ideas and methods into a process to develop tools and assets that are applied to produce the members of a family. The first three ideas are key to FAST, and thus, they are considered in more detail in the following subsection.

3.3 Focusing the key ideas

Abstraction is related to variability. With abstraction, it is possible to provide several variant ways to implement a task. Such a task can be, for example, communication with a device. The decision about the communication way can be hidden by the abstraction. The abstraction provides an interface to the users to access the services, but conceals the details of how to control a device.

Figure 5 shows how abstractions are used in the FAST process. Domain engineers create abstractions to be used in designing the application modeling language and in designing the family for the domain. Abstractions can be used to create a library consisting of adaptable components. These components can be exploited in generating family members.

Information hiding uses abstractions to conceal those decisions that are most likely to change. An information hiding module provides an abstract interface via which users can refer to the services of the module. The abstract interface separates the concern of which services the module offers from the concern of how

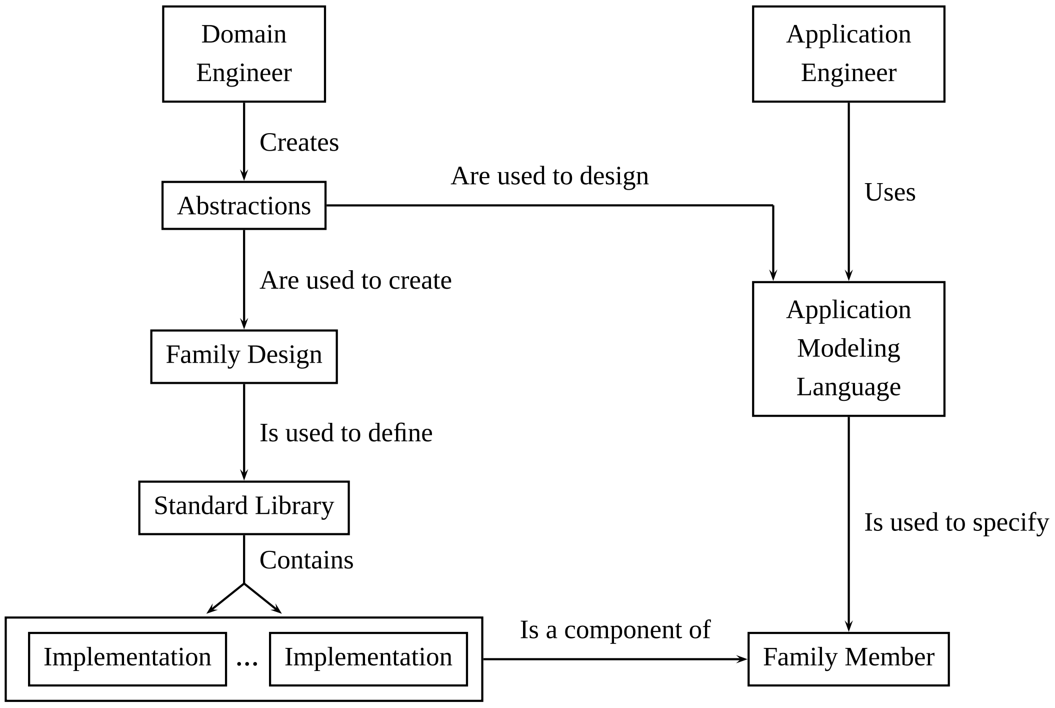

Figure 5: Use of abstractions in the FAST process [WL99]

those services are implemented. Information hiding is used to implement the variabilities of a family.

Separation of concerns [Aks96, HL95] means division of software into parts according to its different aspects. In addition to the basic algorithm, a program may have other concerns such as synchronization or real-time constraints. Separation of these concerns is important, because it supports reuse and management of variabilities between different programs. Separation of concerns has been obeyed in aspect-oriented programming [KIL$^+$96]. Both of these concepts (separation of concerns and aspect-oriented programming) are related to product-line architectures, because features needed in different software products can be composed of different sets of aspects [Gri00].

Change prediction can be based on earlier changes. The change history of a software family is a starting point for prediction. By examining past changes, it is possible to find such parts of the software that are most often been the targets of changes. These parts are important from two aspects. On one hand, they are the places where new features are implemented. On the other hand, they are bottlenecks where the most part of the change effort has been invested.

In addition to the earlier changes, information about change prediction can be obtained from the people whose business is to predict marketplace and technological changes. Moreover, people who have worked a long time with the software and are experts of the domain can provide valuable information.

4 Domain qualification

Domain qualification analyzes a software family from an economic perspective. It estimates the number and value of family members and the cost to produce them.

According to FAST, investment in domain engineering to find out a product family pays back in application engineering when producing family members (see large arrows in Figure 7). Figure 6 depicts this assumption. It shows two lines describing two situations. Line A corresponds the situation with no domain engineering. In this case, the cost of producing a new member of the family is constant, denoted by C_T. (If the cost of different products variates, C_T can be considered the average cost over the products of the family.) In each case, the cost of producing N family members is $N * C_T$.

Line B shows the situation with domain engineering. Let I be the cost of domain engineering. After this investment, producing a family member is more efficient, denoted by C_F. In this case, the cost of producing N family members is $I + N * C_F$. In order to get the domain engineering profitable, C_F should be less than C_T. If this holds, saving per family member (without considering I) is $C_T - C_F$. For N family members, saving is $N * (C_T - C_F)$. To make domain engineering paying, N should be large enough such that $I < N * (C_T - C_F)$. For example, in Figure 6, payback occurs after producing three family members.

The lines shown in Figure 6 are different for different domains. For some product families, investment in domain engineering provides greater degree of automation in the production of family members. Thus, in different domains, the payback point may occur after producing different amount of family members.

Besides variation in different domains, automation varies in different phases in application engineering. For example, a part of application engineering consists of interaction with the customers to determine the requirements for a family member. These parts cannot be automated. However, some other phases of application engineering enable automation. Moreover, even achieving little automation is paying because careful domain engineering makes the domain more structured to accomodate changes.

Domain engineering still has a risk to failure. It is possible that the investment does not provide better adjustment to changes than any other software development technique. However, in unsecure situations, investment can be increased in several steps. The ability to predict changes can be learnt by experience, and change predictions are based on earlier changes. First with a little experience, it is

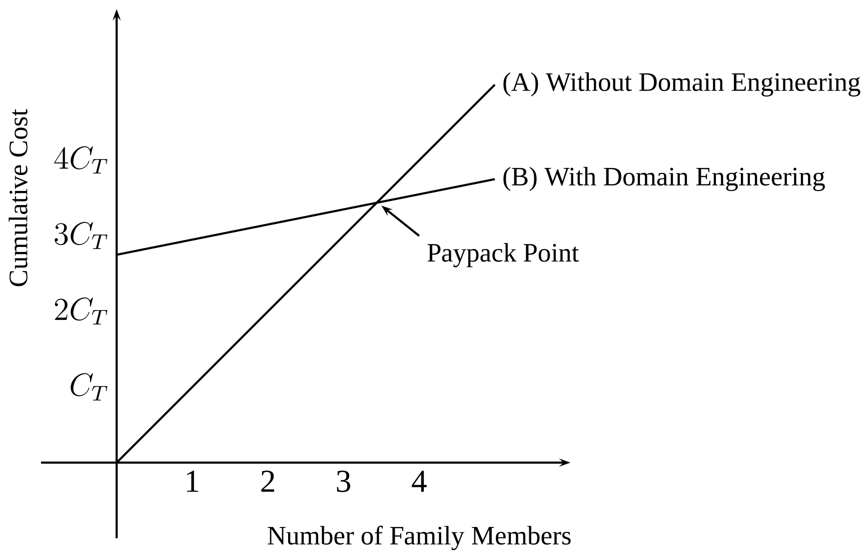

Figure 6: Cost of producing family members [WL99]

reasonable to put only a low investment on the production facilities for a family, and see how well the predictions come true. When confidence on the prediction ability grows, it is possible to return along the feedback loop in Figure 7, and increase the investment in the production facilities. In this way, the system becomes more robust even to unpredicted changes.

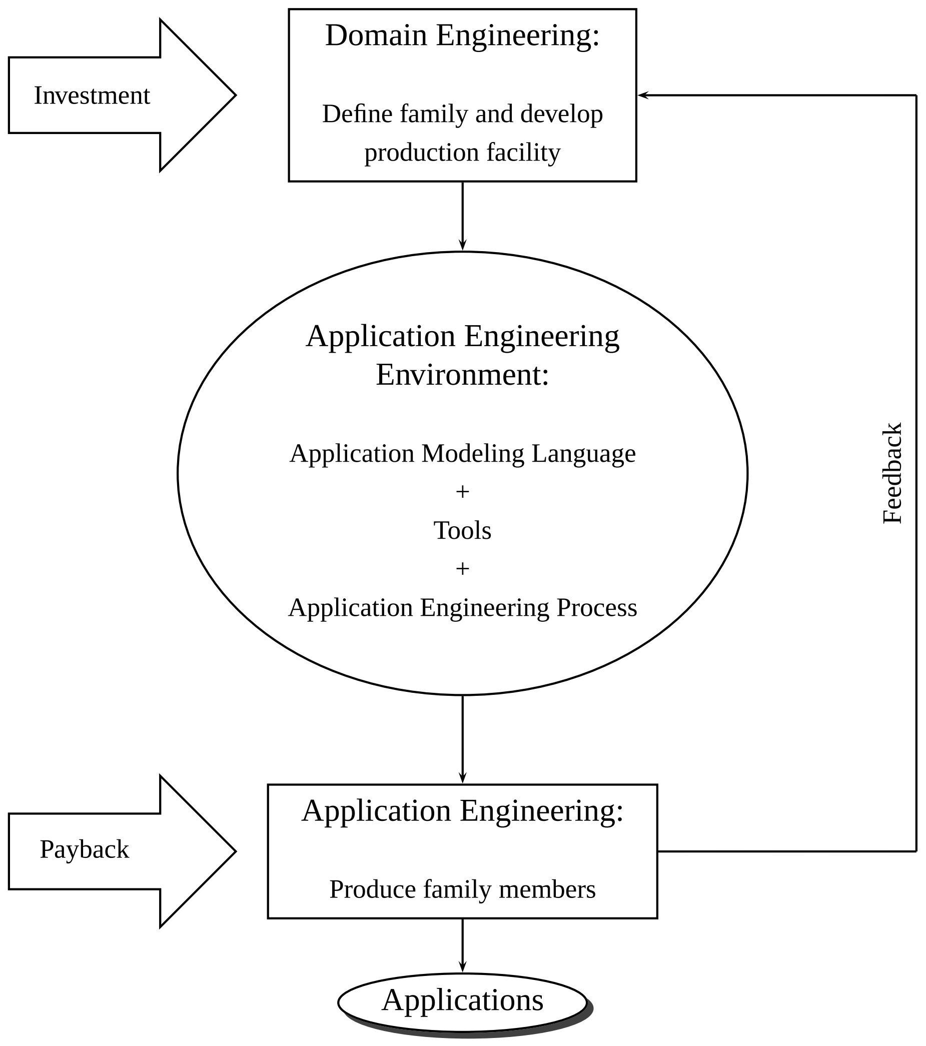
Investment
Domain Engineering:
Define family and develop production facility
Application Engineering Environment:
Application Modeling Language
+
Tools
+
Application Engineering Process
Feedback
Payback
Application Engineering:
Produce family members
Applications

Figure 7: FAST process with investment and payback [WL99]

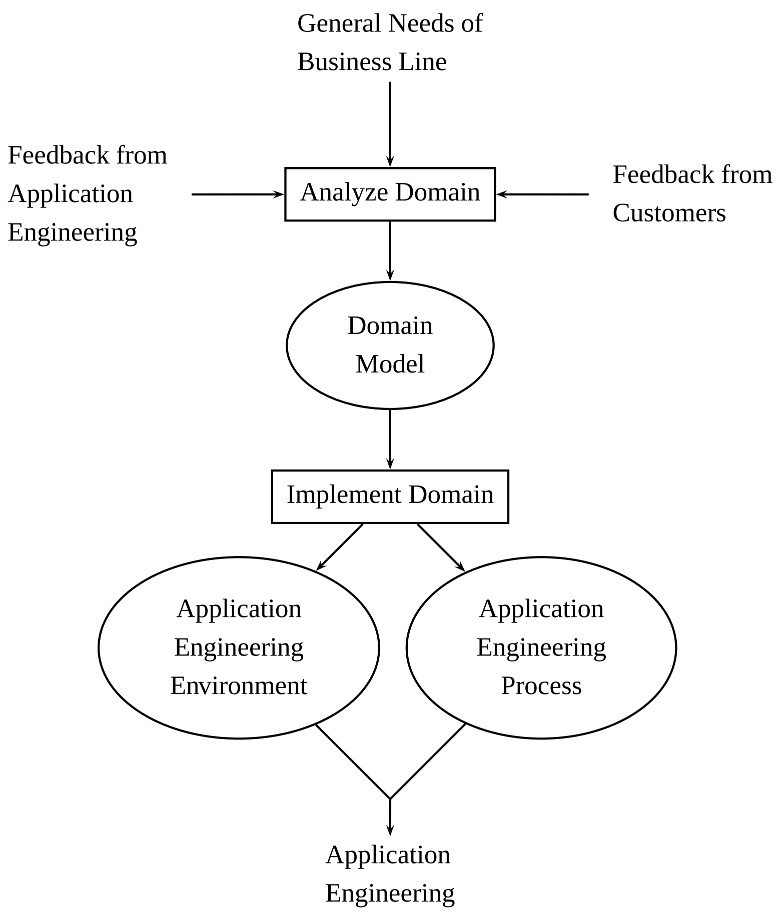
General Needs of Business Line
Feedback from Application Engineering
Analyze Domain
Feedback from Customers
Domain Model
Implement Domain
Application Engineering Environment
Application Engineering Process
Application Engineering

Figure 8: Domain engineering process [WL99]

5 Domain engineering

Domain engineering studies how the products of the same family share the common basis and how they differ from each other [ADH+00]. In addition, domain engineering develops and acquires the core assets of the product line [BFG+00].

Domain engineering can be divided into domain analysis and domain implementation (shown in Figures 1, 4, and 8). The result of domain analysis is domain model (compare Figures 3 and 4) which also acts as a basis for domain implementation. This can be seen in Figure 8.

5.1 Domain analysis

Domain analysis provides different approaches. It can concentrate on describing what is inside the domain, what is the boundary of the domain, or what is outside the domain [Sch00]. The first case describes the items that constitute the domain, or identifies other domains that together form the actual domain (domains can have sub-domains). The second case describes the rules of inclusion and exclusion. In addition, structure and context diagrams can be produced both to describe the boundary of the domain and to show the relation of the domain to the outside.

The purpose of domain analysis is to produce a domain model. In FAST context, domain model means a specification for the application engineering environment. In more general, domain model is a precise representation for specification and implementation concepts [PA91]. It includes concepts concerning system specification, plans to map specification into code, and rationales for the specification concepts, and their relations to the implementation plans. Thus, domain model captures common knowledge about the domain and guides reusing the components of the domain.

FAST process divides domain analysis into subtasks (as has been done in Figure 4). Similarly, the artifacts produced during domain analysis (i.e. domain model) can be divided further (see Figure 3). One of these artifacts, economic model, is considered in Section 4. Commonality analysis comprises identification of both common and variable aspects among family members. (Commonality analysis is considered in more detail in Subsection 5.1.1.)

The decision model of domain model defines the decisions that an application engineer must make to specify and produce a new member of the family. Decision model also determines the order in which the decisions should be made to produce an application. These decisions can be, for example, choosing a value for a parameter described in commonality analysis.

A part of domain model is an application modeling language (AML) that can be designed either via compositional approach or via compiler approach. (Application modeling language is considered in more detail in Subsection 5.1.2.)

The purpose of domain model is to specify an application engineering environment and a process for using this environment to model and generate applications (see Figure 8). This specification acts as a basis for the environment to be developed during domain implementation.

5.1.1 Commonality analysis

One task of domain engineering is to consider the similarities and differencies between the members of the product family. This aspect of domain engineering is called commonality analysis, although it covers also considering the variabilities.

Commonality analysis is the basis for designing a family (or a domain). It identifies and makes useful the abstractions that are common to all family members [Wei98]. There are two main sources of abstraction: terminology and commonalities. Precise terms concerning product-line architecture make communication among developers easier and more accurate. As another source of abstraction, commonalities are actually assumptions that are true for all family members. Besides the commonalities, it is important to consider variabilities among family members. Variabilities provide a way to prepare for potential changes by pointing those decisions concerning family members that are likely to alter over the lifetime of the family.

The result of commonality analysis is commonality document consisting of the following sections [ADH$^+$00]:

Overview
describes the domain and its relation to other domains.

Definitions
provide a standard set of technical terms.

Commonalities
consist of a structured list of assumptions that are true for every member of the family.

Variabilities
consist of a structured list of assumptions about how family members differ.

Parameters of variation
consist of a list of parameters that refine the variabilities, adding a value range and binding time for each.

Issues
form a record of important decisions and alternatives.

Scenarios
are examples used in describing commonalities and variabilities.

Commonality analysis can be used for several purposes [Wei98]. It can be used in the further phases of the FAST process, for example, in designing a domain specific language and then in generating code and documentation from the language specification for each product. It serves as a basis for a family architecture and as reference documentation. Commonality analysis can also be exploited in reengineering the members of a product family. It can be used as a training aid for new project members. In addition, a plan for evolution of the family can be derived from commonality analysis.

5.1.2 Application modeling language

Application modeling language (AML) is a key part of domain model. It is called modeling language because the specifications written in the language should be models. They should actually be abstractions of applications. The application engineering environment enables analyzing the specifications written in AML. It also provides ways to generate code from the model, or in other words, to map the abstraction into an implementation.

FAST process has two approaches for generating family members from the AML: compilation and composition. The composition approach creates a modular design for the family and implements each module as a template. In addition, a composer is needed to generate family members by composing completed templates. The family specification determines the templates to be used for a particular family member.

Composition approach requires family design (or domain design) that is common to all family members and acts as a basis for generating family members. In addition, composition mapping between the AML and the family design is needed in generating family members.

Compilation approach requires building a compiler including a parser for the AML, a semantic analyzer for the parsed form, and a code generator. The generated code could be high-level (such as C or Java), machine code, or any other.

The decision between the approach depends on the domain. For some domains, the compilation approach is more suitable, while in others, the composition approach may prove to be more natural. In addition, the experience of domain engineers can be critical. If the domain engineers have no experience in implementing a compiler, that approach could be too complicated.

AML is designed and implemented by domain engineerings and used by application engineers. However, application engineers need not know the approach applied in the AML or other internal decisions made by domain engineers.

5.2 Domain implementation

As the second part of domain engineering, domain implementation develops or refines an environment that satisfies the domain model. This environment should also support the application engineering process. If the chosen AML approach is compositional, it is necessary to implement the family design, composition mapping and a composer. In the opposite case, using a compiler approach, it is necessary to implement analysis tools and a compiler.

Domain implementation includes providing the toolset that forms the application engineering environment. This toolset comprises both generation tools and analysis tools. Generation tools are used to generate code and documentation for applications. Analysis tools are used to analyze application models to help the application engineer validate the models. Documentation concerning both kinds of tools is also produced.

Domain implementation covers creating a library of templates. It is needed in implementing code and documentation for applications. Additional documentation is required to understand how to use the application engineering environment.

6 Application engineering

Application engineering can be performed parallel with domain engineering. Domain engineers can refine parts of the application engineering environment at the same time when application engineers use other parts of the environment to model an application.

Application engineers use the production facilities (application engineering environment) provided by domain engineers to produce applications of a family quickly. The applications should satisfy customer requirements, and thus, application engineers are connected to customers either directly or via other people such as salespeople or system engineers. Customers may be external or internal of the organization, and requirements can be established by formal contract or informal discussion.

Application engineering is an iterative process (see Figure 9). The customer identifies or refines the requirements for the application. Then the application engineer represents the requirements as an application model. Actually, she can provide a number of different models for a family member, and analyze them in different ways and refine them several times to be sure that the model corresponds to the requirements of the customer. According to the application model, the application engineer generates a deliverable set of code and documentation. The customer checks the received application. This may include testing the application and viewing the result of analyses made by the application engineer.

If the customer is not satisfied with the application, the requirements are refined and the whole process starts from the beginning. The activities presented in Figure 9 terminates when the customer accepts the application or is sufficiently dissatisfied with the process to stop participation.

Application engineering process produces artifacts presented in Figure 3. Application model is created on the basis of the application engineering environment by using AML. Application engineers use AML to specify and produce family members. The application engineering environment supports analyzing specifications written in AML. AML is also needed in generating code from the application model.

In addition to application model, application engineering process produces code for the application. The code is generated from the application model by using generation tools provided by domain engineers. The tools included in the application engineering environment can also be used in generating documenta-

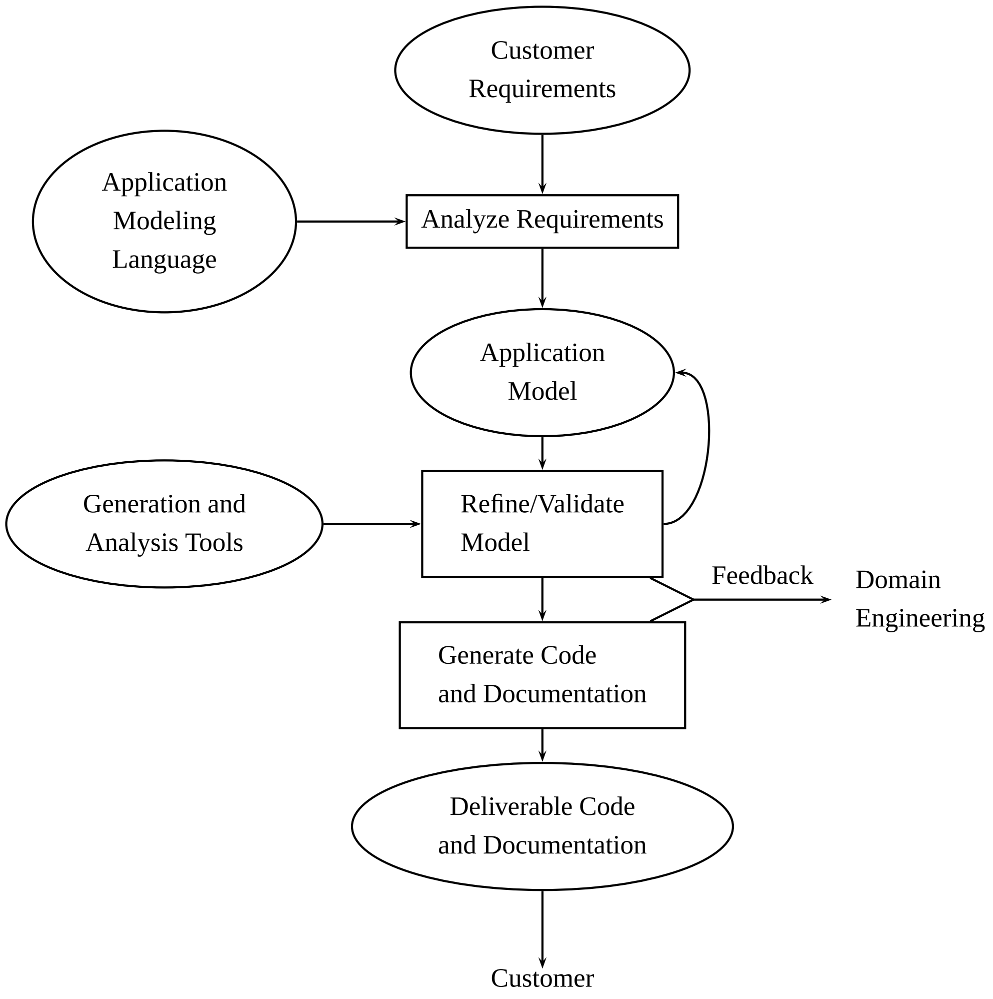

Figure 9: Application engineering process (simplified from [WL99])

tion for the customer about the application.

7 PASTA model

To describe FAST process precisely, a special model called PASTA (Process and Artifact State Transition Abstraction) can be applied [WL99]. PASTA provides a systematic way to describe software engineering process (particularly FAST process). Thus, it supports communication and iteration of the process.

Software engineering process can be considered as a sequence of decision making activities. These decisions concern requirements of the software, programs implementing the requirements, required properties such as program structure, interface, and performance. In addition, decisions about how to verify that the programs meet their requirements are needed. PASTA model guides making these decisions. It informs of what decisions software engineers can make, when they can make them, what the results mean, and how the results should be presented.

Describing FAST process means describing the three aspects: artifacts, activities, and roles. Application and domain engineers produce several artifacts such as requirements documents, code files, and organization charts. These artifacts contain the various decisions made. To produce these artifacts, engineers perform different activities, and play different roles. Describing these aspects produces a process model.

However, PASTA model or any other formal model cannot describe software development completely. For example, a total ordeding of the activities is not provided. PASTA model only shows at different points a set of activities that can proceed next. The engineers make selection among these alternative activities.

7.1 Elements of PASTA model

PASTA model consists of state transitions and their abstractions. It contains the following elements:

Role
: represents responsibility, assignment, authority, or work force. It indicates who can perform an activity.

Artifact
: represents a final or intermediate work products or the information needed to produce it. It indicates what must be produced.

Artifact state

is the condition of the artifact. Artifacts change state according to the activities addressed to them. State changes indicate the progress toward completing the artifact.

Process state

is a set of activities performed in a particular situation. The situation is defined by an entry condition that must be satisfied before the process state is entered. An exit condition specifies the effect of the process and determines transition out of the process state. The entry and exit conditions are specified in terms of artifact states. Process states can be organized into substates. They indicate what must be done to make progress, when it can be done, who can do it, and what the criteria are for completing an activity.

Operation

is an activity to be performed on one or more artifacts. Operations form the lowest level of the process state hierarchy, they are not further organized into subactivities. Operations indicate what must be done and who can do it.

Analysis

is an activity that provides information about the state of the process, its artifacts, and the roles of the participating people. Analyses indicate what progress has been made, how fast it has been made, what resources have been used, and what the quality of the artifact is.

Relation

shows relationships among process elements. In addition to built-in relations between artifacts, activities, and roles, other special relationships may be needed. For example, a completeness relation says that a design document is not complete until the corresponding requirements document is complete.

7.2 State machines

PASTA model represents decision-making activities as state machines that can execute in parallel. States correspond to activities performed by people acting in different roles. This kind of state-based model enables representing concurrency and backtracking. Although some FAST activities can be performed in parallel, this does not hold for all activities. In these cases, the order of the activities is specified with entry and exit conditions. They determine when an activity can begin and what conditions must be true when it terminates.

FAST activities (see Figure 4) correspond state machines. The whole FAST process is divided into five substate machines: qualify domain, engineer domain, engineer application, manage project, and change family. States composed of substates are called *composite states*, while *elementary states* have no substates. Elementary process states consist only of operations and analyses.

Like activities, also artifacts can be represented as state machines. Artifact states can be, for example, as follows:

Referenced
: An artifact has been referenced somewhere in the process. Thus, it is needed and must be created if it does not exist.

Defined_and_Specified
: The contents of the artifact have been defined and specified.

Reviewed
: The artifact has passed a review and has been accepted.

To identify clearly the names of states, the words of them are written in different style and connected with underscores. For example, *Defined_and_Specified* is the name of an artifact state. Table 1 shows some sample decisions, corresponding artifacts, and artifact states.

7.3 Describing PASTA model

PASTA uses tree diagrams (Figures 2, 3, and 4), forms (Table 2), and state transition diagrams (Figure 10) to describe process elements. Forms, represented as tables, define process states, artifacts, and their states. Process state forms include state entry and exit conditions, state transitions, and the operations and analyses that can be performed in the states. Transition diagrams show process state machines and artifact state machines.

Table 2 is the process state definition form for the process state called *Qualify_Domain*. Figure 10 shows the corresponding process state transition diagram with subprocess states. For example, the entry to substate *Gather_Data* is a function of the artifact *Economic_Model* when its artifact state is *Referenced*.

Decision	**Artifact**	**States**
Is the family economically viable?	Economic model	*Referenced* *Started* *Reviewed*
What are the members of the family?	Commonality analysis	*Referenced* *Standard_Termilogy_Established* *Commonalities_Established* *Variabilities_Established* *Variabilities_Parametrized* *Reviewed*
How should family members be described?	Application modeling language	*Commonality_Analysis_Reviewed* *Language_Type_Identified* *Language_Specified*
How should the software for the family be organized to take advantage of both commonality and predicted variability in family members?	Family design	*Referenced* *Defined_and_Specified* *Reviewed*
What is the implementation of a component of the family design?	Code for the component	*Referenced* *Designed* *Reviewed* *Tested*
What progress toward engineering a domain has been made?	Report of milestones achieved and resources used	*Referenced* *Delivered*

Table 1: Sample decisions, corresponding artifacts, and artifact states [WL99]

Name	*Qualify_Domain*
Synopsis	Domain qualification produces *Economic_Model* determining the economic viability for a domain. A domain is economically viable if the investment in domain engineering is more than repaid by the value obtained from it.
Main role	*Domain_Engineer*, *Domain_Manager*
Entrance Condition	state-of(*Economic_Model*)=*Referenced* or state-of(*Change_Report*)=*Domain_Change_Authorized*
Artifact list	*Economic_Model*
Information artifacts	Environment

OPERATION LIST

Name	*Gather_Data*
Synopsis	Gather the data needed to decide whether a domain exists and is worth engineering.
Name	*Analyze_Data*
Synopsis	Create an *Economic_Model* for the domain that can be used to evaluate the cost and time savings from applying domain engineering.
Name	*Reject*
Synopsis	The *Economic_Model* has been created and evaluated for the domain. It is not worth investing in the domain.
Name	*Accept*
Synopsis	Based on the evaluation of the *Economic_Model* for the domain, it is worth investing in the domain.
Exit condition	state-of(*Economic_Model*)=*Reviewed*
Informal specification	Gather data on the expected family members, current cost and time to develop the members. Characterize current process to develop family members and identify potential savings in time and cost from automation. Create an economic model that shows the difference in cost and time between current process and domain engineering. Use the model to decide whether the investment in domain engineering is worth the savings in cost and time.

Table 2: Process state definition form for *Qualify_Domain* [WL99]

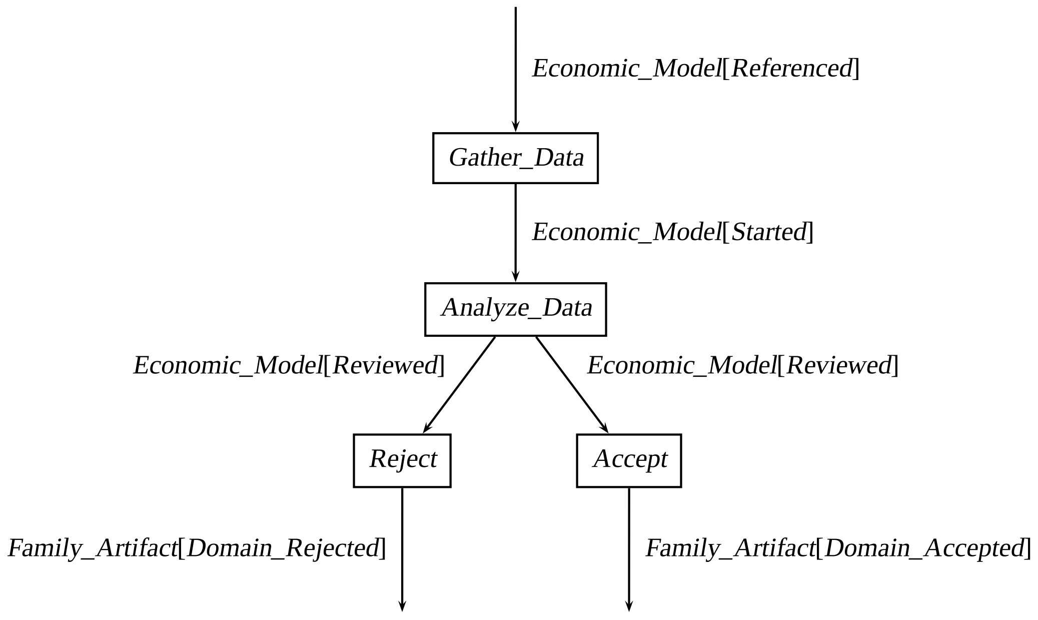

Figure 10: Process state transition diagram for *Qualify_Domain* [WL99]

7.4 Creating PASTA model

PASTA model need not be created as a single step. It is possible to create first a simple version of the artifact, activity, and role trees and supplement them later. After iterating and refining each tree, a short synopsis is created for each tree node. These synopses act as a basis for the definition forms for artifacts, process states, operations, analyses, and roles. During the creation process, the consistency between different descriptions are checked continuously.

When considering the creation of the PASTA model, necessary steps in more detail are listed below:

Decide which artifacts are to be created, modified, and used during the process.

Artifacts can be organized into subartifacts. For example, a document can be organized into sections, and code into modules. Artifacts and their subartifacts are represented as a tree hierarchy. Each artifact in the tree is described in detail using a table.

Determine the states of the artifacts and the transition among them.
A state machine diagram is constructed for each artifact.

Determine the states of the process.
States can be organized into substates. For example, a design state could consists of a modular design state and an interface design state. States and their substates are represented as a tree hierarchy. Each state in the tree is described in detail using a table and the corresponding state machine diagram. They also describe sequencing and transitions among the states, which is not shown in the tree.

Determine the operations and analyses that can be performed in each state of the process.

Each operation and analysis is described using a table.

Determine the roles of the people participating the process.
Roles are organized in a tree hierarchy. Each role in the tree is described using a table.

Determine the relationships among the artifacts.
Each relationship is described using a table.

The above steps are meant to guide process modeling. In each time, although with the same participating people, the process will most evidently be different.

PASTA provides flexibility in creating and performing the process in different ways. Each aspect (such as an artifact, activity, or role) can be described in any amount of details. When the trust in the process and the willingness to invest more effort in process modeling increases, the process can be described in more detail.

8 Mobile Terminal Example

This section shows how FAST process could be applied in a domain concerning mobile terminals. The domain in question is first introduced. After that, each phase of FAST process is considered and applied in this particular domain. Figure 1 depicts FAST process in general cases. Figure 11 (derived from Figure 1) shows FAST process in the mobile terminal domain.

8.1 Overview of mobile terminals

The chosen example is derived from [Moi01] concerning a software platform for mobile terminals. Mobile applications or wireless applications can be used in a wireless terminal like a mobile phone or a PDA (personal digital assistant). Terminals typically communicate wirelessly with other devices, for example other terminals or servers. A wireless application is usually executed partly outside the terminal. These new kinds of terminals are small but still effective. They are provided with colour displays and powerful processors. They can show moving video pictures, 3D graphics, and effective sounds.

Application programming for mobile terminals is different from conventional program development. Memory requirements are typically more strict. Displays vary in their size and shape from one terminal to another. Keyboard and button equipments differ from each other. These differences require application customization according to the properties of each device.

Mobile terminals typically have an open operating system which allow anyone (in addition to the producer of the device) to develop applications referring to the features of the operating system. Thus, the applications can be made powerful because they can directly use the properties of the underlying device. However, such applications require large modifications when porting them to other devices.

8.2 Domain qualification

Domain qualification considers how useful and profitable it is to follow product-line convention in software engineering process (see Figure 11). In the mobile example, turning into a common programming platform makes the development of different mobile applications easier. The main purpose is to make programming tasks consistent with each other such that they do not depend on the underlying device. However, simplification of the application development can be seen as a

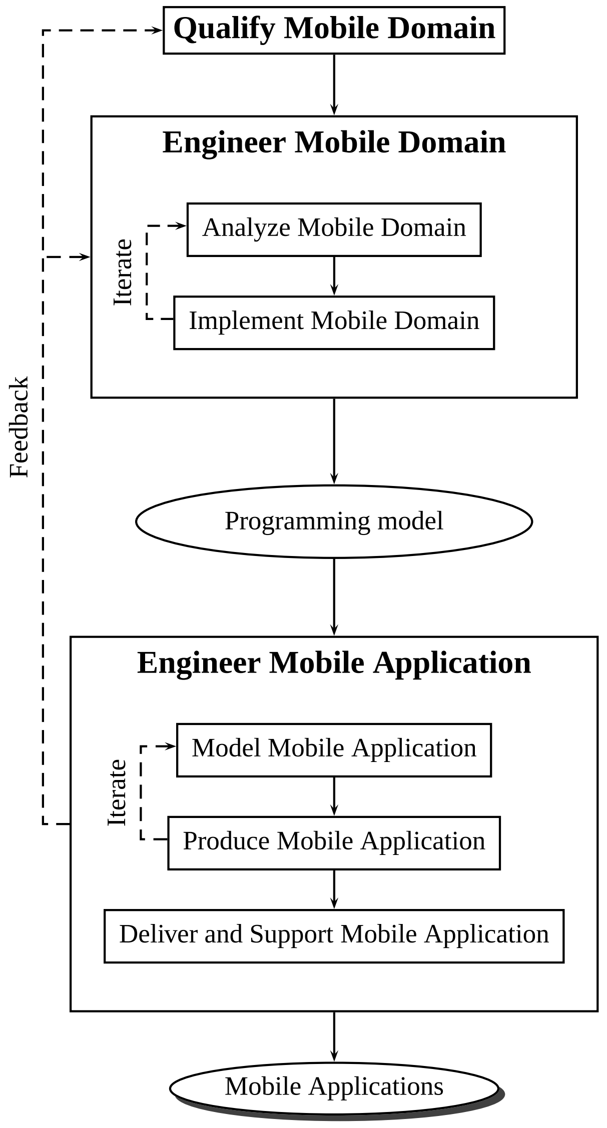

Figure 11: FAST process for mobile domain

way to reach also financial advantage.

The common programming platform tries to solve the conflict between the following aims. On one hand, it is preferred to take advantage of the special properties of each terminal. On the other hand, it is desired that the same applications without any modifications can be run in different terminals. The common programming platform provides a layer above the detailed properties of different terminals. However, it enables customization of the application for each specific terminal.

8.3 Domain engineering

Domain engineering can be divided into domain analysis and domain implementation (see Figure 11). Domain analysis comprises commonality analysis which, in the particular mobile example, covers the identification of different operating systems. In this example, supported operating systems are PocketPC, EPOC, Linux, Palm OS, and Windows. Terminals may have variant properties such as memory resources and data communication. Supported devices are, for example, PDAs and smart phones.

Besides commonality analysis, domain analysis covers family design. In the mobile example, applications are typically based on client/server architecture. Terminals act as clients using the services of servers. An application can be executed both in the terminal and in the server. The programmer of an application can decide which part of the code is run in the terminal and which part is executed by a server. This division may depend on the restrictions of different devices. For example, it is possible to give for an ineffective device more server time than for a more effective one.

The other part of domain engineering is domain implementation which provides tools and an environment for developing applications. In the mobile example, the environment is divided into following two parts. *Development library* supports application development, while *service platform* is meant for operators and service providers.

The development library has a layered architecture consisting of three layers: operating system abstraction, system services, and media services. The lowest level, operating system abstraction acts as an adapter to the services of the actual operating system. It provides operating system services such as memory management, file system, threads, and synchronization. System services of the middle level raise the abstraction level of the services by providing useful and more portable concepts such as network channel and resources. However, the applications using the services of this level need not care about different terminals supporting different network channels. Media services of the top level provide solutions for specific problems. The solutions are customized for each different terminals. Examples of these services are packaging and unpackaging video pictures.

Domain implementation provides an environment for developing both client (terminal) and server applications. The purpose of the terminal environment is to provide a common way to program different terminals. Thus, it should not be necessary to rewrite the program for different terminals. The same purpose concerns client environment, too. However, there is less variability among the operating systems than in the terminal environment. When developing a server, the restrictions such as memory requirements are less severe than in the case of clients. Both of these environments are as similar as possible to make it easy for the users to learn the environments.

In addition, domain implementation provides necessary development tools for compiling, linking, and automatic processing of application data. Tools are also needed when starting to support new devices.

8.4 Application engineering

The purpose of domain engineering is to produce an application engineering environment. The environment created for developing mobile applications is introduced in the previous subsection. Using the environment is called following a *programming model* (see Figure 11). The programming model provides a uniform environment to develop programs for different terminals. Application engineering uses the programming model to implement mobile applications.

Application engineering can be divided into modeling, producing, and delivering and supporting applications (see Figure 11). Application modelling concentrates on customer requirements. In the mobile example, the most important requirements are reliability, reactivity, and security. Reliability is important because the users prefer and require safe applications on which they can rely even in critical situations. Reliability can be achieved by accurate programming conventions and version control.

Reactivity means that the response time is as low as possible. Response time is low enough if the user cannot notice any delay. Time-consuming services can be provided as asynchronized services. In such services, requests can be left to wait for service and immediately continue normal processing. The reply to the request may take for a while and it will be handled as soon as it comes. However, it is essential that the waiting time is not wasted idly.

Security is important because an application in mobile terminals can directly use the device instructions. Thus, it would also be possible to remove or read the user's personal data or send it to other devices. To be sure about the security of the system, the user should be able to control the programs to be loaded in her terminal.

Producing an application covers the basic programming tasks implemented by following the programming model. C++ has been chosen as the implementation language. Special C++ classes are used to provide a common interface for different terminals. These interface classes are implemented as abstract classes and used via inheritance and dynamic binding. They hide the differences between terminals. For the hiding purpose, macros and C++ templates can be used, too. Some of the compilers of the devices do not support exception handling. Thus, for handling exceptions, a specific way imitating C++ exception handling has been implemented.

Delivering and supporting a mobile application has its specific characteristics. Delivering may require different arrangements such as contracts with phone operators that offer delivering services. In addition, network connections are needed. These characteristics have been considered in the service platform to make the mobile applications easily available for customers.

The described system with its development library and programming model is currently in an initial phase. However, as an application example, a multi-player game has been produced to be able to demonstrate the system. The game has proved that the system could be successful.

8.5 Conclusions

The described example is not originally processed via FAST. Actually, the example is afterwards tried to accommodate to FAST, or tried to outline how FAST could be applied in this particular situation. However, taking into account the deficiences of the situation, FAST fits to the chosen example surprisingly well. The subprocesses are rather easy to find even afterwards. However, FAST has not been applied in a detailed way. The purpose was to identify only the main phases of FAST process.

The example does not provide any explicit application modeling language. However, the programming model can be considered as an AML. Thus, in this particular example, AML is a restricted group of C++ programming concepts. The restriction is formed according to the restriction of the different terminals.

This example concentrates only on the basic FAST process. FAST artifacts and roles have not been considered. They are important during FAST, but not so essential afterwards. PASTA model has not been taken into account, neither. Because it was not possible to observe the process during processing it, it is difficult to follow PASTA model. PASTA model provides advice during the process. It is not meaningful to find the steps and selections afterwards.

9 Conclusions

Dividing the product-line engineering process into domain engineering and application engineering is not unique to FAST. Instead, there are several process models using similar division. However, some of the methods have first concentrated on domain engineering, and they are afterwards extended with application engineering.

FODA (Feature-Oriented Domain Analysis) [KCH$^+$90] considers the features of similar applications in a domain. Features are capabilities of the applications considered from the end-user's point of view. Features cover both common and variable aspects among related systems. They are divided into mandatory (or common), alternative, and optional features. FODA includes different analyses such as requirements analysis and feature analysis. It produces a domain model covering the differences between related applications. FODA is currently a part of the model-based approach of domain engineering [Sof00]. This approach covers both domain engineering and application engineering. The domain engineering part consists of domain analysis (FODA), domain design, and domain implementation. In addition, FODA is further extended into FORM (Feature-Oriented Reuse Method) [KKLL99] to include also software design and implementation phases. FORM covers both analysis of domain features and using these features to develop reusable domain artifacts.

ODM (Organization Domain Modeling) [SCK$^+$96] mainly concentrates on the domain engineering of legacy systems. However, it can be applied to the requirements for new systems. ODM is tailorable and configurable, and it can be integrated with other software engineering technologies. It combines different artifacts such as requirements, design, code, and processes from several legacy systems into reusable common assets. ODM is supported by DAGAR (Domain Architecture-based Generation for Ada Reuse) [KS96]. DAGAR process does not cover domain modeling. Thus, it applies ODM or other methods for this purpose. Instead, DAGAR process includes activities both for domain engineering and application engineering.

RSEB (Reuse-Driven Software Engineering Business) [JGJ97] is a systematic, model-driven reuse method. It composes sets of related applications from sets of reusable components. RSEB uses UML (Unified Modeling Language) [RJB99] to specify application systems, reusable component systems, and layered architectures. Variabilities between systems are expressed with variation points and attached variants. FeatuRSEB (Featured RSEB) [GFd98] connects features (from FODA) with RSEB. Actually, FODA and RSEB have much in common.

Both of them are model-driven methods providing several models corresponding the different view points of the domain. Thus, they are compatible with each other.

PuLSE (Product Line Software Engineering) [BDF$^+$99] divides product line life cycle into three parts. In the initialization phase, PuLSE is customized to fit the particular application. Adaptation is affected by the nature of the domain, the project structure, the organizational context, and the reuse aims. In the second phase, product-line infrastructure is constructed. This step includes scoping, modeling, and architecting the product line. In the third phase, the product-line infrastructure is used to create individual products. This concerns instantiating the product-line model and architecture. Each of these phases is associated with product-line infrastructure evolution. Each phase should consider changing requirements and changing concepts within the domain. PuLSE has several components. PuLSE-DSSA (PuLSE - Domain-Specific Software Architecture) [ABFG00], for example, develops a domain specific architecture based on the product-line model. As other examples, PuLSE-Eco concentrates on economic scoping, and PuLSE-EM on evolution and management.

As shown, there exist several process models rather similar to FAST. According to these process models, the process division into domain engineering and application engineering has proved to be a useful convention. Because of the similarities between the models, accomodating the mobile terminal example (presented in Section 8) to other process models would probably be very much the same as adapting it to FAST.

References

[ABFG00] Michalis Anastasopoulos, Joachim Bayer, Oliver Flege, and Christina Gacek. A process for product line architecture creation and evaluation, PuLSE-DSSA–version 2.0. Technical Report IESE-038.00/E, Fraunhofer Institut Experimentelles Software Engineering, June 2000.

[ADD$^+$00] Mark Ardis, Peter Dudak, Liz Dor, Wen-jenq Leu, Lloyd Nakatani, Bob Olsen, and Paul Pontrelli. Domain engineered configuration control. In Patric Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 479–493. Kluwer Academic Publishers, 2000.

[ADH$^+$00] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software — Practice and Experience*, 30(7):825–847, 2000.

[Aks96] Mehmet Aksit. Separation and composition of concerns in the object-oriented model. *ACM Computing Surveys*, 28(4es), 1996.

[BDF$^+$99] Joachim Bayer, Jean-Marc DeBaud, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, and Tanya Widen. PuLSE: A methodology to develop software product lines. In *Symposium on Software Reusability (SSR'99)*, pages 122–131, May 1999.

[BFG$^+$00] John Bergey, Matt Fisher, Brian Gallagher, Lawrence Jones, and Linda Northrop. Basic concepts of product line practice for the DoD. Technical Note CMU/SEI-2000-TN-001, Software Engineering Institute, Carnegie-Mellon University, 2000.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Fifth International Conference on Software Reuse (ICSR'98)*, pages 76–85, June 1998.

[Gri00] Martin L. Griss. Implementing product-line features by composing aspects. In Patric Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 271–288. Kluwer Academic Publishers, 2000.

[HL95] Walter L. Hürsch and Christina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, February 1995.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[KCH$^+$90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[KIL$^+$96] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es), 1996.

[KKLL99] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, and Kwandoo Lee. Feature-oriented engineering of PBX software for adaptability and reuseability. *Software — Practice and Experience*, 29(10):875–896, 1999.

[KS96] Carol Diane Klingler and James Solderitsch. DAGAR: A process for domain architecture definition and asset implementation. In *ACM TriAda'96 Conference*, December 1996. Available from: http://source.asset.com/stars/darpa/Papers/ArchPapers.html.

[Moi01] Jussi Moisio. Software platform for mobile terminals (in Finnish). Master's thesis, Tampere University of Technology, Software Systems Laboratory, May 2001.

[Myl02] Tommi Myllymäki. Variability management in software product lines. Technical Report 30, Institute of Software Systems, Tampere University of Technology, January 2002.

[PA91] Rubén Prieto-Díaz and Guillermo Arango. *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, 1991.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[Sch00] Klaus Schmid. Scoping software product lines. In Patrick Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 513–532. Kluwer Academic Publisher, 2000.

[SCK^{+}96] Mark Simos, Dick Creps, Carol Klingler, Larry Levine, and Dean Allemang. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, June 1996.

[Sof00] Software Engineering Institute, Carnegie-Mellon University. *Domain Engineering: A Model-Based Approach*, January 2000. Available from: http://www.sei.cmu.edu/domain-engineering/.

[Wei98] David M. Weiss. Commonality analysis: A systematic process for defining families. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families*, volume 1429 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1998.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.