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.
In this chapter we describe the design task aspect design. In the first section the notion of an aspect as used by the BBM is defined. The second section describes the architectural concern analysis, which is performed to identify aspects. The third section describes an approach for deriving starter sets for aspect identification. The fourth section deals with the list of aspects in relation to the product family. The fifth section describes the design of aspect functionality. The sixth section takes a look at aspects in connection with BBs. The seventh section shows how aspects support architectural design in general. The last section places aspects in the context of multi-view design.

5.1 Definition of an Aspect

The motivation for the introduction of aspects is twofold. First, reasoning over and design of large systems is eased by explicitly identifying functionality which results from quality attributes and technology, and complements the application domain functionality. Second, upgrading of a system is eased by using unique designs for these different kinds of functionality. Application BBs which comply to these design are easy to integrate.

Aspects are a non-hierarchical, complete, functional decomposition of software functionality. To construct this decomposition, certain types of functionality are identified and factored into aspects. 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. A software aspect, except for the operational aspect, is a certain type of functionality cutting across objects.

The list of aspects should be anchored in the application domain and the operational context of the product family and is defined for the entire family.
To identify software aspects we first look at the high-level functions of the system to be built. Relating these functions to the identified domain-induced objects will result in one of the following cases:

- A function has relations to and/or defines functionality of a few objects only, for instance billing in telecom switch;
- A function will be used by almost all the objects, for instance logging;
- A function defines a type of functionality which is part of almost all the objects, for instance error handling.

In the first case the function will be handled as (part of) some domain-induced object. In the second case the function will also be handled as an object, but it will form part of the system infrastructure. In the third case the function cuts across objects and can be factored out from domain functionality to be an aspect. Which of the above mentioned cases applies to a specific function depends on the requirements for that function.

For example, consider the function access control. If access control is to be done only whenever a user attempts to enter a system and, once access has been granted, the user is free to use all functionalities, access control can be localised as an access control object which implements the required functionality. On the other hand, if access control should be more sophisticated and depend on user profiles and user groups that have certain rights at certain times, the functionality logically belongs to the application objects. A design may use access control lists and a state model enabling each object to decide whether access is to be granted. Access control could, then, be defined as an aspect of all the domain objects. An implementation could be split into a generic access control object which implements common functionality, and the domain objects which have to implement their specific access control functionality. The generic component would form part of the system infrastructure (see section 7.5.4). This example shows how the second and third cases defined above can be related.

**BBM Aspects and Aspect-Oriented Programming**

A related definition of an aspect is given by Kiczales et al. [KLM*97]. Independent of architectural discussions, limitations of object-oriented programming have been recognised. Examples are described where object-oriented modelling is too limited and leads to very complex code. An additional structuring is looked for, which leads to a natural design structure for those complex examples. Kiczales et al. call their approach aspect-oriented programming (AOP). They define an aspect to be functionality which crosscuts objects. An AOP-based program consists of a module containing the aspect source text per aspect. These source texts are automatically integrated by an aspect weaver into the normal object-oriented code. Source texts are not pol-
luted by code from other aspects and are claimed to be easier to maintain. However, automatic weaving of aspects relies on rules to be incorporated in the aspect code such as "before and/or after execution of command x do the following aspect statements". This anchoring of aspect code in the normal code introduces potential dependencies between different aspects. Problem-oriented (sub-)languages are designed in AOP for each aspect. This approach makes aspects very problem-specific. Since the concern of Kiczales et al. is programming and development of next generation programming languages, it could be said that their approach is a bottom-up approach while the BBM method is a top-down approach from the system point of view.

In contrast to AOP, aspects of the BBM are a complete partitioning of functionality. As described above, identification of aspects looks for crosscutting functionality to be factored out from the operational aspect. Analysis for different types of functionality is very important for large systems. A lot of functionality in large systems is not actually concerned with the application itself but with providing support for the application and with achieving the quality requirements for the overall system. In the tss system, for instance, only 20% of the code is application functionality. The rest is functionality like recovery functions to initialise and to bring the system in an operational state, database handling functions for persistent state information and error handling functions to detect error, isolate them and support a recovery of (possibly degraded) functionality.

Furthermore, aspects in the BBM are dealt with primarily on the design level and are not units of configuration like in AOP [CE00]. AOP and the BBM are complementary in this respect. Aspects as used in the BBM are standardised for the complete product family. Additional functionality often comes with new objects (see section 3.2.4). Introduction of an new aspect is a non-local change. In the BBM, common aspect implementations are factored out in system infrastructure generics (see section 7.5.4). For implementing BBM aspects see section 5.7.3.

**Steps of the Aspect Design Task**

We give now a list of steps done during aspect design:

Initially taking the complete functionality as one aspect.

*Heuristic 16: Take the complete functionality as the first aspect called operational aspect.*

Analysing domain-induced objects for crosscutting functionality

*Heuristic 17: Look for common behaviour of domain-induced objects. Allocate similar cross-cutting behaviour to one aspect.*
Performing an architectural concern analysis to find additional aspects (section 5.2).

Using starter sets [CE00] of potential aspects to support the aspect identification:

Previous design experience is reused to identify aspects. This often shortens the aspect analysis. Several sources for the creation of starter sets will be given in section 5.3.

Standardising the list of aspects for the product family:

This may lead to fewer aspects because some may be only relevant for a single or a few products. (The identified product-specific crosscutting functionality may be supported by the design of a generic BB during composability design.)(section 5.4)

Determining the functionality per aspect:

After the identification of aspects the precise functionality has to be determined. This may involve more functionality than originally analysed. For instance, error handling may consists of error localisation, notification, recovery and logging (section 5.5).

Making a global aspect design:

A unique design of an aspect increases conceptual integrity. The number of design concepts will be easier to limit with this global scope. This may lead to specific models like initialisation model and configuration management model.

Factoring out common implementation parts in system infrastructure generics.

(This is part of composability design.)

Defining rules and guidelines per aspect.

The following section describe the concepts and the design steps in detail.

---

5.2 Architectural Concern Analysis

System design is a multi-disciplinary task with many stakeholders. Besides having required application functionality, a system must provide this functionality
with the required qualities. In addition, various stakeholders have different concerns about the system, its operational environment and its development context. The architecture has to address these qualities and concerns which often lead to additional functionality.

High reliability may lead to specific recovery and error handling. High uptime may lead to on-the-fly HW and SW corrections, updates and extensions and portability may lead to abstraction layers.

The BBM uses an architectural concern analysis to identify additional functionality and in particular to identify new software aspects. However, functionality which does not lead to a new aspect may lead to new objects or to an extension of existing ones.

The architectural concern analysis consists of several steps and is presented in section 5.2.1. In section 5.2.2 we take examples of architectural concerns and analyse them for system aspects, the first step in the architectural concern analysis. In section 5.2.3 we describe how system qualities and available technology can induce software aspects.

5.2.1 The Analysis Steps

We use an architectural concern analysis to identify new aspects. The outcome, however, may also lead to new objects or extend existing aspects and objects. The architectural concern analysis takes examples of lists of architectural concerns which are compiled in various contexts to consolidate design experience. These lists are analysed to identify new software aspects for the product family which we design.

The analysis consists of five steps and is shown in figure 30. The analysis is presented now step by step.

In the next section we will present four lists of architectural concerns and analyse them for software aspects of a hypothetical system. The list of architectural concerns are taken from checklist or standards and represent design experience of certain domains.

The first step is to analyse the architectural concerns whether they influence the system implementation or only the context of system implementation. We call the architectural concerns which influence the system implementation system aspects.
Examples of architectural concerns which influence the context of implementation are cost structure and testing strategy, while performance and security are system aspects.

The second step is to analyse system aspects whether they directly specify system functionality or only put constraints on how system functionality is to be implemented. System aspects which specify functionality may be application specific or result from the operational environment.

Examples for system aspects which come from the operational environment are field service, field test and user guidance.

System aspects which only constrain the implementation of system functionality fall in two categories. Either they constrain the implementation from an outside perspective and are called system qualities or they constrain the implementation from an inside perspective and need to be expressed by design guidelines. System qualities can indirectly induce functionality (see section 5.2.3).

The third step is to analyse system aspects specifying functionality whether they specify functionality to be implemented in software or in hardware.

The fourth step is to analyse system aspects specifying software functionality whether this functionality cross-cuts domain-induced objects or is a domain-induced object itself.

The fifth step analyses functionality which cross-cuts domain-induced objects whether it is product specific functionality or stable for the complete product family (see section 5.4).

The architectural concern analysis leads to several possible outcomes (the leaves of the tree in figure 30). An architectural concern

- influences the context of system implementation, that is, the development organisation has to take it into account; these are out of the scope of the BBM; additional methods and strategies have to be employed; or
- constrains the system implementation from the outside, i.e. a system quality, or from the inside, i.e. design or implementation guidelines; system qualities are analysed in section 5.2.3; or
- specifies hardware functionality; these are out of the scope of the BBM; or
- specifies a domain-induced object; this is an input for refinement of object design or
architectural concerns

- influence the context of system implementation
- specify system functionality
  - system quality, guidelines (constrains system implementation)
  - hardware functionality
  - software functionality
    - domain-induced object
      - orthogonal to domain-induced objects
        - product specific
          - software aspect (stable for product family)

Figure 30: Architectural Concern Analysis

- specifies aspect functionality which is specific for a certain product; this is an input for refinement of object design and possibly composability design to analyse if the functionality can be supported by a component framework; or
- identifies a software aspect.

Note that a specific architectural concern does not always lead to the definition of an aspect.

5.2.2 Architectural Concern List Examples

In the following subsections four different collections of architectural concerns will be presented. These concerns are taken from different contexts and we use them to demonstrate a high-level architectural concern analysis. They can be used as a starting point for the identification of new software aspects for a particular product family. The first example is the popular distinction in functional and non-functional requirements. It is too context-dependent to be useful for deriving additional functionality. The second example is a list of quality attributes collected by people from the Software Engineering Institute. The third example is a list of system management functional areas taken from a telecommunication...
standard. The fourth example is a list architectural concerns collected from design experience in medical imaging systems.

These lists are not part of the BBM but are examples from design experience which we can take as input for analysing functionality of a specific product family for new aspects.

*Heuristic 18:* Use lists of architectural concerns from design of similar systems for analysing the required functionality for the identification of aspects.

**Functional and Non-Functional Requirements**

A distinction is often made between functional and non-functional requirements. A well-designed system is required to exhibit many more properties, besides its functional characteristics. Depending on the system to be built, system properties such as performance, safety, technology choices, testability, reuse, portability, use of standards, etc. may be among the customer requirements. In general, a customer may choose not to specify those requirements at all, to specify them only partially or to specify them fully. However, implicit system properties which are expected to be present in all systems of a certain class in a specific market segment have to be added to those explicitly specified. Furthermore, additional requirements may come from a development organisation for achieving internal benefits such as consistency with a product policy. Therefore, the distinction between functional and non-functional requirements is too context-dependent to be useful for direct use in finding software aspects. System aspects (see section 5.2.1) cover both functional and non-functional properties.

**Quality Attributes**

Quality attributes constitute a important view on a system. Bass et al. [BCK98] give a classification in four classes. They distinguish between business qualities, quality attributes discernable at run time, quality attributes not discernable at run time and intrinsic architecture qualities.

The following business qualities are mentioned: time to market, cost, projected lifetime of the system, targeted market, roll-out schedule and extensive use of legacy systems

Quality attributes discernable at runtime are, for instance, performance, security, availability, functionality and usability.

Quality attributes not discernable at runtime are, for instance, modifiability, portability, reusability, integrability and testability.

Finally, intrinsic architecture qualities are conceptual integrity, correctness and completeness and buildability.
These qualities are useful for guiding the process of architecting a system. The architectural concern analysis leads to the following results. Business qualities influence the context of system implementation only. Intrinsic architecture qualities are system aspects and constrain the system implementation through their guidance for internal architectural characteristics. Quality attributes, both those discernable at runtime and those not discernable at runtime, constrain the system implementation from an outside perspective and are system qualities. They will be further analysed in section 5.2.3.

**Operator-Oriented System Functionality**

In the field of telecommunication infrastructure systems, tasks and procedures of operators have been classified. Classification groups tasks and procedures so that different types of operators can be assigned to each class.

A traditional classification comprises operation, maintenance and administration tasks. It is often abbreviated as OMA.

FCAPS is the classification of the OSI system management functional areas (SMFAs) [X700]. The functions are divided into fault management, configuration management, accounting management, performance management and security management.

These operator-oriented function classifications influence system implementation directly and are system aspects. In the tss product family fault management, configuration management and performance management were software aspects. Accounting management and security management lead to domain-induced objects. Accounting management was implemented as a billing application and security management was implemented as a login and operator rights management application.

**Checklist of Architectural Concerns**

G. Muller made a checklist of architectural concerns [Mul98] for designing medical imaging systems. His checklist is relevant for designing other systems as well. Muller pointed out that system architects have to take all these concerns into consideration, i.e. know the specific requirements, communicate with the various stakeholders, assess the relative importance of the individual requirements, etc.

The architectural concerns mentioned in the list represent a wide variety of mostly technical views. They relate to the system (to be built), its development environment and its use environments. We mention the list because of its breadth of technical issues. It helps to design a system without bias to certain technical issues. We have separated the list into items that represent system aspects and those which are broader architectural concerns. The following system aspects are listed:

- application requirements,
- functional behaviour,
functional chain specifications (print, store, etc.),
information model: world standardisation, company standardisation, department standardisation, application specific,
image quality,
performance, throughput, response time,
typical load,
resource usage (CPU, memory, disk, network, etc.),
module design, process design, function allocation (method, file, component, package),
selection and use of mechanisms,
installation, configuration, customisation, etc.,
configuration management (technical and commercial),
safety, hazard analysis,
security,
interfacing to other applications,
factory and field testability.

Architectural concerns that are broader than system aspects are:
test strategy, harnesses, suites, regression,
re-use consequences, provisions, development process impact, organisational impact, business impact,
interoperability, other connected systems, selected partners, other vendors,
verification,
assessment of strong and weak aspects, road map for all views,
technology choices (software, hardware, computer, dedicated digital, make/buy),
system engineering (cables, cabinets, environment, etc.),
cost structure (material, production, initial, maintenance, installation),
logistics, purchasing (long lead items, vulnerability, second sourcing).

A complete architectural concern analysis for a medical imaging workstation would describe most of its design decisions.

These lists are used as a starting point for the identification of software aspects. The results of the architectural concern analysis depends on the product family for which the analysis is done.
5.2.3 System Qualities and Available Technology

Similar to the discussion above about the architectural concern analysis, software aspects can also be induced by system qualities and technology. As mentioned in section 3.4 there are no straightforward design methods for system qualities.

The characteristics of available technology play a crucial factor in designing a system having certain qualities. These characteristics determine if the simple use of a certain technology is sufficient or if a careful design using that technology is necessary. Design to achieve these qualities often leads to additional functionality, the selection of specific design mechanisms and an implementation masking the shortcomings of underlying technologies.

For instance high reliability may be realised through automatic recovery, diagnostics and error handling functionality. Early products of the tss family contained an error correction unit for memory access because memory technology was unreliable.

Supporting functionality is new system functionality and will be analysed according to the presented scheme (figure 30). The analysis identifies software functionality either as domain-induced objects or crosscutting to objects, and furthermore determines whether the functionality is a stable software aspect for the product family.

Summary

The functionality of software aspects is induced by the application functionality itself, by functionality for the operational context, by system qualities and by technology.

Examples of aspects are the operational aspect induced by the application functionality, a field service aspect and a user guidance aspect induced by the operational con-
text, an error handling aspect and a diagnostics aspect induced by system qualities, and a data replication aspect induced by distributed HW (see figure 31).

![Diagram of software aspects and system aspects]

Figure 31: Examples of SW Aspect Stimuli

The relation between system aspects and software aspects can be summarised as follows. A system aspect either

- has no relation to software aspects, e.g. it
- constrains a system implementation (system quality, guidelines), or
- is (partially) handled outside SW, e.g. by HW, or
- is realised by a functional unit, e.g. domain object, BB, subsystem, or
- is handled by a specific SW aspect, or
- is subsumed under another SW aspect, or
- is distributed between several other SW aspects and/or functional units.

Describing how system aspects are dealt with in a specific system and which software aspects are used, is a way to consolidate design experience (see section A.3.3.2 for an example).

### 5.3 Starter Sets for Aspect Identification

Additionally, SW aspects can be identified by using starter sets [CE00]. Starter sets are lists of aspects from other products. They represent design experience from other projects. The use of such experience may shorten the analysis for aspects described in the previous section. Starter sets of SW aspects and examples of specific aspect designs can serve as consolidated design experience.

*Heuristic 19: Use lists of aspects from other systems as starter sets for aspect identification.*
In the rest of the section we will give examples of such aspect lists which can be used as starter sets. Examples of specific aspect designs are described in section A.3.3.3 and section A.3.3.4.

Example: tss SW Aspects

We shall give the list of SW aspects of the tss system [Bau95] as an example. These aspects are derived from the requirements and are a consequence of the system architecture (see section 10.1.1 and appendix A):

- system management interfacing
- recovery
- configuration control
- data replication
- test handling
- error handling
- diagnostics
- performance observation
- debugging
- overload control
- operational

The aspects are described in more detail in section A.3.3.1. The tss software aspects make reference to the system management functional areas (see section 5.2.2). Note that the areas accounting management and security management are not aspects in tss. The reason is that accounting management and security management are realised as objects. Accounting management is implemented as a set of BBs in the logical resource management layer (see appendix A) and security management is implemented generically via login procedures and user profiles in the operation and maintenance terminals (section A.2.1).

Example: Aspects of the Intentional Programming System

Another example is the list of aspects of the intentional programming system as described in [CE00]. An intention is a set of programming language features. The intentional programming system allows to extend programming languages with extensions which are close to the semantics of the application domain, that is, the intentions of the application specialist. The intentional programming system allows the user to define intentions and provides generic support for certain kinds of functionality. These kinds of support functions are orthogonal to intentions and are SW aspects. They are:

- editing, that is a set of language features may have their own way of editing, e.g. with graphical or textual support;
display, that is a set of language features may have graphical, textual or mixed way of displaying;
translation, that is a set of language features may have their own way of translating it into the internal syntax graph representation. Similar
debugging,
code optimisations,
profiling,
testing, and
error reporting may be supported by each intention specifically.

The intentional programming system provides interfaces per aspect which may be used by the functions of an intention.

5.4 List of Aspects for the Product Family

Aspects are identified by analysing the functionality of a system. If not already done from the beginning, the results of this analysis have to be placed in the perspective of the product family. Some of the aspects may be important in some products only.

Heuristic 20: Select only those aspects which are relevant for the complete product family as SW aspects.

This may lead to fewer aspects because some may be only relevant for a single or a few products.

Heuristic 21: Support identified product-specific crosscutting functionality through the design of a generic BB during composability design.

The operational aspect has a specific character. It contains all functionality not factored out into other aspects. For example, domain functionality such as the handling of calls in a telecommunication switching system or the taking of images in a medical imaging system may be part of the operational aspect.
5.5 Designing Aspect Functionality

A major part of the aspect design is the design of the aspect functionality. The functionality of each aspect has to be defined. This may involve further refinement of functionality. For instance, error handling may consist of error localisation, notification, recovery and logging.

*Heuristic 22: Limit the number of different design concepts per aspect to increase conceptual integrity.*

A unique design for a complete aspect makes the aspect easy to understand. This leads to specific models like an initialisation model and a configuration management model. However, design trade-offs have to be made to achieve all relevant system qualities.

An example of a design of an aspect from tss is the recovery aspect. It consists of a common recovery model where all BBs can register 9 different types of initialisation methods. Each of the 6 types of recovery executes a subset of these methods in a predefined order. The design is given in more detail in section A.3.3.4. As a further example the tss configuration control model is described in section A.3.3.3.

[Ren97] describes a design for exception handling similar to the one used for the tss system. Eight patterns addressing different facets of the design form a pattern language.

Sometimes it is more appropriate to have a small number of specific designs instead of a single one. For instance, error handling for HW faults will be different from handling of communication failure. Areas of aspect functionality with similar characteristic should be identified and uniform design concepts should be used within each area.

*Heuristic 23: Weigh the smaller number of aspects with potentially different designs against a larger number of small aspects with a unique design.*

The goal for the introduction of aspects is a better overall design of the system functionality.

Common implementations of elements of aspect functionality is put into system infrastructure generics (SIG) (see section 7.5.4).

For instance, the implementation of the recovery aspect of tss is completely factored out into a SIG. The implementation of the error handling aspect of tss is only partially
factored out. The guideline for the error handling aspect states that failures should be analysed and faults be handled as local as possible. Appropriate recovery actions should be taken as soon as possible. Reporting should be done only for original faults. This functionality could not be factored out entirely because it involved to much local knowledge about the possible failures and faults. Only the reporting of faults was implemented by a SIG.

5.6 Aspects and Building Blocks

As we argued in section 3.3, we assume that most of the systems designed evolve mainly in the object dimension. Most of the BBs contain one or more complete domain-induced objects. Aspects can be used as a standard structuring for each BB.

Heuristic 24: Introduce a standard structuring for BBs by letting all aspects be present in each BB, even if some of the aspects are empty in a particular BB.

Figure 32 shows five BBs together with a standardised list of aspects within each BB. The dashed lines connect identical aspects.
Note that some of the aspects such as debugging may require functionality to be present in all the BBs. Functionality of other aspects such as error handling may be empty in BBs where no errors occur, for instance those which do not handle hardware or external interfaces.

Product families designed with the BBM are primarily decomposed into BBs. Since BBs often contain clusters of objects, aspects constitute a second-order design-time decomposition within the BBs. Introducing a new aspect into a product family will involve an almost maximum change effort because all domain-induced objects and their BBs are affected.

5.6.1 Aspect Completeness of Building Blocks

Ideally, application features are modelled so that they can be added to an installed system without changes (see section 8.3.1). The goal of the BBM is to implement systems from plugable BBs only. To achieve this property BBs have to be independent, that is, the insertion of a BB into a system must not necessitate changes to be made elsewhere in the system. Independence, however, is always relative to a given infrastructure. Aspects combined with a well-designed infrastructure are a means for achieving independence of BBs. We introduce the term aspect-complete for such an independence and define that a (set of) BB(s) is aspect-complete if it is responsible for allocating all of its required resources itself, and implements all aspect functionalities [Mül97]. Such a BB is a self-describing component (see section 7.8) because it contains descriptions for all it needs from the rest of the system.

5.7 Further Usage of Aspects

Besides for designing functionality, aspects can also be used for structuring of reviews, documentation and implementation.

5.7.1 Aspects and Reviews

The list of aspects can be used by the architects to check the functional completeness of the identified BBs. Functionality has to be specified for each of the aspects, such as the initialisation actions of BBs, the faults to handle and the configuration data.
Heuristic 25: Use the list of aspects for checking completeness during review sessions. Structure large review team by allocating aspects to specific reviewers.

5.7.2 Aspects and Documentation

In the BBM the notion of a BB is pervasive, that is, it is an entity of specification, design, implementation and deployment (see section 11.5). Each BB has its own documents.

Heuristic 26: Make a separate chapter per aspect in the BB documents.

Furthermore, the list of aspects may be used for completeness checking in review sessions of BB documentation (see above).

5.7.3 Aspects and Implementation

In the implementation each object method is characterised by a triple \(<\text{object, process, aspect}>\) in the design space, i.e. each method is part of an object, is driven by a process and is part of an aspect (see section 3.3). Making aspects standard structures of a BB (see section 5.6) leads to a uniform modularity. Files, programming language modules or naming conventions are means of implementing this modularity.

Heuristic 27: Structure the implementation of a BB according to aspects.

5.8 Aspects and the Whole

Aspects are a means for structuring functionality. Whereas stakeholder concerns and system qualities are multiple external views of a system’s functionality, aspects are types of functionality from multiple internal views. Aspects, stakeholder concerns and system qualities, are ways of capturing the whole of a system. A stakeholder will usually have a single-view approach to a system. Things which are not visible at the first levels of the single-view decomposition are details, with respect to that view, at lower levels of the decomposition hierarchy. The problem with this approach is that important system characteristics are described at different levels. This makes a system hard to understand.

A multi-view approach addresses the whole from different angles. In a multi-view approach, the important system issues are not hidden by a dominating view
but are addressed by their own view. Different views complement each other. In the end the whole can be tackled more easily than with a single-view approach. A single-view approach is like addressing all construction problems with a Swiss Army knife, and this requires a lot of effort.

Object-orientation in itself is such a single-view approach. Every application concept is an object or coupled to an object. The concurrency design and the different aspects as introduced in this thesis complement object-orientation to a multi-view approach.

Heuristics Overview

Heuristic 16: Take the complete functionality as the first aspect called operational aspect.

Heuristic 17: Look for common behaviour of domain-induced objects. Allocate similar cross-cutting behaviour to one aspect.

Heuristic 18: Use lists of architectural concerns from design of similar systems for analysing the required functionality for the identification of aspects.

Heuristic 19: Use lists of aspects from other systems as starter sets for aspect identification.

Heuristic 20: Select only those aspects which are relevant for the complete product family as SW aspects.
Heuristic 21: Support identified product-specific crosscutting functionality through the design of a generic BB during composability design.

Heuristic 22: Limit the number of different design concepts per aspect to increase conceptual integrity.

Heuristic 23: Weigh the smaller number of aspects with potentially different designs against a larger number of small aspects with a unique design.

Heuristic 24: Introduce a standard structuring for BBs by letting all aspects be present in each BB, even if some of the aspects are empty in a particular BB.

Heuristic 25: Use the list of aspects for checking completeness during review sessions. Structure large review team by allocating aspects to specific reviewers.

Heuristic 26: Make a separate chapter per aspect in the BB documents.

Heuristic 27: Structure the implementation of a BB according to aspects.