An Integrated Model for Requirements Structuring and Architecture Design

Juha Savolainen, Tuomo Vehkomäki

Nokia Research Center
{Juha.Savolainen | Tuomo.Vehkomäki}@nokia.com

Mike Mannion

School of Computing and Mathematical Sciences,
Glasgow Caledonian University
{M.A.G.Mannion@gcal.ac.uk.}

Abstract

Requirements guide the development of a software intensive system, whereas the software architecture largely dictates the achievable properties of the system. This interplay of requirements and architectures has been largely accepted by the researchers and practitioners alike. Despite the common understanding of the general approach, the exact guidelines on how to develop systems in practice are missing. Features are often used to map customer requirements into product properties. However, our experience shows that features are often misused. Their real role is not understood or they are used for premature design and solution specification purposes. A number of different methods for either analysing and modelling requirements or for designing architectures exists, but the combination and customisation of these methods is left for the practitioners. The transition from problem definition to architecture is mainly dependent on the creativity and problem understanding of the chief architect. In this paper, we argue how 4 existing models, problem domain models, context diagrams, feature models, and architectural descriptions can be used together to make the transition process more transparent.

Keywords:

Requirements, features, software architecture.

1. Introduction

The goal of software product development is to create systems that fulfil the needs of various stakeholders. The needs are usually expressed as requirements which then drive the construction of a software architecture that will satisfy those needs. However the transition from requirements definition to architecture is mainly dependent on the creativity and problem understanding of a chief architect and is often not transparent. One reason for this is the confusion in defining the boundaries of the problem and solution domains and in understanding the relationships between requirements, context specifications, features and architecture.

To address this issue, we argue that it is possible to integrate different modelling techniques by considering them as different viewpoints on the same systems development problem. The techniques we selected are: problem frames for problem domain definition (Jackson, 2001), context diagrams for defining the interface between a system and its environment in the problem domain (Yourdon, 1988), feature models that provide a bridge between requirements and their architectural realisation and architectural component descriptions used internally. In

our experience each of these approaches addresses a unique set of concerns and provides useful views of a system that assists the practical development process.

We have selected the problem frames because it has been well documented and it provides a complete method for problem domain modelling. The method is easy to integrate with common context modelling methods. Other methods such as conceptual modelling or object-oriented analysis of the problem domain do not, in our opinion, provide as stable basis to ground the context model to the problem space.

The Yourdon method for context modelling has been chosen because it explicitly captures interaction between the system and its environment. This allows immediate check the consistency of the problem domain model and set the scope of the system. Use case models could be also used for this purpose, but understanding the actual information exchange requires inspecting the specification of each of the use cases. We find the data and control flow diagrams easier to understand at the high level of abstraction.

This paper is organised as follows. First we provide some definitions of requirements, features, context diagrams and architecture. Then we describe a viewpoint model for integrating the different modelling techniques. Then we show how the model can be used with a detailed worked example. Finally we discuss some outstanding issues.

2. Requirements, Features, Architecture

An IEEE standard (1990) defines the term requirement as:

(A) A condition or capability needed by a user to solve a problem or achieve an objective.

(B) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

(C) A documented representation of a condition or capability as in definition (A) or (B).

Underpinning this definition is the notion that requirements describe a desired state of the world after a system has been constructed to interact with the world. The relevant part of the world is called a system's environment. The environment can affect the behaviour of a system and the behaviour of a system can affect its environment. Before modelling the environment of a system, a good designer first concentrates on understanding the problem to be solved. A common mistake during software development is to focus on solution formation before the problem is well understood and analysed. Matters are not helped by commercial pressures that affect project schedules e.g. design and implementation must begin as soon as possible.

Many software development process models assume that stakeholders will describe their product requirements using only problem domain entities. This rarely happens in the practice and more often stakeholders explain their requirements in terms they understand and this may include using solution domain entities. For example, in the domain of mobile phones a user may identify a requirement for " a sliding cover for the mobile phone". Arguably, this requirement is a solution to the underlying problem which may be that they want to protect the phone against pressing keys accidentally or that they want to improve the aesthetic appeal of the phone. In this paper we refer to requirements that are described in terms of the properties of a product as *features* of that product.

An IEEE standard (1990) defines the term feature as:

(A) A distinguishing characteristic of a software item (for example, performance, portability, or functionality)

(B) A software characteristic specified or implied by requirements documentation (for example, functionality, performance, attributes, or design constraints).

Clearly it is highly important to guarantee that the features described by each stakeholder do actually satisfy their real needs. Features are important because they allow a customer to more readily express their needs. In the development of a product line they are often used to distinguish between products and establish product configurations. The main difference between the features and the requirements is that *features express requirements in terms of the solution*. This makes features an ideal bridge between the problem domain and the solution domain. Features are typically implemented by one or more architectural components.

The role of software architecture is to distribute responsibilities to components and to co-ordinate their behaviour so that they provide the needed features. Features can be assigned to one architectural element or as the common responsibility of a set of elements. Typically, large features express the basic behaviour of a system and therefore affect most of the main architectural elements of the system. The mapping of features to components and other architectural structures is crucial in order to guarantee the satisfaction of the main requirements.

3. System Development Models

System development models are used to define the problem to be addressed, to describe a solution that satisfies the problem and to ease the transition from problem definition to solution description. The choice of models used on a project in a development organisation can depend on the type of product, on how well the model types integrate with each other, and on the level of expertise of the staff in the development organisation in understanding and using these models.

One of the main goals of the initial problem domain analysis is to find subproblems that can be managed, at least partly, independently. Doing this effectively requires comparing and contrasting the problem to some known problem types. Having an existing problem frame (Michael Jackson (1995, 2001)) constrains the problem by fitting it into a known problem type. This allows using known methods for dividing the problem and analysing its properties.

The problem domain model shows all relevant domains that either directly interact with the system under development or affect the system by imposing or influencing the requirements that the system should fulfil. Everything that is relevant to the requirements must appear in some part of the problem domain model.

We have experimented on using frame diagrams to describe real products. The most common problem encountered is how to restrict the frame diagrams such that they only show the problem domain concepts. This is especially difficult in the projects dealing with large amounts of legacy assets. We feel that even then it is helpful to discuss and analyse the problem without using explicit solution techniques. This allows evaluating the chosen solutions and predetermined constraints for that problem.

Context diagrams are helpful as they provide an explicit boundary specification. They show dataflows and those domains that directly interact with the system. These domains are the adjacent domains of the system. The space that is outside the system and which is inhabited by the adjacent domains is called the system's environment. However, the context diagram with its dataflow representation shows only data (and control) exchange over the system boundary. It does not show the responsibilities that the entities have regarding these

interactions. For a system consisting of many different products, it is essential to agree a common understanding of the role that each of the products has within the system. This observation holds for both functional requirements as well as for any quality characteristics.

The main difference between a problem domain model and a context diagram is that the problem domain model shows all the domains that have any effect on the problem that the system tries to solve, whereas the context diagram shows only those entities that have a direct interaction with the system. The problem domain model is not limited to the parts of the environment that are directly interacting with the system. Also the use of these models is different. The problem domain model is useful for organising and analysing requirements whereas the context model shows the scope of the system explicitly and identifies the interfaces for specification purposes. The problem domain models such as frame diagrams focus on the shared phenomena between two domains as the main type of interaction. Context diagrams typically describe actual data flows but on a rather high conceptual level. This means that actual interaction or messages are not shown – only the type of the interaction and the main flow of information are displayed.

Feature modelling has been an integral part of product line research. Most methods (Cohen, Standley, Peterson and Krut, 1992) (Lee, Kang, Chae and Choi, 2000) (Czarnecki and Eisenecker, 2000) provide mechanisms for representing the commonality and variability in the product line. However, often feature modelling methods do not provide sufficient techniques to model complex dependencies between features. Our previous work has focused on the dependency modelling among features (Ferber, Haag and Savolainen, 2002). The resulting model shares the key concepts of the previous contributions, such as the *part of* -hierarchy among features, variability assumptions, and feature dependency modelling.

Architectural design is responsible for converting requirements and constraints as defined from outside of the system boundary into distribution and refinement of those requirements for logical components. The requirements and finally satisfied by making design decisions that shape the architecture. Design decisions create the high level structure of the software by adding new requirements and constraints and distributing the existing ones (Savolainen and Kuusela, 2002). The logical components represent the initial structuring of the software into separate entities, which allows specifying and designing at least partly independently. The final properties of the system are determined by the design of those elements as well as the collaborations among the components.

4. Viewpoint Approach For Model Definition

A viewpoint defines a way of looking at a system (IEEE, 2000). A viewpoint can be described by a metamodel that expresses the type of entities and relationships that are of concern from the specific perspective. A view, on the other hand, instantiates a viewpoint and conforms to its rules. A view uses types, which are defined in the corresponding viewpoint, and describes concrete architectural entities by instantiating new objects as parts of the view. A viewpoint defines types and a view instantiates those types to describe the structure of the software – similar to the distinction between classes and objects in object-oriented design.

Dividing a difficult development problem into a set of viewpoints helps to manage the problem's complexity. When viewpoints are instantiated to views, the challenge is to integrate the views to ensure completeness and consistency in problem understanding. One way to integrate views is to consider what each of the viewpoints provides and requires from the other viewpoints (Hillard, 1999). Viewpoint responsibility is defined by the elements the viewpoint provides. No other viewpoint can declare elements of the same kind.

The IEEE recommended practice (IEEE, 2000) defines the basic set of information that is needed for each viewpoint. Each viewpoint shall have:

- a viewpoint name;
- a list of stakeholders to be addressed by the viewpoint;
- a set of concerns to be addressed by the viewpoint;
- the language, modelling techniques, or analytical methods to be used in constructing a view based upon the viewpoint;
- the source e.g. for a library viewpoint the source could include author, date, or reference to other documents.

In the following subsections we consider the 4 different modelling formalisms as separate viewpoints to cover the problem-solution space. The fact that some of the viewpoints have overlapping information does not render them unfeasible. In fact, it improves their practical usage as the duplicated information provides a tool to verify model consistency and helps to assess the suitability of the chosen concepts from multiple perspectives.

4.1 Problem Domain Viewpoint (Using Frame Diagrams)

Purpose

Problem domain concept viewpoints form a common vision on the main concepts in the application domain. They provide a shared precise vocabulary of relationships among domain concepts.

Stakeholders

Product marketing

Product management

Requirements engineers

Architects

Concerns

Identification of the main concepts

Unification of vocabulary

Understanding the problem

Communication and sharing of domain knowledge

Elements and relationships

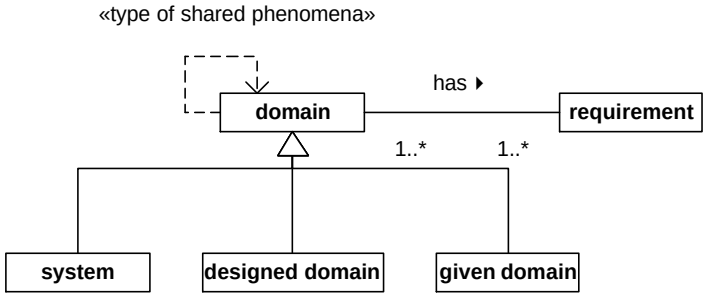

Figure 1. Metamodel Of The Problem Domain Viewpoint

Figure 1 shows that a problem has many requirements. The domains of interest can be divided into three different types. The system[1] under design is a unique domain - that is there can be only one such domain in the problem domain view. The other domains can be either designed domain or given domains. A given domain is a domain over which we have no control. All it's properties shall be taken as given – we cannot change any of those, the only option is to adapt the design of the system to take the characteristics of the given domain into consideration. The third domain type is the designed domain.

Provided elements

Problem domains

Shared phenomena

Main requirement groups

Required elements

Requirements

4.2 Context viewpoint

Purpose

The purpose of a context diagram is to provide a definition of the system scope by describing systems and/or users that interact with the system and the definition of the data and control flows through the identified interfaces.

Stakeholders

Management

Requirements engineers

Software architects

Concerns

Scope definition

Environment - system interaction

Elements and relationships

[1] We use the term *system* to separate the product that we are building from the other domains. The reader should note that Jackson uses the term *machine* for the same purpose.

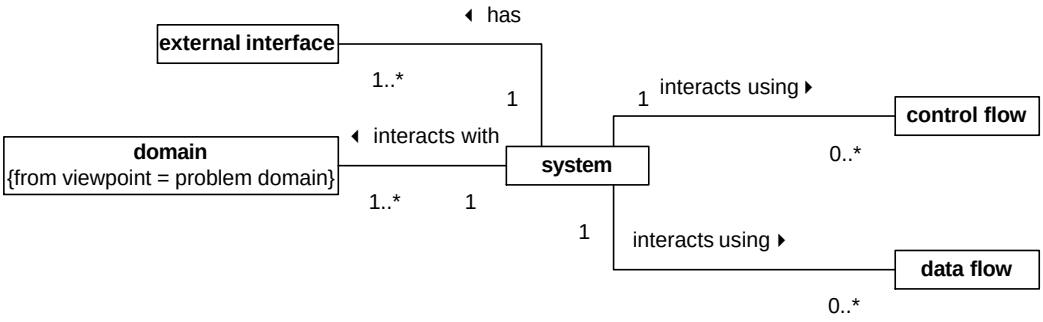

Figure 2. Metamodel Of The Context Viewpoint

Figure 2 shows that a system will interact with its environment through several external interfaces using control flows and dataflows. These interfaces are towards some of the domains of the problem domain viewpoint. The interaction can happen only with the domains that are adjacent in the problem domain model and that are shown in the context model.

Provided elements

Logical external interfaces

External data flows

External control flows

Required elements

Surrounding domains (from the problem domain viewpoint)

Shared phenomena - between the surrounding domains and the system (from the problem domain viewpoint)

4.3 Feature viewpoint

Purpose

The feature viewpoint describes the main features of a system. It shows the dependencies and bridges the requirements and the architectural specification.

Stakeholders

Product marketing

Product management

Requirements engineers

Architects

Concerns

Identification of the main features

Identification of the minimal configuration (in the product line context)

Elements and relationships

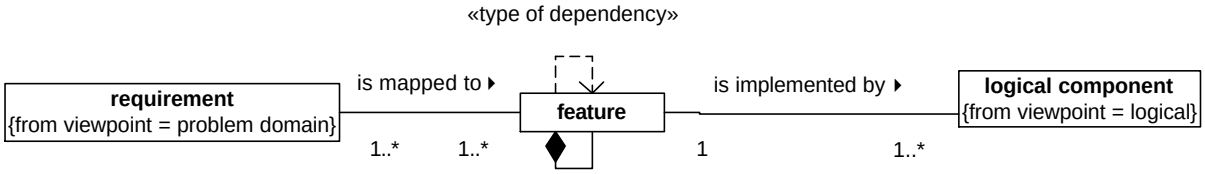

Figure 3. Metamodel Of The Feature Viewpoint

Figure 3 shows that requirements identified in the problem domain viewpoint, and mapped to features, can be implemented by architectural components that are identified in the logical viewpoint model. The feature dependency modelling supports deriving valid configurations for product lines. The dependency models can also be used to describe the minimal core functionality that a set of products have. However, the dependency modelling issues are out of scope of this paper.

Provided elements

Features

Feature dependencies

Variability assumptions (note that product line issues are out of scope of this paper)

Required elements

Main requirement groups/sources (domains from the problem domain viewpoint)

Logical components (from the logical viewpoint)

Requirements (from the problem domain viewpoint)

4.4 Logical viewpoint

Purpose

The logical viewpoint defines the system's internal structure and interactions among its parts.

Stakeholders

Subsystem architects

Software designers

Concerns

Identification of dependencies between components

Refinement of functionality

Elements and relationships

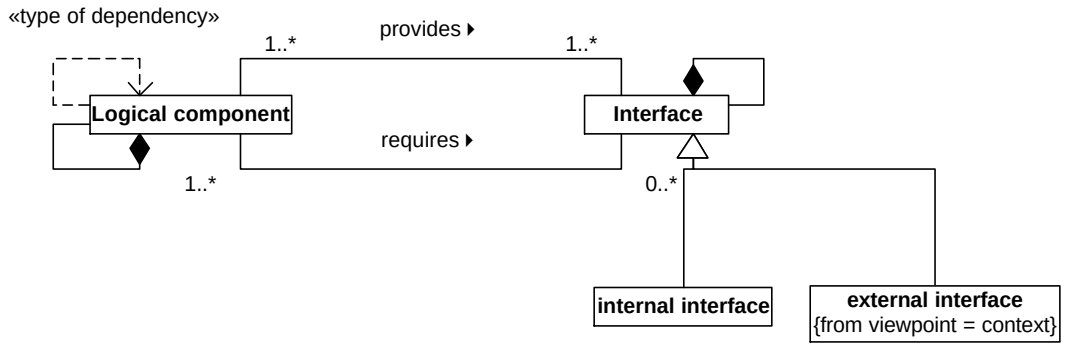

Figure 4. Metamodel for the logical viewpoint

Figure 4 shows that the architecture consists of many logical components, each of which can have internal and external interfaces, the latter being presented in the context diagram viewpoint. The logical viewpoint also defines the relationships among logical components. One of the most typical relationships between two logical components is the <<uses>> dependency.

Provided elements

Logical components

Internal interfaces

Dependencies between logical components

Required elements

External interfaces (from the context viewpoint)

5. Worked Example

In the following worked example we demonstrate our method using a limited, imaginary product.[2] Consider a mobile phone product that has a typical portfolio of features. Almost all current mobile terminals support short message services (SMS). The SMS service provides user to user communication with short text messages that can be entered with the keyboard of the mobile phone.

Figure 5 shows a simplified problem domain model for the SMS messaging. In order to save space and reduce the complexity of the diagrams, we do not show the definition of the shared phenomena between the domains and Table 1 shows only some of the requirements. We use the notation developed for frame diagrams by Jackson (2001).

[2] Please note that all information on the mobile phone products is based on the public knowledge of the mobile terminal domain. The information presented here is not a basis of actual products from any supplier. This example is only given to demonstrate our viewpoint model and should not be used for other purposes. The analysis is made by the authors and does not represent the opinion of our employers.

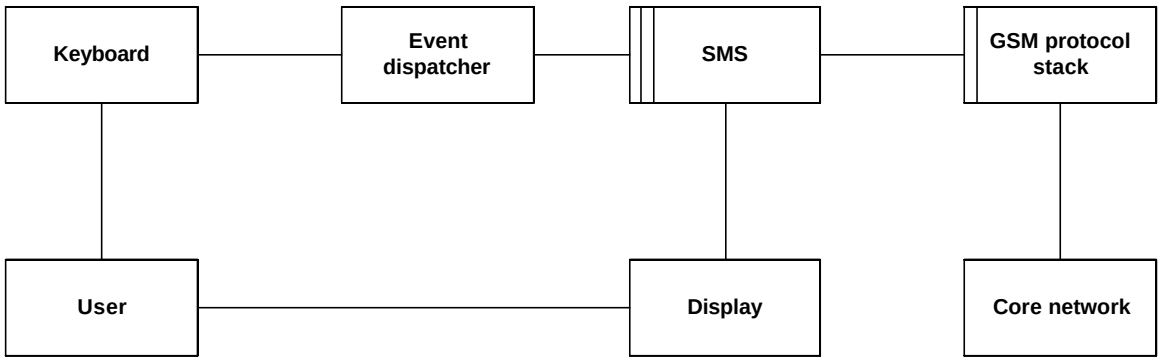

Figure 5. Problem Domain Model

A user interacts with the system by pressing the keys on the keyboard. The keyboard activates the event dispatcher that translates a key press *shared phenomena* into a set of computer events and provides basic error control and buffering. The SMS domain (the system under design) then creates the SMS messages and sends them through the GSM protocol stack that decodes the messages and deliverers them to the network. The GSM protocol stack also takes care of receiving SMS messages.

Requirement	Description
Send SMS	Edit a text message (SMS), select a recipient contact from the phonebook and sent the message to the recipient.
Send business card	Select a contact from the phonebook, select a recipient contact from the phonebook and send the details of the first contact as a text message to the recipient.
Receive SMS	Receive a text message, store it into the message database and notify the user about the new message.

Table 1: Mobile Phone Requirements

Incoming and edited text messages are broadcast to the display that shows the current message to the user of the mobile terminal. Based on the displayed message the user can take an appropriate action and send or modify the message by interacting with the keyboard. In a real mobile phone a lot of other features are controlled and monitored using the user interface software, but these features are beyond the scope of this example. However many difficult problems arise from the fact that different features use shared resources such as the display of the mobile phone. For more information consult the work by Lorentsen, Tuovinen, and Xu (2001).

Figure 6 describes a *partial* context model. The domains are drawn from the problem domain model viewpoint (Figure 5).

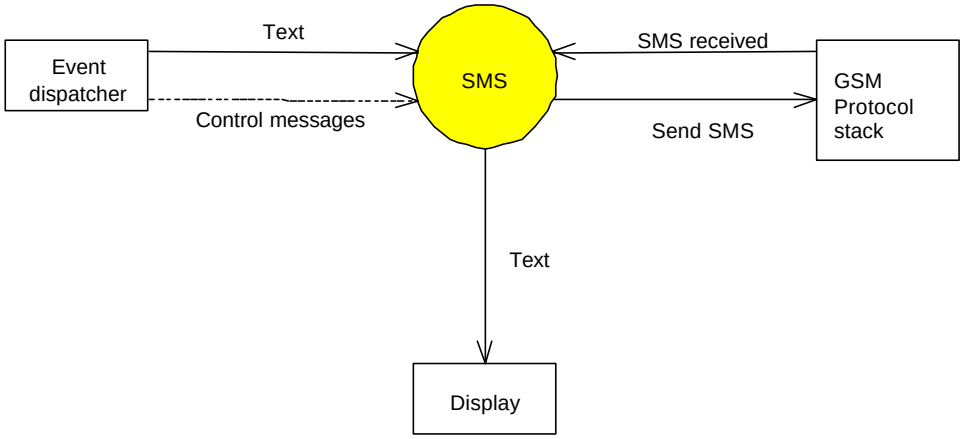

Figure 6. Context Model Diagram

The context diagram elaborates the definition of the problem domain model. It shows the actual data and control flows with which the system interacts with its environment. The context model gives an improved understanding about the interfaces the system shall implement in order to provide services and wanted functionality.

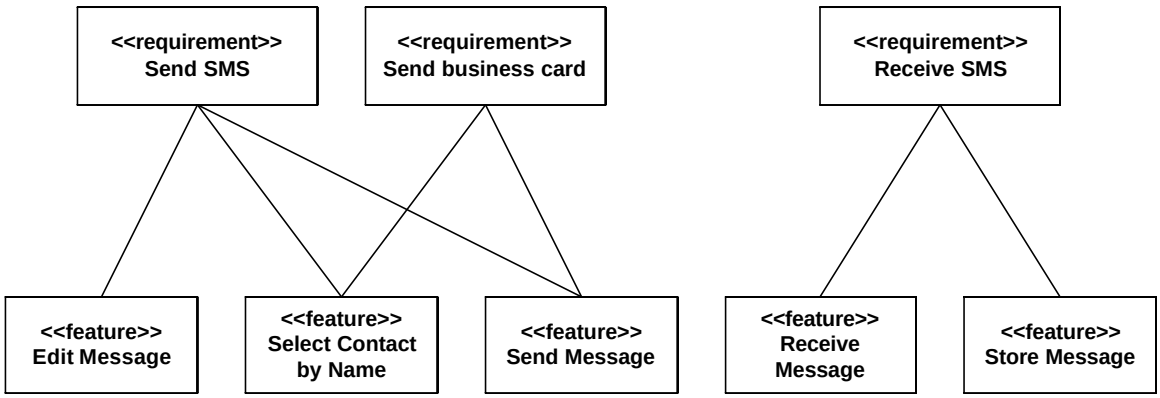

Figure 7. Feature Model

Figure 7 shows a set of sample features (Table 2) that have been identified to fulfil the requirements from the problem domain model. The lines between the features and requirement imply how the requirements can be mapped to the features of the mobile terminal.

Feature	**Description**
Edit message	Interact with the user to compose a text message.
Select contact by name	Fetch a contact from the database using the name as a search key.
Send message	Send a complete text message to a recipient via the GSM protocol stack
Receive message	Receive a message from GSM protocol stack and notify the user.
Store message	Store a message persistently into the database.

Table 2: Mobile Phone Features

Figure 8 describes how the features are realised by multiple logical components. We also introduce a grouping entity, *feature group*, with a related stereotype. As demonstrated in the figure, some of the features are realized by only one logical component. A good example of this is the feature for *message editing*. This feature is implemented solely by SMS client logical component. No others logical components *within* the SMS messaging application participate to implement this features. On the other hand, to realize the *receive SMS* feature a collaboration between two logical components is needed. As already discussed during the viewpoint definition, the dependencies among features are not shown since the product line configuration issues are out of scope of this paper.

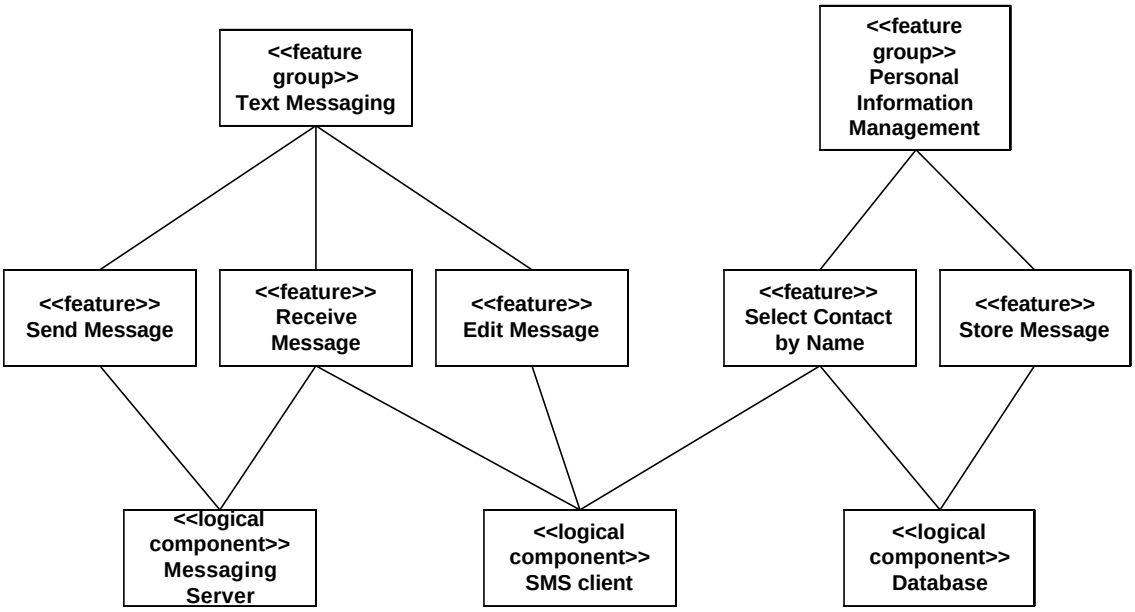

Figure 8. The Mapping of the Features to the Logical Components

Figure 9 shows a simple logical view for our SMS application. The application has three logical components (Table 3). The message client is mainly responsible for providing the user with the functionality to edit, read, and modify the SMS messages. The database component manages the persistency of stored SMS messages. SMS messages can be stored for later viewing, if enough memory is available. The database can also store draft copies of the current message whilst the message is being edited.

The mapping of the features to the logical components is based on known software design methods. The goal of the design is to create components that provide a cohesive set of responsibilities through their interfaces (Wirfs-Brock, Wilkerson and Wiener, 1990). The logical component may also serve as an information-hiding unit. The Messaging Server component hides the communication protocols that are used when interacting with the network. The SMS client manages the user communication and the Database component provides a control over the shared resource. This division helps achieving even distribution of responsibilities among the logical components.

Software patterns may also be used to bridge the gap between the solution and the problem domain. Patterns provide tested solutions to commonly encountered problems. They are complementary to our approach. Patterns can be used to structure the system into subsystem

and for developing class structure of the subsystems. Especially, patterns help achieving the identified quality aspects of the system. But during real-life development they provide solutions for a very small number of problems – mapping the requirements to a few places in the architecture.

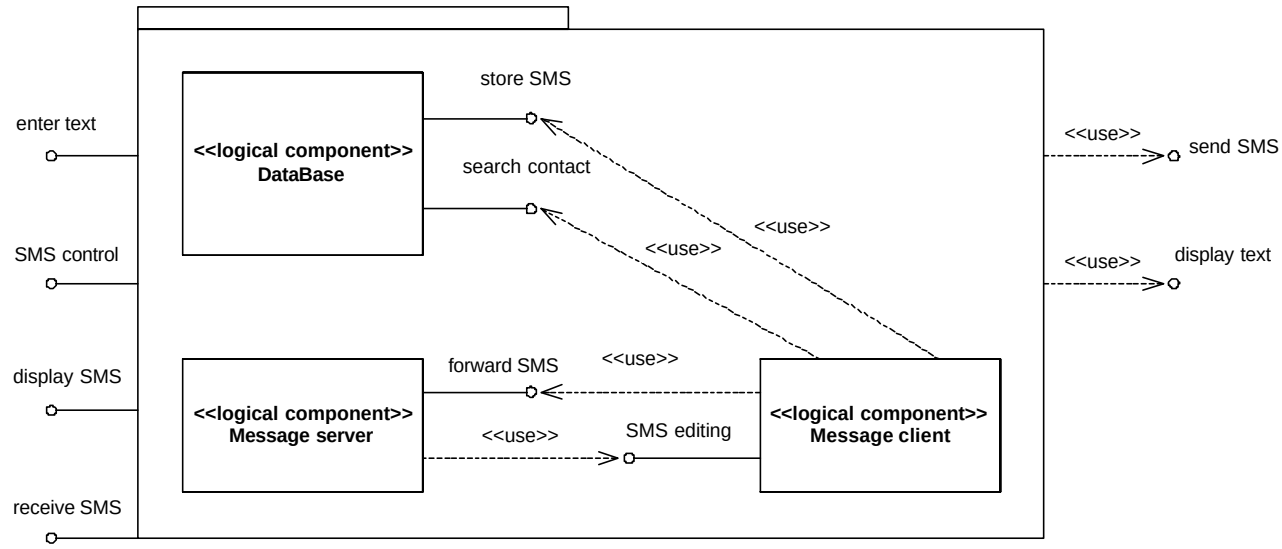

Figure 9. The Logical Architectural View of The SMS Messaging

In the logical model, we have used two different ways to represent dependencies between the logical components and the system interfaces. Within the SMS application, the collaboration of the different elements is shown using provided interfaces and dependency relations from the components that use those interfaces. Since only the SMS application is shown in the figure 9, it is not appropriate to show how the SMS application component depends on other components. The component should not depend on the actual components but rather the interfaces that the component needs to provide its services. To represent this information for the integration purposes, we show the required interfaces by dependency relation to the interface that this component requires (*send SMS* and *display text*). This notation is typically used to show the outside boundary of a component when other components are not show in the current scope of the model.

The table 3 provides a brief description of the main responsibilities of each of the logical components that are part of the SMS messaging application.

Logical Component	**Description**
Messaging server	Messaging server communicates with the GSM protocol stack to send and receive text messages. The messaging server is capable of notifying a messaging client about received messages.
SMS client	SMS client is responsible of notifying the user about received text messages and allows the user display and edit text messages.
Database	The database offers a persistent storage of contacts and text messages and provides search services to find a contact using the name as a search key.

Table 3: Mobile Phone Logical Components

6. Discussion

The four different modeling approaches were designed to articulate different stakeholder needs at different stages of the transition process from requirements to architecture specification. The viewpoint approach is used to integrate the models and make the transition process more transparent. Viewpoints also allow different stakeholders to remain focused on one part of the transition process at a time.

The approach gives prominence to feature modeling as a bridge between requirements and their realization in architectural design. This is important because features are a natural way for customers and users to express their needs.

The approach is based on intensive use of different models that share entities. This can easily lead to inconsistencies between different views. It will be essential to have computer-aided support for creating and browsing the viewpoint models and to ensure consistency between them.

One of the main benefits of the viewpoint model is that is does not strongly force to adapt a certain development process or method. The set of viewpoints that are used for a particular development project can be tailored based on known major concerns. The information on the required entities allows us to select a set that has all the needed concepts. If selected viewpoints require entities that are not introduced by other selected viewpoints then additional viewpoints shall be added based on those dependencies.

Even if can be argued that the viewpoint approach supports multiple methods and process, it is clear that our selection of viewpoints and their entities imply certain practises. The four viewpoints introduced in this paper are in practise often used as follows.

First a problem domain model is created or an existing model is adapted for the new situation. Then the main features of the system are collected and structured to form the initial feature hierarchy. Most products have existing feature lists that form the basis of that work. When an improved version of a system is created it often supports most of the features of the previous system. Simultaneously, the requirements for the system are collected and structured according to the problem domain model. The requirements are then mapped to the feature models and the resulting structure is analysed for possible gaps and overlapping requirements. An important step during the process is to identify architecturally significant requirements and to which features these are attached.

After creating the initial models for the first three viewpoints, the first skeleton architecture is drafted. This is mainly based on the most important features and the identification of the architecturally significant requirements. The main objective for the initial architecture is to demonstrate that the main features can be implemented while satisfying the key requirements. After this each of the models are updated and elaborated when new features and requirements are added. The work typically progresses in highly iterative manner where new increments are added and the understanding of the feasibility to satisfy specified requirements improves during the iterations.

The main contributions of the paper are the following. We have identified the main steps of a general product development process and specified abstract viewpoints to support these steps. We have explained, how these different methods and techniques interrelate and how they can be used together during practical development. To achieve this, we introduced an integrated model that provides metamodels for each of the four viewpoints. Finally, we demonstrated our model with a worked example.

Acknowledgements

This paper has been partially done in CAFE project. CAFE is project 00005 in ITEA organization a part of Eureka Σ! 2023 program.

7. References

Cohen, S., J. Stanley, W. Peterson, and R. Krut (1992): *Application of feature-oriented domain analysis to the army movement control domain*, Technical Report CMU/SEI-91-TR-028, Software Engineering Institute, Carnegie Mellon University.

Czarnecki, K., and U. Eisenecker (2000): *Generative Programming — Methods Tools and Applications*, Addison-Wesley.

Ferber, S., J. Haag, J. Savolainen (2002): Feature Interaction and Dependencies: Modeling Features for Re-engineering a Legacy Product Line, *in the Proceedings of the Second Product Line Conference* (SPLC2), 235-256.

Griss, M., J. Favaro, and M d'Alessandro (1998): Integrating feature modeling with the RSEB. *In Proceedings of Fifth International Conference on Software Reuse*, IEEE, pp. 76–85.

Hillard, R., (1999): Views and Viewpoints in Software Systems Architecture, a position paper for *the First IFIP Working Conference on Software Architecture* (WICSA1).

IEEE (1990): *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE.

IEEE (2000): *Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Std 1471-2000, IEEE.

Jackson, M., (2001): *Problem Frames – Analyzing and structuring software development problems*, Addison-Wesley.

Lee, K., K. Kang, W. Chae, and B. Choi (2000): Feature-based approach to object-oriented engineering of applications for reuse, *Software Practice and Experience*, vol. 30, pp 1025–1046.

Lorentsen, L., A-P. Tuovinen, and J. Xu (2001): Experiences in modelling feature interactions with colored petri nets. Tibor Gyimothy, editor, *In the Proceedings of Seventh Symposium on Programming Languages and Software Tools* SPLST 2001, pp. 221–230.

Savolainen, J., and J. Kuusela (2002): Framework for Goal Driven System Design, *in the Proceedings of COMPSAC 2002*, 749-756.

Wirfs-Brock, R., B. Wilkerson, and L. Wiener (1990): *Designing Object-Oriented Software*, Prentice Hall.

Yourdon, E., (1989): *Modern Structured Analysis*, Prentice Hall.