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.
3 The Core Method Overview

The BBM is a component-based architectural design method for large software-intensive product families. Its emphasis on large software-intensive systems implies a focus on the actual construction elements of software systems. The feasibility of evolution and extension of large systems is largely determined by its deployment structures. Development effort can only be limited if changes are limited to a small number of deployment units.

Lakos [Lak96] names several problems in the development of large-scale software:
- poor encapsulation which hinders reuse and hampers testability;
- circular dependencies which leads to tight physical coupling making effective modular testing impossible;
- excessive link-time dependencies artificially increasing the deployed code;
- excessive compile-time dependencies increasing the time to not only recompile the complete system but also the time for each translation unit, for instance when a globally visible include file has to be updated; and
- global name space for variables leading to surprising name clashes.

The BBM uses software components and restricts the allowed dependencies. In general, different kinds of modularities are employed to improve the overall modularity of the software.

The presentation of the BBM is split up into a core method and method specialisations. An overview of the core method is described in this chapter followed by a number of chapters, which present the method in more detail. A method specialisation is described in chapter 10.

The BBM consists of five main design tasks: object design, aspect design, concurrency design, composability design and deployability design. Like the description of the architecting process (section 2.6), the BBM is described as a rational design process. We start the description of the BBM by looking at its prerequisites.
3.1 Prerequisites of the BBM

The BBM as an architectural design method for large software-intensive product families requires that certain tasks of an overall architecting process (see section 2.6) are (partially) completed. We only require partial completion because the

**Figure 14: Prerequisites for the BBM**

BBM is not part of a waterfall-like development process model. The whole architecting process described in section 2.6 is a rational process only and does not prescribe a specific execution order.

As shown in figure 14, the scope of the product family should be determined. An application model in form of domain entities, domain functions and/or domain procedures and their most important relationships should exist. The functionality of the products in the family and the required system qualities should be specified. A commercial design should have identified commercially relevant features and their dependencies. Relevant implementation technologies should be decided on.

For all these steps, relevant for an architecting process, different methods exist and can be used in combination with the BBM.

These prerequisites are used as input for the design tasks of the BBM (see figure 22). Object design, aspect design, concurrency design, composability design
and deployability design take these inputs as starting points for their modelling. We will walk through the tasks and describe what they are about.

### 3.2 Main Design Tasks of the BBM

The five main design tasks: object design, aspect design, concurrency design, composability design and deployability design are presented in a rational order. Objects are identified. Functionality which crosses objects is identified as aspect functionality. The functionality of objects and aspects is mapped to threads. Objects, aspects and threads are packed into BBs. BBs are grouped into libraries and executables.

![Figure 15: Main Design Tasks](image)

An initial design activity will execute the tasks in the specified order. However, since the design tasks influence each other, they may be executed in arbitrary order. Each task produces results, which are taken by the other tasks as triggers for making their own designs consistent with that of the other tasks. The process stops when the results of each task are stable.

Now we explain the major concepts of the different main design tasks.
3.2.1 Object Design

The concept of an object is very general. We will use objects in the context of the BBM at four different levels. Application domain objects describe entities of the application domain. Hardware domain objects describe elements of the hardware system. They are both part of the first level. On the second level we have domain-induced objects. They are a mirroring of the objects of the first level into the software design space. They are generated from inputs of the BBM. On the third level we have design objects. They are refined and refactored due to the design tasks of the BBM. On the fourth level we have implementation or programming language objects. They are mostly a mirroring of design objects. However, specific implementations may introduce new objects.

A detailed description of object design is given in chapter 4.

3.2.2 Aspect Design

The set of aspects is a partitioning of the complete functionality of a system. The application domain functionality is only part of a system's functionality. Other functionality is induced by quality requirements. This additional functionality is often larger than the application functionality. To construct the aspect partitioning, certain types of functionality are identified and factored out. Initially all functionality is said to be part of the operational aspect. From the operational aspect those types of functionality are factored into aspects which crosscuts domain-induced objects (see figure 16). An application domain object such as telephone call or a medical examination needs to be initialised, to be configurable, to deal with erroneous situations and to support operator interaction. These types of functionality are factored out as separate aspects. The remaining functionality will define the operational aspect.

For each of the factored-out aspects we can make designs, which apply throughout the system and give the system design a certain uniformity. Common
implementations are factored out in specific component frameworks, called system infrastructure generics.

Aspect design takes the quality specifications as input. They are analysed for necessary additional functionality to achieve the qualities. Additionally, an architectural concern analysis based on checklists from prior design experience is performed to check for comprehensiveness of the specified functionality. Both analyses may result in additional aspects and objects.

A detailed description of aspect design is given in chapter 5.

3.2.3 Concurrency Design

Concurrency design is about mapping of functionality to processing resources. A concurrency structure is designed for the complete system and will be expressed in aspects and/or objects. Concurrency design starts with the behaviour of the application domain and results in a concurrency model consisting of threads.
Objects, aspects and threads are independent and span a design space of three design dimensions (see section 3.3). This means that a thread may involve one or more objects or one or more aspects without design restrictions.

A detailed description of concurrency design is given in chapter 6.

### 3.2.4 Composability Design

Composability design is about defining modularity to support the composition of products in the product family, to obtain manageable development units, to realise a simple feature mapping and to allow for incremental integration and testing.

BBs are design and deployment units, which are identified in the architectural phase [Szy98]. There are two important questions with respect to BBs: What is the content of a BB and what relations exist between BBs? The identification of BBs starts with partitioning the network of objects. A BB is initially a cluster of related domain-induced objects (see figure 18). BBs are refactored to contain clusters of design objects. BBs do usually not contain entire processes or aspects. The main criterion is configurability and situations are possible where an aspect or a process is itself a unit of configuration. The set of BBs covers the entire functionality in a non-overlapping way.

The BBs are technically the dominant decomposition [TOH99] of a BBM-based system (figure 19). The possibility to assign functionality along one of the three axes (object, aspect and thread) or a combination thereof provides flexibility to choose the decomposition which best supports the evolution of the product family. The tyranny of the dominant decomposition [TOH99] is thus avoided.
Component Frameworks

An important point in the design of a product family is the separation of generic and specific functionality. Generic functionality is implemented once and is used by other BBs. Component frameworks, called generic BBs in the BBM, and plug-ins, called specific BBs, are designed to encapsulate generic and specific functionality.

Incremental Layering

Relations between BBs are derived from the relations between the encapsulated clusters of objects, from relations created by splitting up aspects and threads and from inter-aspect and inter-thread relations. The BBM restricts these relations by requiring that the dependency graph forms a partial ordering of all the BBs. Thus, additional design may be necessary to conform to this restriction. In a graphical representation we mostly use lines without arrowheads. BBs being located higher in a figure depend on BBs located lower (see figure 20).
BBs are designed such that they can be integrated and tested layer by layer. Such layering of BBs is called *incremental layering*. Layering is used on two levels. Coarse layers are derived from the layers introduced during object design. These layers are refined such that the coarse layers are also internally layered. Specific kinds of generic BBs are used to allow exchange of BBs in lower coarse layers. Incremental layering and *plugability of BBs in lower layers* are key concepts of the BBM.

**Architectural Skeleton**

Taken together the generic BBs form an architectural skeleton on different layers. The *architectural skeleton* is the basis for the product family architecture.

**Features and Product Family Architecture**

Commercial product features are another input of composability design. The dependency structure of features should be reflected in the BB dependency structure. As commercial features describe a product commercially the product should be buildable from BBs which implement these features. This is called feature orientation of the product family architecture. The design of an architectural skeleton is a means to create a *feature-oriented product family architecture*.

A detailed description of composability design is given in chapter 7 and chapter 8.

**3.2.5 Deployability Design**

Deployability design is about possible deployment scenarios of the products. The input for deployment scenarios may come from requirements for geographic distribution or from technology assessment requiring a certain HW partitioning. Geographic distribution, if required, will often come directly from the customer. This input may lead to a refactoring of objects, threads and BBs.

The deployment model defines the allocation and/or allocatability of deployment sets to hardware instances. Deployment sets consist of BBs or clusters of BBs.

A detailed description of deployability design is given in chapter 6.
3.2.6 Implementation of BBs

BBs are also units of implementation. The functionality of BBs is defined during object and aspect design. However, object design, aspect design and thread design are about design and not directly about implementation.

Implementing a BB in an OO language means that everything is implemented in objects. One may use design objects directly during implementation as implementation objects. This means that aspect functionality is implemented by aspect-specific object (or class) methods. However it is also possible to introduce a fourth level of implementation objects which implement the functionality of an aspect. A design object, then, is represented by a set of implementation objects.

There is a similar relation between aspects and threads at the level of implementation. Both, aspects and threads, are not visible explicitly by programming language constructs. Aspect functionality is implemented by methods or objects as explained above. Our notion of thread is sometimes referred to as reach of a thread and consists of all methods and objects, which execute under its control.

In an OO language, an interface will either be implemented as an abstract class or via the interface construct.

3.2.7 Design Artifacts

The results of the main design tasks are accumulated as design fragments (see figure 22). Each new execution of a design task may update some design frag-
merits or create new ones. Other design tasks are triggered by updated and new fragments to make their own fragments consistent with the updated ones.

The results of the main design tasks are a set of architectural models and a list of construction elements. The architectural models are an object model, the list of aspects and their designs, the concurrency model, the BBs and their dependency relation, and the deployability model (see figure 22). The construction elements are the list of BBs and their designs, executables, DLLs and data files.

In table 2 we give an overview of the steps which have to be executed per main design task. The steps are described and detailed in subsequent chapters: the steps of object design in chapter 4, the steps of aspect design in chapter 5, the

*Figure 22: Input + Output of Design Tasks*
### Design Task

<table>
<thead>
<tr>
<th>Design Task</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>object design</td>
<td>creating an initial object model</td>
</tr>
<tr>
<td></td>
<td>adapting the object model to required functionality</td>
</tr>
<tr>
<td></td>
<td>factoring out HW-implemented functionality</td>
</tr>
<tr>
<td></td>
<td>modelling HW resources in SW</td>
</tr>
<tr>
<td></td>
<td>refactoring domain-induced objects into layers of design objects</td>
</tr>
<tr>
<td></td>
<td>creating design objects for communication, interfacing, registration, containers and aspects</td>
</tr>
<tr>
<td>aspect design</td>
<td>initially taking the complete functionality as one aspect</td>
</tr>
<tr>
<td></td>
<td>analysing domain-induced objects for crosscutting functionality</td>
</tr>
<tr>
<td></td>
<td>performing an architectural concern analysis to find additional aspects</td>
</tr>
<tr>
<td></td>
<td>using starter sets of potential aspects to support the aspect identification</td>
</tr>
<tr>
<td></td>
<td>standardising the list of aspects for the product family</td>
</tr>
<tr>
<td></td>
<td>determining the functionality per aspect</td>
</tr>
<tr>
<td></td>
<td>making a global aspect design</td>
</tr>
<tr>
<td></td>
<td>factoring out common implementation parts in system infrastructure generics</td>
</tr>
<tr>
<td></td>
<td>defining rules and guidelines per aspect</td>
</tr>
<tr>
<td>concurrency design</td>
<td>starting with behaviour of domain objects</td>
</tr>
<tr>
<td></td>
<td>determining independent external sources</td>
</tr>
<tr>
<td></td>
<td>prioritising aspect functionality</td>
</tr>
<tr>
<td></td>
<td>if necessary, encapsulating specific objects in a thread</td>
</tr>
<tr>
<td></td>
<td>refining the logical threads into physical threads</td>
</tr>
<tr>
<td></td>
<td>determining interfacing between threads</td>
</tr>
</tbody>
</table>

*Table 2: Overview of Steps per Design Task*
<table>
<thead>
<tr>
<th>Design Task</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>composability design</td>
<td>clustering objects into BBs</td>
</tr>
<tr>
<td></td>
<td>identifying variation points of functionality which belongs to different features</td>
</tr>
<tr>
<td></td>
<td>factoring out common functionality in separate BBs.</td>
</tr>
<tr>
<td></td>
<td>identifying interfaces of BBs</td>
</tr>
<tr>
<td></td>
<td>designing component frameworks and plug-ins</td>
</tr>
<tr>
<td></td>
<td>identifying system infrastructure generics</td>
</tr>
<tr>
<td></td>
<td>defining layered subsystems of BBs</td>
</tr>
<tr>
<td></td>
<td>designing for incremental integratability and testing</td>
</tr>
<tr>
<td></td>
<td>doing detailed design of the BBs</td>
</tr>
<tr>
<td>deployability design</td>
<td>determining fault containment units</td>
</tr>
<tr>
<td></td>
<td>determining possible deployment scenarios</td>
</tr>
<tr>
<td></td>
<td>packaging BBs to deployment sets</td>
</tr>
<tr>
<td></td>
<td>generating data files</td>
</tr>
</tbody>
</table>

*Table 2: Overview of Steps per Design Task*

steps of concurrency design in chapter 6, the steps of composability design and the steps of deployability design in chapter 7.

Besides the main design tasks, the BBM does not exclude other design tasks (see section 3.4 and section 3.5). The results of other design tasks are inputs for one or more of the main design tasks. In general, design models should be built for all relevant concerns to guide the development of the system. Preferably, design models should be quantitative, for instance by using design budgets for critical resources.

Further results are rules and guidelines, which have to be applied throughout the whole system design. Rules and guidelines together provide a set of internal system standards, which are essential for achieving and maintaining conceptual integrity [Bro75]. Rules and guidelines complement the design of individual components.
Many of the rules and guidelines will be related to aspect designs. Coding standards and resource usage are other examples. An example of the tss system is that search operations in lists were not allowed because the performance of the operation varies with the length of the list. A tss system with a high load would need more time. Instead, designs have to be used where elements can be selected from the heads or the tails of a fixed set of parallel lists.

Catalysis gives three classes of system standards: horizontal or infrastructure standards defined by the infrastructure to be used by all application, vertical standards which apply to all systems in an application domain, and connector standards which are to be used for intercomponent communication [DW99].

The execution structure of the BBM design tasks (figure 22) confirms to the blackboard style (see section 9.3.1). Such a general model raises the question when to stop the BBM design tasks.

### 3.2.8 Stopping Criteria for Design Tasks

An important question is about a stopping criteria for the various design tasks. This is especially important since we do not give a fixed order in which the design tasks have to be executed such that after the last task the design would be finished.

It is important to realise that architectural design is part of an overall development process. The time given to architectural design has to be decided in that context. An important internal criterion for stopping is when structures become stable and implementable.

When one traverses from one design task to the next, changes may be required for the structures designed in the first design task. Several cycles through the tasks are often necessary because technical systems are often at the edge of technical possibility and the applicability of prior designs is limited. Design experience and early feedback are important in dealing with this situation. Short development cycles after which various kind of users can give feedback are favourable.

Another point is the experience reported by several framework designers that framework interfaces need two to three redesigns to become stable ([RE99], [BGK*99], and also tss design experience). Feedback, again, is essential.
3.3 Design Dimensions

The idea of structuring of domain-induced objects, aspects and threads in multiple dimensions is introduced to support the freedom of system design. If design concepts, which address different facets of the same item, can be separated so that there are no mutual restrictions, the concepts are orthogonal. We can then talk about design dimensions. Every dimension can hence be designed independently by projecting each item in the design space to one dimension. The BBM identifies three specific design dimensions.

Note that this discussion is on the design level and not on the implementation level.

The first point is to keep object structuring independent from the use of execution units. Domain-induced objects result from object design (see section 3.2.1), which has as input the domain object model. Threads determine the use of processing resources for independent, cooperating and/or sequential actions.

The designer should be free to design threads without consequences for the design of objects. They constitute two orthogonal dimensions, i.e. a method of an object may be driven by one or more threads and a thread may drive methods from different objects.

Modules were separated from processes in [HFC76] and [Cla85] already.

The second point is to construct aspects orthogonal to domain-induced objects. Those global functions to which potentially all objects contribute are handled as SW aspects. Each object method in the system is part of one object and part of one aspect.

The BBM combines these two ideas. This leads to three design dimensions since threads and aspects are also independent, that is, an aspect may be driven by different threads and a thread may drive different aspects.

A Mathematical Formulation

From a mathematical perspective, the design independence is described by the three design dimensions: object dimension, aspect dimension and thread dimension (figure 23) forming a design space $D$. The design space $D$ is a discrete space. The values of the first dimension are object classes of the set $O$ of object classes. The values of the second dimension are aspects of the set $A$ of aspects. The values of the third are thread types of the set $T$ of thread types. Formally:

$$D = O \times A \times T = \{ (o, a, t) | o \in O, a \in A, t \in T \}$$
We shall use object methods as basic terms in our discussion of the design space. An object method is part of exactly one object class and of one aspect but may be driven by several thread types. Furthermore, several object methods may be part of the same object class and the same aspect and the same thread type simultaneously. This means that the points in the design space represent sets of object methods.

Formally, let $OM$ be the set of object methods. For all elements $(o \in O, a \in A, t \in T)$ of $D$ we define the following projections:

$$OM(o) = \{ f \in OM | f \text{ is a method of the object class } o \}$$

and

$$OM(a) = \{ f \in OM | f \text{ belongs to aspect } a \}$$

and

$$OM(t) = \{ f \in OM | f \text{ runs under control of a thread of thread type } t \}$$

The object dimension covers the decomposition of the system into object classes, thus:

$$OM = \bigcup_{o \in O} OM(o) \quad o_i \cap o_j = \emptyset \quad \forall i, j, i \neq j$$

by construction, that is:

$$OM = \sum_{o \in O} OM(o)$$

(The $\Sigma$ sign stands for a disjoint union of sets.)
The aspect dimension partitions the functionality into specific views, such as recovery, configuration management, fault handling, etc. Thus:

$$OM = \bigcup_{a \in A} OM(a)$$

$$a_i \cap a_j = \emptyset \quad \forall i, j, i \neq j$$

by construction, that is:

$$OM = \sum_{a \in A} OM(a)$$

The thread dimension describes overlapping subsets of system functionality which are driven by threads. Thus

$$OM = \bigcup_{t \in T} OM(t)$$

Then, we link points $o, a, t$ of the design space $D$ with sets of object methods by defining:

$$\langle o, a, t \rangle = OM(o) \cap OM(a) \cap OM(t)$$

We now see that object methods are basic terms: a point in the design space represents a set of object methods.

Furthermore, we can characterise object methods in terms of the design space. Let $f \in OM$:

With

$$OM = \sum_{o \in O} OM(o)$$

we obtain a unique $o_f \in O$ with:

$$f \in OM(o_f)$$

Analogously, we obtain a unique $a_f \in A$ with:

$$f \in OM(a_f)$$

However, with

$$OM = \bigcup_{t \in T} OM(t)$$

there may be more than one $t \in T$ with $f \in OM(t)$. 
The Core Method Overview

With \( T_f = \{ t \in T \mid f \in OM(t) \} \) we obtain:

\[
f \in \bigcap_{t \in T_f} OM(t)
\]

Summarizing, we obtain:

\[
f \in OM(o_f) \cap OM(a_f) \cap OM(t) \quad \forall t \in T_f
\]

In terms of the design space:

\[
f \in \langle o_f, a_f, t \rangle \quad \forall t \in T_f
\]

This means that an object method \( f \) is spread over the straight line parallel to the thread axis with object axis value \( O_f \) and aspect axis value \( A_f \).

The independence of the design dimensions is an important methodological starting point. This independence is complemented by design patterns which describe relations between specific objects, aspects and threads. They describe experiences of good design for a specific design context.

System Evolution and Design Dimensions

A system, which evolves in the three dimensions simultaneously, is very complex and hard to manage. It would be good if evolution could be restricted to a single dimension. The following tentative considerations argue that such a situation exists for the most common changes for central controller software. However, worst-case changes affecting all dimensions are also possible.

Changes common to, for instance, the central controller software (see section A.3) occur with the introduction of new controlled equipment, and new services for the environment. This will most likely result in new objects and less likely in new aspects. The extensions of a system are obtained by extending objects and/or introducing new objects. Changes are local since only a few components are affected.

The two design dimensions, object and aspect, make use of object-oriented and functional modelling. Aspects are seen as an alternative form of modelling. Aspects are functions which crosscut all or most objects. The list of aspects should be standardised for an entire product family to achieve stable software structures. Adding a new aspect or extending the functionality of an existing aspect affects all the related objects. Because BBs are, usually, a cluster of objects, adding a new aspect induces changes in most or all of the BBs. Locality of change does not exist when an aspect is added.

As explained in chapter 6, concurrency design starts by looking for execution independence of objects. In the case of most systems we assume that a situation is aimed for in which the structure of threads is standardised for an entire family, either
in specific threads or in rules which guide the creation of a concurrency model. Without such a structure, to understand a large evolving system is very difficult.

For those cases where the above considerations concerning stable aspects and threads hold, we say that the system evolves in the object dimension only. In such cases the change effort will be minimal because a new feature will be implemented in a few new and/or updated BBs.

An example (chapter 5) which does not follow the above-mentioned evolution is a product family where one product uses a general login procedure for the whole of the system and another product, functionally equivalent, uses specific login and access capabilities per object. The second product could be derived from the first one by adding an extra access control aspect. Implementing this aspect as part of the BBs would lead to a second set of adapted BBs. A better way to implement this is to create a specific plug-in per BB. However, neither are local changes any more.

3.4 System-Quality-Based Design Tasks

System qualities are important input for architectural design. They can induce specific functionality, cause the selection of specific mechanisms or influence the way in which functionality is implemented.

The main design tasks of the BBM are not grouped according to system qualities but the BBM design tasks have to take system qualities into account. However, the way in which the BBM deals with system design supports these design tasks. In the following we describe the relation of the design for various qualities with the main design tasks of the BBM.

3.4.1 Performance Design

Performance design is mainly done in the concurrency design. The use of separate threads for functionality with different priorities supports the design for timeliness. Functionality must be carefully factored such that time-critical paths are minimal.

But also other design tasks may be influenced by performance design. Examples are the selection of data structures or the trade-off between communication via data messages vs. the use of shared data.

In the tss systems, for example, transparent layers (see section 7.4.3.2) are used to avoid calling overhead and an in-memory database increases database update speed.
It must not be forgotten that performance design is also a hardware design issue. The hardware capabilities should at least permit a software solution, which meets the performance requirements. If hard realtime requirements are of paramount importance for the application, it is good design practice to factor out hard realtime functionality and assign it to a separate processor. Such an approach relieves application programmers from programming soft and hard real-time SW for the same processor.

If necessary, specific performance design methods must be used to complement the BBM. For example, rate monotonic analysis may be used to find an initial thread structure to meet deadlines.

### 3.4.2 Reliability Design

The topic of reliability design is best introduced with a quote from Birman [Bir96]: "Through decades of experience, it has become clear that software reliability is a process, not a property. One can talk about design practices that reduce errors, protocols that reconfigure systems to exclude faulty components, testing and quality-assurance methods that lead to increased confidence in the correctness of software, and basic design techniques that tend to limit the impact of failures and prevent them from propagating."

In such a setting reliability design with the BBM can be done via a number of different design concepts.

Specific aspects like persistency and recovery handling are means to improve the reliability of an application in the presence of HW failure. Persistency allows to keep system state over system crashes. Recovery handling consists of actions to recover from failures.

The aspect exception handling handles SW errors. The aim is to bring the system into a state which is presumed to be without error.

[Ren97] describes a pattern language for exception handling compatible with the BBM.

Transactions can be used to guarantee consistency of data updates.

The tss system, for example, used a database to explicitly administer persistent state information. The reliability design of the tss system is described in section A.3.3.2.

Note that the use of HW redundancy is very important for the design of high-availability systems. HW failures are handled by redundant HW, for an example see section A.2.2.
3.4.3 Security Design

Security is about preventing unauthorised users from making use of the system. Security design is done by choosing appropriate mechanisms for the design of access points and communication channels.

Examples are, sandboxing which provides a secure execution environment for foreign applications; encryption of stored and communicated data hinders unauthorised reading or change; capabilities are a way to structure various forms of user rights; and logging provides a history of events in a system.

Security can be handled as an aspect to structure overall access right handling.

3.4.4 Extensibility Design

Extensibility design is supported in the BBM task composability design through the design of various generic BBs. New features and applications can take advantage of a semantically rich infrastructure. New BBs can register themselves to these generics on the deployed systems. Necessary resources may be allocated via respective resource handling generics. The design for feature extension supports extensibility in general (see section 8.4).

3.4.5 Integratability and Testability Design

For large systems, integration, testing and the necessary rework contribute considerably to both the development time and the effort for product updates and new product features. It is essential that throughout the design of large products measures are taken to support integration and testing. The BBM supports integratability through BBs, component frameworks, incremental system integration and dynamic loading of deployment units in deployable systems (see chapter 7).

3.5 Other Design Tasks

Besides the quality-based design issues a number of other design issues are important. We give a short list of those issues. They present another perspective on the main design tasks of the BBM.
3.5.1 Feature Mapping Design

Product features are a major input for composability design. BBs are carefully factored to allow localisation of code, which implements the features. It is clear that product features are not completely isolated from the rest of the product. Therefore, feature relations, with the dependency relation being the most important one, are used. BB dependencies, which mirror feature dependencies, do not hinder flexible composition of products (see section 8.1).

3.5.2 Architectural Style Design

Architectural styles are an important means for designing the overall structure of a system. The most prominent architectural style of the BBM is layering. Objects are refactored according to layers in object design (see section 4.2) and BBs are refactored according to layers in composability design (see section 7.4). The BBM uses incremental layers for BBs to allow for incremental integration and testing.

Other architectural styles such as pipes and filters, and blackboards can be used in the main design tasks of the BBM. Both pipes and filters, and blackboards are architectural styles which describe the communication behaviour of objects. They are used during object design and, in the case where parallel execution is involved, during concurrency design.

The usage of architectural styles as single-view architectural approaches is discussed in section 9.3.

3.5.3 Data Structure and Algorithmic Design

Data structure and algorithmic design is mainly done during object design and aspect design. During object design data structures and algorithms are chosen for objects. During aspect design data structures and algorithms are chosen which support a complete aspect.

Composability design may lead to a refactoring of objects to place data structures and/or algorithms into generic and/or specific BBs.

System infrastructure generic may refactor objects to support certain aspects by providing generic data structures and/or algorithms.

If the existence of variation points leads to separation of data and operations a refactoring of objects for generic and specific BBs is necessary.
Furthermore, various kinds of interfaces require a design of data structures, for instance, interfacing between processes and threads may lead to shared data, buffers or queues.

### 3.5.4 Resource Usage Design

If resources are in a significant manner constrained, an explicit design of the usage of resources is appropriate. This leads to rules and guidelines about resource usage. It may also lead to the design of generic BBs which administer resources explicitly. System infrastructure generics are usually the place for administering pools of memory, I/O channels, file handles and thread classes and priorities. The same may hold for domain-specific resources where coordinated usage is supported by resource pools. Concurrency design deals with the factoring of code to enable an appropriate allocation of processor time.

### 3.5.5 Interface Design

Interface design is distributed over several other design tasks. Composability design deals with interfaces between BBs (see section 7.2 and section 7.6) and specifically between generic and specific BBs (see section 7.5). Concurrency design deals with interfacing between threads and processes (see section 6.2.2). Object design deals with external interfaces and for managed objects with distribution interfaces (see section 10.2.1).

### 3.5.6 COTS-Based Design

Commercial-of-the-shelf (COTS) packages can reduce the own development effort of an organisation. Examples of general COTS packages are operating systems and middleware packages for (graphical) user interfaces and communication. Various application domains are supported by commercial packages as well. COTS-based design is handled by composability design. Source libraries are included in BBs. Binary packages are dealt with as BBs.

An important question is the interfacing with these packages. Design strategies can be either to directly use the provided interfaces or to hide those interfaces behind some abstraction. There are several circumstances when extra implementation effort is required:

Binary packages require bidirectional linkage with a using package. A adaptation BB is necessary to achieve uni-directional coupling by providing a binding interface. The cluster of the binary packages together with the adaptation BB fulfils the requirements of a normal BB.
Binary packages may have resource allocation and usage strategies, which are not compatible with the rest of the system. A separate BB is necessary to shield this from the rest of the system.

Concurrency design may also be hampered if some packages are not thread-safe. A separate BB may handle application threads instead.

Products should have the flexibility to work with alternative packages possibly from different suppliers, which may have different interface abstractions. This may lead to specific interface BBs.

In general, the value of COTS packages depends not only on their provided functionality but also on the overhead and cost, which their use imposes on the system and the developing organisation.

3.6 Qualities of the BBM

We finish this overview chapter by taking a look at the qualities of the BBM. As noted in the beginning of the chapter, the focus on the actual construction elements is vital for the design of large systems. With the BBM, we develop both, global architectural models and construction elements.

Understandability of the architecture comes from these global models and the roles of layers and various types of BBs.

Factoring out of functionality into various kinds of generic BBs like component frameworks and system infrastructure generics leads to a certain leanness of the code [Wir95]. It has to be noted that generic BBs are not easily understood. But this is compensated by several advantages; first, generic BBs have explicit interfaces in contrast to OO frameworks [Szy98], second, the scope of the validity of system infrastructure generics is the complete system and third, component frameworks stand for domain-specific generic solutions. Furthermore, compared to monolithic frameworks, generic BBs are relatively small and there are many of them so that changes often can be kept local.

Leanness of products is supported by the fact that products are configured from the minimal set of necessary BBs (configuration to minimum, see section 8.3.4). The ease of building new products depends on the appropriateness of the architectural skeleton and frameworks for the new application. This is achieved by relying on the input from the application domain modelling.
Brooks defined conceptual integrity [Bro75] to be: "design conceived by a single mind". We refine his definition of conceptual integrity to be the suitability and orthogonality of a set of chosen design concepts for a certain system (class) as perceived by an expert designer. The BBM supports the achievement of conceptual integrity by focusing on the selection of design concepts from multiple perspectives. The explicit use of multiple perspectives provides a source for consistency across perspectives.