The building block method. Component-based architectural design for large software-intensive product families
Müller, J.K.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
2 Context

The successful use of a method requires an understanding of both the method and its intended application area. In this chapter we give important definitions, such as our definition of the terms technical system, architecture and method, and describe the specific system context for which the BBM has been designed. We will start with a definition of the term system.

2.1 What is a system?

A technical system is an assemblage of parts forming a complex or unitary whole that serves a useful purpose [Wie98b].

Figure 1: System Theory: A System
Since we only refer to technical systems in our context we shall use the term *system* instead.

The parts must interact in a way that ensures that the system as a whole has a useful function for one or more entities in its environment (see figure 1). Systems may be hierarchically structured, that is, a system is part of an enclosing supersystem and the parts may be subsystems themselves. It is therefore important to indicate the system of interest one is referring to.

A system delivers a service to its environment by interacting with it. Interactions can be described as functions. The functions communicate with the environment. The ordering of interactions in time is called behaviour [Wie98b]. Functions, communications and behaviour are properties of a system.

The recursive notion requires to indicate the selected level when we talk about a system (see figure 1). The enclosing system is also called the environment of a system and establishes its operational context (the *why* of the system). The functionality of the system describes the *what* of the system and its internal structure the *how* of the system.

A system is developed according to a set of requirements. Requirements are often described via externally observable properties of the system. These properties concern interactions of the system with entities in the environment. The software system handles symbols whose meanings refer to items outside the software system. The part of the world referred to by these symbols is called the application domain of the system.

Wieringa uses the term *subject domain*, that is, the symbols refer to the subject domain of the system's external interactions [Wie98a].

Sometimes an explicit application domain model represents the application domain. The application domain model consists of domain entities and of functions, which describe relevant interactions between domain entities and the system [Wie98a] (see figure 2).

The functionality of a system, however, is not only determined by its interactions with the environment but also by the precise characteristics of these interactions. The interactions have to exhibit certain qualities. One often speaks collectively about system qualities like reliability, performance and security, abstracting from the fact that for different interactions the quality requirements are different. Furthermore, the characteristics of the used technology may influence the system functionality; for instance, potential failures may require error handling functionality. To summarise, system functionality is induced by the
required application domain functionality, the qualities required of the system and the used technology (see figure 3).

For the context of the BBM, the software of the system is the main focus of attention. However, it is clear that the software is part of the larger system enclosing software and hardware and of the system which form the environment (see figure 2). In the remainder of the thesis, we will simply use the term system to imply software and hardware, and we will use the term environment for the enclosing system. For the term application domain we will sometimes use simply domain.
2.2 What is architecture?

As a point of departure we take a definition given by Rechtin:

"..., the term architecture is widely understood and used for what it is - a top-down description of the structure of the system." ([Rec91], p.9).

The word architecture, traditionally established in the field of construction, has been used for some time in different engineering disciplines to denote the top-level structures of complex systems. In these disciplines, especially the activity of architecting, in contrast to traditional engineering, puts strong emphasis on an overall design task [RM97]. Architecting denotes the activity, which balances functional requirements, available technologies, interests of involved stakeholders and desired system qualities to create an overall architecture (see section 2.6). Rechtin and Maier [RM97] adapted the definition given by the Webster’s Dictionary for architecting in the context of construction to define:

"Architecting is the art and science of designing and building systems".

Architecture, therefore, is a specific level of design. Design means the arrangement of parts or details of something so that a set of qualities is achieved ([Wei88], p9). The architecture provides a development context and builds the basis on which typical engineering tasks take place. This is the meaning of the term *top-down description* of Rechtin’s definition (see above).

We distinguish three different types of architectures: system architecture, software architecture and family architecture.

The *system architecture* consists of at least the following elements:

- the system structure which is broken down into hardware and software components;
- the externally visible attributes of these components, such as interfaces, resource usage, and other quality characteristics;
- constraints imposed on the component designs to achieve the desired properties of the system;
- system standards that must be met by all components.

Additionally, the *software architecture* should be able to handle complexity
by describing views that support understanding of the SW system from different perspectives,

by providing mechanisms for an incremental SW system which is assembled from a kernel and successive additions,

by means of a modular design which lets the most likely changes be implemented locally within one or a few components.

Moreover, a *product family architecture* should be *future-proof*, which can be characterised as

covering a whole family of current and future products, enabling reuse;

providing for extensions with the most likely features a customer may want.

---

2.3 What is a method?

The term method is very generally defined by the Webster’s Dictionary [Web13] as an

"orderly procedure or process".

However, we want to base our use of the notion of *method* on the definition given by the Esprit project ATMOSPHERE [Kro93]

"... a method is defined as consisting of: An underlying model, a language, defined steps and ordering of these steps, guidance for applying the method".

The ATMOSPHERE project considered this definition particularly useful in the context of systems engineering of software-intensive systems. The systems for which we have developed the BBM belong to this domain. The core BBM addresses large software-intensive systems, while the specialisations address a specific class of software. Methods, which aim at a very wide application area, can only be very general. The advantage of a restricted application area is that guidelines and examples can be more specific. In the context of the usability of general methods in engineering, Michael Jackson says:

"There is no place for constructive or universal methods. The methods of value are micro-methods, closely tailored to the tasks of developing particular well-understood parts of particular well-understood products" [Jac98].
In this thesis not all elements of a method as defined above will be given equal attention. We will pay most attention to explaining the underlying model, the so-called product part [Kro93]. The underlying model is introduced in chapter 3 and compared to underlying models of other methods in chapter 9. We do not intend to describe a language for objects, aspects and processes. Languages of other methods may be used, e.g. UML. The so-called process part [Kro93] of the BBM is given in two chapters. In chapter 3 we introduce the main design tasks of the core BBM and give an overview of the design steps of the design tasks. In chapter 11 we address issues of development process and organisation. Guidelines and examples are given throughout the text.

2.4 Models and Meta-Models

Another important point for positioning the BBM is its relation to real systems and models of these systems. The BBM is a design method for architecture. It explicitly contains a notion of what architecture consists of. This we call the architectural meta-model of the BBM. Architecture itself, however, is a model of/for a real system. To make this relation more explicit, we will compare it with the modelling of data (table 1).

<table>
<thead>
<tr>
<th>Abstraction Hierarchy</th>
<th>Data modelling</th>
<th>Data modelling Example</th>
<th>Architectural modelling</th>
<th>Architectural modelling Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model of the Model</td>
<td>Meta Data Model (Data Model)</td>
<td>Relational Model</td>
<td>Meta Architectural Model</td>
<td>Architectural Meta Model of the BBM</td>
</tr>
<tr>
<td>Model of the Real World</td>
<td>Data Model (Scheme)</td>
<td>Tables of inhabitants register</td>
<td>Architectural Model</td>
<td>SW Architecture for Product Family</td>
</tr>
<tr>
<td>Real World</td>
<td>Data</td>
<td>Inhabitants of Eindhoven</td>
<td>System / Product</td>
<td>Products of a Family</td>
</tr>
</tbody>
</table>

| Table 1: Analogy of Models

A meta-model has been established in data modelling, which has a long modelling tradition [Dit97]. Beginning with the concrete data instances, a data model describes
the facets of reality, which are relevant for the modelling purpose, e.g. a collection of tables or objects. The data meta-model describes the allowed structures for data models, e.g. the relational model or an object-oriented model. The terms in parentheses give alternative terms, which are sometimes used. The second column gives a concrete example.

Similar to the modelling of data, the SW architecture of a product family is a meta-structure for a concrete product family (table 1). A product family is dependent on its architectural model which describes a set of possible product family implementations. The architecture in itself is again constrained by the possibilities of the architectural meta-model of the BBM. The BBM has sufficient intrinsic complexity to allow the design of sufficiently flexible architectures for families of software-intensive systems (see section 9.1). Other methods and approaches, which we will discuss in chapter 9, are not sufficiently rich for creating architectures that show the required qualities.

2.5 About Some Terms

The terms we use to describe the context of software-intensive systems come from different areas such as telecommunication systems, systems engineering and software engineering. We do not intend to invent any new terms because the terms are only important for setting the context for the BBM. We will mention some of the terms' different meanings. If no further qualifications are given, it should be clear from the context which meaning is intended.

In telecommunication systems, an important distinction is made between system control and system management. Management comprises functions of a system, which are operator-oriented. System control comprises functions, which are automatic-action-oriented, for instance for recovery and graceful degradation. The background of this distinction is that high availability systems cannot rely on operators only. However, it is not common to make this distinction explicit in describing the functions configuration management (CM) and fault management (FM). The term fault control is not customary in telecommunications, while the term configuration control is used synonymously with configuration management. Therefore configuration management and fault management comprise functions of both system management and system control.

Another problem comes from software engineering, in which the term configuration management refers to functions in the software development environment. In this respect our use of configuration management could be called on-line configuration management.
Yet another area of potential confusion is the use of the terms physical and logical. In telecommunications, physical equipment or physical resources are the touchable or real or hardware things. Logical equipment usually refers to some data representation of physical equipment, while, in addition, a logical resource may also be a resource from the domain of computing like a file or a buffer.

2.6 A Rational Architecting Model

In this section we sketch a model of architecting to explain the context we assume for the use of the BBM. Several tasks are described in terms of their results. The inputs and outputs of the BBM are taken from this model.

The model for architecting (figure 4) relies on a model described in [AMO*00]. It is a rational architecting model in the sense of Parnas and Clements [PC86], that is, the model describes not an actual process but a rational process reordered according to the logic of the cause and effect. Earlier tasks are the causes for later tasks. The model starts with the question about what the essence of a customer use of a product is. It then concentrates on the application of the product itself. The third task deals with the design from a commercial point of view. It specifies the functions and features of a product and gives its commercial rationale. The fourth task is the architectural design from a product construction point of view. This is where the BBM fits in. The last task is about technology and determines technologies used for implementation.

| Customer Business Modelling | Application Domain Modelling | Commercial Design | Architectural Design | Technology |

Figure 4: Model for Architecting

The five tasks can also be characterised by the questions they answer. The first two tasks, customer business modelling and application domain modelling, are about the customer and the problem domain. The first task deals with the customer’s objectives, that is his business goal. The second task addresses the use of the system by the customer, that is the application. Together they answer the question why the product is needed. The last three, commercial design, architectural design and technology, are concerned with the product or the solution. The
commercial design answers the question what the product must be capable of. The last two answer the question how the product is/must be built.

Note that in traditional object-oriented development there are only two tasks, application domain modelling or analysis and system design. Design decisions are motivated directly by requirements from the application domain.

Moving to a task on the left gives the reasons why something is done, while moving to a task to the right gives the means how something is done.

We use such a wide model for the process of architecting to provide anchors for reasoning in the product development context. That does not mean that architects have a prime responsibility for all of these tasks. In fact they have prime responsibility only for architectural design. But they will be involved in the other tasks as well and have to take care that technical decisions and commercial decisions are aligned over the whole process of architecting.

We will use in the following the term *modelling* to denote that the character of a task is more concerned with faithful rendering and the term *design* to denote that the character of a task is more concerned with purposeful arrangement. But that is not black and white and all task have facets of both characters.

### 2.6.1 Customer Business Modelling

Customer business modelling is about understanding what the customer really wants. Leading questions for this task are: What is essential to the business of the customer? How does he make his money?

This task results in the customer value drivers, the customer business models and the model of the markets the customer is in, that is its competitors and complements [Mul02]. This is important to understand because it determines the value a product has for the customer. The customer value drivers are an important source of reasoning about priorities of product development. A product has to be built such that it supports the achievement of these values.

For example, a telecommunication manufacturer needs to understand consumers, which are the customers of the telecom operators. They want an easy-to-use and reliable service. A telecom operator makes money with the duration of connections implying that he wants to have a high rate of successful calls. Call facilities such as call forwarding, voice box and automatic ring back are a means for more successful calls.

In a similar way, a manufacturer of medical imaging equipment, which sells to radiology departments of hospitals, needs to understand that the quality of the images
taken from the various parts of the human body is of prime importance to a radiologist in being able to make a good diagnosis. A second value driver is the efficiency in which a radiology department can handle patients. The average time, which is needed per patient examination, is an important indicator in this respect. A third value driver is the safety of patients and operating personnel. Radiation has to be reduced to the necessary minimum and equipment handling has to be such that patients and operating personnel are safe from injuries.

2.6.2 Application Domain Modelling

Application domain modelling deals with capturing the application in a model, which can be used as a basis for product design. Important subtasks are the identification and scoping of the application domain itself, the identification of application stakeholders and the modelling of the application. Domain modelling models the application context of a system as shown in figure 5.

![Figure 5: Application Domain Modelling](image)

The stakeholders of the application are actors in the environment of the system and are modelled as domain objects. It is important to analyse what they do and how they view the system. Note that this may lead to conflicting views about the system. Conflicts are not resolved at this task, it is just important that a faithful rendering of the various views take place. The resolution of conflicting views may be done at commercial or architectural design where different strategies may be taken. Typical examples are the design of different products or different features which emphasise the one view or the other or take various compromises.

For example, when we do customer business domain modelling we look at the commercial stakeholder of the customer whoever that may be; the user itself, a customer
or a financial department of the customer organisation or other. In a similar way, other stakeholders such as product managers, marketing people, developers, testers and maintainers are part of the development organisation.

**Domain Object Model**

The domain will usually be modelled in a domain object model, as shown in figure 5. It captures the entities of the domain as objects, determines their behaviour and their relations to other objects. The externally visible behaviour of the systems to be built is also part of the application domain model. The language used is that of the customer (or user) and the product managers. The intention of the domain object model is to describe

what the system embraces,

the system’s environment, e.g. with which interfaces it has to comply, and

other constraints for the system.

The domain is analysed to identify objects using heuristics.

Objects include physical entities, such as the controlled equipment and equipment interfacing with the system, as well as logical entities such as images, image sequences, telephone calls and speech announcements. In general, an approach for object identification as described in OMT [RBP*91] can be applied. Objects are identified by extracting nouns from domain descriptions. These tentative objects are then reduced by eliminating spurious objects. [RBP*91] describes guidelines for selecting spurious classes.

Domain object modelling has to be performed for the functionality of the entire product family. The reason why we use the term *domain object* instead of *object* is to emphasise this need for a modelling technique that extends specific system requirements to those requirements of a family or, even more, to the application domain as it exists independently of the specific system family.

Rumbaugh [Rum94] makes the distinction between domain and application explicit. He uses the concepts of a domain object and an application object. Application objects are computer aspects of the application that are nevertheless visible to the users. We come back to this difference in section 4.1.1.

The necessity to base object modelling on the functionality of the complete product family is underscored by the observation that systems often have an unstable object structure during their initial versions. This is because requirements are often oriented at a specific customer only. The stopping criterion for refinement
of the object structure is one of stability or robustness with respect to foreseeable evolution [JCJ*92].

The domain object model is essentially a model for communication between stakeholders. Additionally, it will be taken as first domain-oriented decomposition of the system [Wie98a]. The BBM uses it as input for its object design.

**Behavioural Modelling**

Behavioural modelling is an essential part of the application domain modelling. Behavioural modelling consists of several steps. First, all relevant domain processes in which the intended systems will participate are described. Then, the interactions of the stakeholders in the domain are described by use cases. Use cases are analysed for their elementary activities. For each activity objects and their interactions are described. Activities are, then, described as object interaction flows. State modelling in connection with attributes and relationships will be used to capture behaviour of the actors of the domain, their actions and automated actions in the system context and by the system itself. The model of the domain behaviour, consisting of domain objects, their interactions and their internal states, will be used as an input for concurrency design of the BBM.

**Modelling Technique**

To model the application domain techniques such as object modelling, dynamic modelling and functional modelling of OMT [RBP*91] may be used. A more recent technique is the use of the Unified Modelling Language UML [Fow97].

Another technique is role modelling as described for OOram [Ree96]. Instead of modelling objects in classes, object interaction is captured in role models. An object plays a certain role in such object interactions. Real world phenomena are described by a number of collaborating roles. Objects are, then, composed from their roles. This is one of the major advantages over traditional object modelling which is class-oriented.

Feature modelling [CE00] is yet another technique proposed for domain modelling. Based on the critique that the notion of an object implies state and behaviour, some have suggested to use more basic conceptual modelling based on perception psychology [CE00]. A further critique is that variability modelling in traditional OO cannot be done free of design. Single inheritance, multiple inheritance, parametrised inheritance, static parametrisation, dynamic parametrisation, as shown in [CE00], are used in OO modelling to express variability. The decision which of these concepts to use is already a design step and this should not be done during domain modelling. An application domain, therefore, is described using concepts as described in perception psychology. A concept comprises features and dimensions, that is, qualitative and
quantitative attributes. In [CE00], Czarnecki and Eisenecker propose hierarchical feature decomposition using several types of features such as mandatory features, alternative features, optional features and or-features. The aim is to create models of the configurability facets of concepts. This is an important modelling to use as a basis for the design of families of systems.

Feature modelling, as originally proposed in [Kan90], is widely used. For example, Griss et al. [GFA98] extended RSEB with feature modelling.

Let us take a look at a domain with maturity of domain artifacts, say cars. A lot of domain-specific objects are used to describe and compare cars. Moreover new cars are described in terms of the attributes of domain objects, e.g. the number of cylinders of the engine and its h.p., the type of gearbox, the interior design, maximum speed, and fuel economy. In a new domain, however, a description of a product uses functions of that product much more often than attributes of domain objects. The identification of objects is still a matter of system design.

As a domain matures and companies want to cover an entire application domain with their products, the focus of the domain modelling changes. While initially, the domain model described the functionality of a single product, as the domain matures, the domain model shifts towards a description of the domain itself.

Examples of domain objects from the telecommunication switching domain include subscribers, their access types such as analog or ISDN and their account, calls and call facilities such as call forwarding, call recording and automatic ring back.

We expect that a domain object model is constructed either directly or indirectly via other models like role models and feature models. The domain object model is a first decompositioning of the system’s functionality. It is the basis for the design steps of the BBM. Several analyses and refactorings are applied throughout the different design steps of the BBM.

2.6.3 Commercial Product Design

Commercial product design is about the design of products to address certain market segments. Products have to be commercially feasible before their development is started. We assume that an analysis of the market is made. Potential product features have to be assessed for their commercial value. From such com-
Commerially viable features products can be designed. This is shown in figure 6.

Feature-Centric Transition:
market segment → products

Figure 6: Feature-Centric Transition

Features are used to make the transition from market segments to products. Products are realised by a cluster of features. Feature matrices are a means that are often used to describe the mappings (figure 7). A product may be used for a certain market segment if it has all required features of that market segment (figure 6)

Figure 7: Feature Matrices

Sometimes the notions of functions, features and options are used. Functions are the main blocks of functionality. Features extend the functions towards certain applications. Options are mainly technology-oriented variants for products. However, for simplicity we will rely only on the notion of a feature, which may also be a function or an option, since we do not further exploit the differences.
As shown in figure 8 a product is configured from features. A base product consists of a number of mandatory features. Possible extensions to the base product are determined by moving through the *railway diagram* downward. Each feature which is passed is selected.

![figure 8: Base Products and Features](image)

In initial markets often a basic product is sufficient. More mature market require in addition to basic features also differentiating features. Product families, for instance, are defined in mature markets to address a variety of customers and applications, both in parallel and as a series of products in time. Mature markets have the advantage that it is easier to predict and plan new products.

Commercial design needs input from architectural design and technology in form of technical feasibility, development effort and cost. This illustrates that the architecting tasks (figure 4) are only a rational process and do not describe a process like the waterfall model.

Requirements and functional specification are two views upon a system's functionality. Requirements describe the system from the perspective of a customer or user, while functional specification describe the system from the perspective of the developing organisation. Both rely on concepts defined during application domain modelling. Additionally to the wishes of the customers, requirements from the development organisations such as its way of doing business, its capabilities and ambitions have to be taken into account.
The most basic function of the application domain model is its use as a common terminology base. Throughout the specifications the terminology of the application domain model will be used. Furthermore, the application domain model will specify relations and behaviour of domain objects. Feature specifications can reference the application domain model and this way be more concise. The level of detail necessary for specifying features depends on the details given in the application domain model.

Also, possible conflicts of stakeholders have to be resolved. There may be conflicts between application stakeholders or between an application stakeholder and a development organisation stakeholder. Even more complex stakeholder scenarios are possible where other organisations are involved, such as buying organisations or maintenance organisations.

Related to stakeholders, quality profiles have to be defined for products.

Not only specification of functionality is important but also the reliability, performance and security of products. Quality profiles, in general, may specify run-time qualities, but also build-time qualities like extensibility, evolvability or maintainability. Furthermore, budgets of development time and resources have to be decided on. Architects will have a major input in the tasks of commercial design.

The output of the commercial design is a mapping from features to products. This mapping is an important input for the product family design of the BBM.

### 2.6.4 Architectural Design

Architectural design is about the structuring of the actual implementation. It is the area of applicability of the BBM. In the following we will describe the input and output of the BBM. But before we go into detail of the BBM we address the split between hardware and software.

**Hardware - Software Split**

Up to now we have not really made a difference in the description of the architecting process between hardware and software functionality.

If we only think about developing a software product, the hardware issue is one of selecting the necessary deployment platforms. However, in the more general case we have to make design decisions about which parts are implemented in hardware and which in software. This is the issue of what is often called system architecture. Several approaches exist here. We may rely on standard hardware modules or we partially design our own HW (HW/SW co-design or sequentially ordered HW before SW). A factoring of HW and SW functionality
needs to take place. In the BBM hardware-implemented functionality is factored out. This factoring is part of the object and aspect design tasks. The design of the hardware is outside the scope of the BBM.

**BBM Inputs**

One of the major inputs of the BBM is the application domain model. It provides a first domain-oriented decomposition of the products (see figure 9). Other inputs are the commercial design outputs in terms of features, product specifications and quality profiles. The last input are technology choices which are determined in the technology task (see section 2.6.5).

Note that stakeholders are not mentioned directly as inputs. However, they are implicitly present through domain objects and behaviour reflecting their needs, and through quality profiles.

Taking the application domain model as first input means that very important modelling has to take place prior to the use of the BBM. It also means that traditional OO development methods can be used together with the BBM (see section 9.2). They provide an object and behaviour model which is used as a problem decomposition used as input for design in the BBM. The BBM takes that input and refactors it to create construction elements and architectural models.

The question might be asked if it is useful to rely on the output of another method and if a description as a single method would not be much clearer.

An important point is that architectural design does not only directly depend on domain modelling but also on features from commercial design, which is often left out in traditional development models.

Furthermore, an analysis of the complete system functionality reveals that the domain functionality is only a small part of it. An important input for this additional functionality is derived from required quality profiles. This will get specific attention in the BBM.

An all-embracing method would be very complex and inflexible.

**Architectural Models and Construction Elements**

The artifacts of the BBM, that is its output, (see figure 9) comprises architectural models and construction elements.

As described by Lakos [Lak96] the evolvability and extensibility of large systems depend to a high degree on the locality of the changes necessary to implement new functionality. That means the physical structure of the source code is important. In
the design of large systems we made the same observation. This is the reason why the BBM emphasises the design of the construction elements.

The BBM leads to several architectural models. A structural view (see [P1471]) is given by the BBs and their dependency relation. An object model showing the relations between design objects. Aspects defined by the BBM for the development of a product family have their own design model. The deployability design maps the functionality to the (possibly distributed) hardware. The concurrency design gives the mapping of functionality to processes and threads within hardware units.
2.6.5 Technology

The technology task concerns the technical choices which are not part of the architectural design. This may be different for different products. The technology choices and technology roadmaps are an important input for the architectural design because they provide the basic means which are used for building systems.

As an example of the variation of the content of the technology task take the hardware, which may be selected as part of the technology task or developed as part of the architectural design task. Similar for other technology choices like operating systems or operating system versions. If a product is to run on Windows and Unix platforms, the specific operating system and its version is part of the technology task. If only specific real-time kernels are used they are part of the architectural design. Other technology choices concern monitors, networks, computing infrastructure, libraries and component models.

2.6.6 Feedback, Navigation and Learning

Having described the architecting process in five major tasks does not mean that they are executed in this order. As occasionally mentioned throughout the description of the tasks, feedback between the tasks is essential.

As Muller states in [Mul02] the actual way of working may even start with the last task, for example, in the case of technology exploration. Starting to experiment with new technology may give ideas of what type of applications are possible to build with it. Architectural consequences are explored and commercial scenarios developed. However, the rational design process remains from left to right (see figure 4).

Navigation through different architecting tasks is one of the essential means for achieving consistency within the architecting process. Major design decisions should be traceable backwards to the customer value drivers.

For example, the questions what does the customer really need? and how does a specific technical function contribute to his value creation? are important to ask at points of design decisions.

Similar is the tracing of diversity. The breadth of the architecting process allows to identify the anchors of diversity.

They include different market segments, different customers, different stakeholders or technology choices. The products should be made such that they combine maximal appeal to the customer with the lowest possible implementation diversity. For example, data parameters are preferable over code diversity.
2.7 Levels of Consolidation

An essential concept of the BBM is the concept of a component framework. Component frameworks are a way to encode domain experience in a reusable asset. Product families are built by developing new plug-ins to existing component frameworks or by refactoring existing BBs into component frameworks and plug-ins.

As we will explain below, the terms generic and specific BBs will be used for the concept of a component framework and plug-ins.

An orthogonal view is presented by Roberts and Johnson [RJ96]. They describe a pattern language for stepwise evolving reusable assets from a set of common examples to the design of a domain-specific language. Several successive levels of consolidation are shown in [RJ96]. The first consolidation step leads to white-box frameworks using inheritance; the second to so-called component libraries; the third to black-box frameworks; the fourth to the use of a visual builder and the last to the development of a domain-specific language.

We had similar experiences as those described in [RJ96]. In the development of the tss product family (see appendix A) the data definition database was developed to generate parts of BBs which were related to system infrastructure generics. Further generation tools were discussed but not developed. In the development of the tss product family the development of new component frameworks was more urgent than easing the development of new applications by providing a domain-specific language for similar plug-ins to existing component frameworks. This, however, was a pragmatic argument for the tss development. In other areas domain-specific languages were developed successfully [DK98].

Component frameworks are chosen deliberately as the level of consolidation for the BBM. First, the BBM is a method for determining components of product families. This means that not every product development will use the BBM. In the development of initial products one will focus on achieving a product which will succeed in its market. Components are a good engineering means for independent development. However, component and their interfaces will usually not be stable initially. Refactoring is not an accident but a necessary second phase after the first principal development.
Product families may be conceived early but experience shows that components and component frameworks will need two or three redesigns to become stable [RE99].

Systems which already have stable component frameworks may be further consolidated by developing domain-specific languages. However it is important to go through the consolidation path level by level. Otherwise focus is easily shifted from making successful products to developing fancy technology. [DK98] gives advantages and disadvantages of developing domain-specific languages.

### 2.8 Historical Background of the BB Method

Twenty five years ago, the software of telecommunication infrastructure systems was quite commonly structured in vertical blocks. Based on the operating system, the application software consisted of the three vertical function blocks: configuration management (CM), fault management (FM) and call handling (CH) (figure 10).

```
CH CM FM
```

**Figure 10: Dependent Functional Block Structure**

This resulted in a strong interdependence of the functional blocks. System tests were only possible using the entire software system. It was very hard to extend the software with new functionality because of these relations. For instance, CH handled calls, CM dealt with the configuration of call-handling functions, and FM handled call failures. New call functionality had to be distributed over these three blocks.

A next step was to factor out common functionality from the application and build an infrastructure layer. This also contained the system’s database. Further-
more, communication between the functional blocks was restricted to proceed via services of the common infrastructure (figure 11).

![Diagram of Independent Functional Block Structure]

**Figure 11: Independent Functional Block Structure**

A further step was to support the introduction of new features directly through modular structuring. Instead of distributing new functionality over the existing software structure, a feature is encapsulated in a module and can be introduced into and taken out of a system. The required infrastructure now comprises also basic services of CH, CM and FM (figure 12).

![Diagram of Feature-Oriented Application Structure]

**Figure 12: Feature-Oriented Application Structure**

The BBM further generalises horizontal layering as a means of structuring the entire system (section 7.4). A system is developed layer after layer. CM and FM are handled as aspects (chapter 5).

Exchange of single modules on the target system was first used for fast error correction only. The BBM lifted this principle to the system level to extend/reduce the system functionality by means of product features (section 8.3). The design of the tss product family is described in appendix A.
2.9 Overview of the Concepts

We end this chapter on the context of the BBM with an overview of the concepts used in the BBM. The concepts, the chapters in which they are described and the main relations between them are shown in figure 13. Higher-order architectural concepts have been derived from general architecture requirements. They are expressed in terms of basic architectural concepts.

Note that for the concept component framework [FS97] we use the term generic Building Block (see section 7.5.1) for historical reasons.

![Diagram showing the concepts of the BBM and their main relations.](image)

**Figure 13:** Concepts of the BBM and their Main Relations

The next chapter will give an overview of the BBM.