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

Citation for published version (APA):
9 Comparison With Other Methods

To compare the BBM with other design methods and approaches we take a look at their architectural meta-models. The architectural meta-model is the underlying model of architectural design methods (see section 2.3). Often, design methods have no built-in notions of architecture. However, each design method induces a specific kind of architecture. SDL, for example, induces systems that consist of asynchronously communicating state machines (see section 9.2.3). We shall call those elements of an architecture which are required or induced by a method the architectural meta-model (AMM) of that method.

In this chapter we shall first summarise the architectural meta-model of the BBM and then compare it with the architectural meta-model of other design methods and approaches.

9.1 The Architectural Meta-Model of the BBM

The architectural meta-model of the BBM consists of

- a domain object model,
- a product feature dependency model,
- the Building Block design dimensions, that is the object model, aspects and the concurrency model,
- the Building Block dependency model, and
- the deployment model.

Note that the first two model are created as part of the architecting context of the BBM.
The domain object model is created as part of the application domain modelling task (see section 2.6.2) which is not part of the BBM per se, but the BBM requires it as a necessary input. It consists of the domain objects, their behaviour and relations.

The product feature dependency model is the result of the commercial product design task (see section 2.6.3) and not part of the BBM per se, but the BBM requires it as input. The product feature dependency model consists of product features and their dependency relation. It is described in section 8.1.

---

### 9.2 Traditional Development Methods

In the following we shall take a look at traditional software development methods and approaches. They have been selected for their historical importance. These methods and approaches shall be examined with respect to their support for software architecture. Each of these methods has either an implicit or an explicit architectural meta-model. A more general comparison is given in [Wie98b].

#### 9.2.1 Structured Design

Structured Design [YC79] brings structure to the task of designing a system by analysing system functionality in a top-down manner. It uses data flow analysis to analyse the flow of data from input to output. A data flow diagram models the functionality of a system in terms of data-transforming functions. So-called Transform and Transaction Analysis is used to map these data flows to a hierarchy of functional modules. Modules at the leaves of the hierarchy, ideally, have the task of performing simple computations, while modules in the hierarchy have the task of controlling and coordinating the flow of data from one leave module to the next. The top of the hierarchy is called executive module. The model for this module structure is the hierarchical organisation as it is often found in human organisations. Structured Design introduces cohesion within modules and coupling between modules as measures for the quality of a design.

Constantine [Con80] sees function-oriented structuring, such as Structured Design, and object-oriented structuring as two extremes. He concludes that which of the two will be used will (usually) depend on the nature of the problem.

We describe the architectural meta-model of structured design as consisting of a hierarchy of functional modules. Structured design does not provide other
modelling means or views. Exceptional cases such as faults are not really handled in structured design. The provided architectural support is minimal.

9.2.2 Bare Operating Systems and Real-Time Kernels

A simple approach is often used in the domain of real-time systems where the modelling of dynamic behaviour has always received the most attention. As a system’s performance is of critical value to its function, system design focused on structuring of execution entities, their prioritising and execution time prediction. Operating systems and real-time kernels provide good support for this dynamic structure.

However, now that systems are becoming larger, reuse of existing components and system evolution support are becoming increasingly important. Moreover, the nature of these systems is unfortunately rarely such that the chosen thread and/or process structures match with reusable parts. The architectural metamodel consists of communicating processes and enclosed threads. The exclusive usage of threads and processes results in a lack of structuring means which should be overcome by additional structures.

9.2.3 SDL

Specification and Description Language (SDL) is a system description language based on state machines. A state machine is also called a process. State machines communicate via the exchange of asynchronous messages. State machines may be grouped into functional blocks. SDL-92 [FO94] is an object-oriented extension of SDL. It adds a distinction between types and instances, specialisation of types into subtypes, and the concept of generic types. A system instance consists of a network of connected peer block instances. These block instances may be composed either from other block instances or from process instances. Process instances build a communicating network inside the connected blocks.

The architectural metamodel consists of communicating statemachines. SDL processes serve as both structural entities and behavioural entities. This is the reason for the intuitive simplicity of SDL. However, it prevents the design of optimised structures independently for structure and behaviour. The structure becomes artificial for algorithmic and data-oriented parts of an application which are not statemachines. In such cases SDL hides the real complexities in transition procedures.
9.2.4 ROOM

The Real-time Object-Oriented Method [SGM*92] is based on an SDL-style of design. It shares the identity of structural and behavioural modelling with SDL. Components are called actors and are functional blocks. An actor is both a static and a dynamic entity. It communicates with other actors via message-based communication. Actors share no memory. Actors are arranged in layers. Layers have interfaces called service access points. An actor communicates either to another actor in the same layer or to another layer via the service access points. Procedural libraries can only be used inside an actor.

The architectural metamodel consists of communicating statemachines which can be placed in layers. ROOM adds layers and provides the possibility for more advanced structuring. However, there is still limited modelling flexibility because there is no distinction between modularisation and execution entities.

9.2.5 OMT

The Object Modelling Technique (OMT) [RBP*91] is probably the most used object-oriented design method. Its main intention is to closely couple problem analysis and system design. Object modelling is used as a vehicle which provides a conceptual continuum from problem analysis to implementation. The analysis phase not only analyses the requirements, but also builds an object model of the system to be built. OMT uses three models:

- the object model itself, which describes classes and associations between classes,
- the dynamic model, which describes state transitions in classes and global event flow between classes,
- the functional model, which describes data flow and functional dependencies between classes.

These models are used during the development phases. The OMT process identifies the three phases analysis, system design and object design.

The analysis phase is concerned with understanding and modelling both the application and the domain within which it operates. Analysis takes the problem statement as initial input. This input is enriched with knowledge about the operational environment and the intended usages. On the basis of these inputs an analysis model of the functionality of the system is built, consisting of the object model, the dynamic model and the functional model.
The overall architecture of the system is determined during system design. On the basis of the object model, the system is structured into subsystems, classes are grouped into concurrent tasks, and further decisions about inter-process communication, data storage, and priorities for design trade-offs are taken. The chapter on system design in [RBP*91] describes several system design concepts (see table 4).

<table>
<thead>
<tr>
<th>Architectural styles such as horizontal layers, vertical partitions and pipeline- and star-like system topologies can be used to structure subsystems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the overall control architecture, three control styles are given to handle externally visible events: procedure-driven sequential, event-driven sequential and concurrent.</td>
</tr>
<tr>
<td>Internal control (within a process) can be purely procedural, quasi-concurrent call-back scheduling or concurrent threads.</td>
</tr>
<tr>
<td>So-called boundary conditions give a functionality classification in normal operation, initialisation, termination and failure.</td>
</tr>
<tr>
<td>Common architectural frameworks for describing classes of systems are: batch transformation, continuous transformation, interactive interface, dynamic simulation, real-time system and transaction manager.</td>
</tr>
</tbody>
</table>

Table 4: System Design Concepts of OMT

During the object design phase, the analysis models are enriched with detail using the dynamic model and the functional model.

We describe the architectural meta-model of OMT as being based on the object model. The dynamic model and the functional model provide other views which refine facets of the object model. During the system design phase classes are grouped into subsystems and into concurrent tasks. The object design phase is again based on the object model. It uses the dynamic model and the functional model to further design the classes and their relations. The architectural concepts of the system design phase are not really integrated into the method. Instead of using the architectural concepts for designing a good system architecture, priority is given to the seamless transition from the analysis models to the object design phase. We conclude, therefore, that the architecture of an OMT-based system consists of a network of classes grouped into subsystems and tasks. Subsystems and tasks are more like annotations to the object model than architectural concepts in their own right. However, this is less of a problem for small non-real-time systems as the authors of [RBP*91] characterise their focus (p. 198, p. 169). Large SW-intensive systems require more explicit architectural modelling for
which the concepts listed in table 4 can be used but OMT does not give any help. However, an application domain model (see section 2.6.2) may be built with OMT.

9.2.6 Object-Oriented Software Engineering

We shall now take a look at Object-Oriented Software Engineering (OOSE) [JCJ*92] as a further representative of object-oriented methods. OOSE works with five different models:

- a requirements model for capturing the requirements,
- an analysis model for giving the system a robust object structure,
- a design model for adapting and refining the object structure to the implementation environment,
- an implementation model consisting of the code, and
- a testing model for verifying the system.

These models are the output of three development phases. The requirements model and the analysis model are the products of the analysis phase. The design model and the implementation model are the products of the construction phase. In the testing phase the test model is produced and the system is tested. The transition from objects in one model to objects in another model is seamless, that is, the identity of objects does not change during transitions.

OOSE defines the requirements model as consisting of a use-case model, interface descriptions and a problem domain model. The use-case model is the most important of all the models. It has a central position for building all other models. The use-case model is expressed in terms of the objects from the domain. The analysis model is structured as an implementation-environment-independent object model derived from the use cases.

The analysis model does not directly use the domain objects from the domain object model. Instead, it derives from use cases three types of objects: entity objects, interface objects and control objects. [JCJ*92] claims that under changing requirements this object structure will be more stable than a standard object model, that is, changes will be local to hopefully a single object. Entity objects model information which is most stable. Entities are often derived from domain objects. Interface objects model information and behaviour that is dependent on the system’s external interfaces. Control objects model functionality which is not captured by the other two object types. They represent the coordination between
entity objects, and between entity objects and interface objects. Often, one use case will result in one control object.

As the focus of OOSE is on analysis, the design is a refinement of the analysis. The concept of a block is introduced to capture the design of an object. There may be interface blocks, entity blocks and control blocks. The notion of a subsystem is introduced to group blocks. Subsystems may contain other subsystems recursively which at the lowest level contain blocks. However, a designer may deviate from the object structure if necessary. Also, a mapping to threads and processes may change the design model. OOSE makes a distinction between application modules, which are called blocks, and components, which are essentially infrastructure libraries. Reuse is discussed only for these library components.

In modelling real-time systems, OOSE attaches real-time requirements to use cases. Behaviour of use cases is mapped onto individual concurrent processes and threads. Threads are seen as orthogonal to objects.

We describe the architectural meta-model of OOSE as based on the three objects types. An OOSE-based system consists of a network of entity, interface and control objects. Block design groups some of the classes and provides interfaces. So-called components provide reusable infrastructure libraries. Threads and processes are used to design the real-time dimension. The emphasis of OOSE is on the process steps which lead to a system. OOSE uses the term architecture on a meta-level to denote the structure of its consecutive models.

The centrality of the use-case model, which is a view from outside the system, is the cause of the lack of emphasis on system internal structuring. The text mentions some exceptions where internal considerations overrule the use-case structure, but they are not really integrated in the method. The design of an architecture suffers from the focus on seamless transition of models in different development phases.

9.2.7 Comparison with Traditional Development Methods

As described in the previous sections, traditional development methods lack the necessary structures for developing large software-intensive structures. An analysis of object-oriented systems led to the observation of the tyranny of the dominant decomposition [TOH99]. Object-oriented methods imply that the world consists of objects only. However, different design needs require different modularities. The BBM, because it originates from the development of large software-intensive systems, provides a richer set of structuring means.
9.3 Architectural Approaches

We shall now take a look at three other approaches to SW architecture. The first, architectural styles, is a single-view approach. The other two have in common with the BBM that they move away from an overall transformational approach to SW development in which the architecture is an intermediate result obtained on the way to an implementation. SW architecture is represented by different views which are not ordered via temporal relations. These views are maintained and evolved independently of the component level derived from them. They remain valid system descriptions during the active development of the system.

9.3.1 Architectural Styles

An architectural style is the dominant structural pattern of a system. Such patterns can be used as a single-view architectural approaches. The design elements of an architectural style are often made visible even in the application domain model. These styles characterise systems in such a way that even customers are made knowledgable about the presence of a particular style.

Several architectural styles are to be found in [SG96], [RBP*91] and [BMR*96]. Examples are

pipes and filters

The pipes and filters architectural style provides a structure for systems that process streams of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters.

blackboard

The blackboard architectural style decomposes a system into three types of components. Knowledge sources are components designed for a specific task. A blackboard can store data that is used to communicate between knowledge sources. A vocabulary describes the data formats which the blackboard is allowed to use. A control component coordinates the knowledge sources at the blackboard.

layers

The layer architectural style decomposes a system into a group of units in which each group works at a particular level of abstraction. A unit makes use
of services of lower layers and provides services to higher layers. Layering has a prominent role in the BBM (see section 7.4).

[JB95] mention the Linda-related [CG89] style SPLICE

The SPLICE architectural style [Boa93b] is a refinement of the blackboard architectural style and relies on a shared data space. For distributed applications, the shared data space is built on top of a SW bus. Applications are then connected to the data space. Data classes are broadcasted on the bus. Applications which are interested in a specific data class subscribe to that data class. The instances of the data class are forwarded to a receiving area of these applications. Applications can poll the area for new data or be notified on arrival. The possibility of buffering decouples update speed of data and reading speed of applications. Measurement data allows for single element buffers in which a new value overwrites the old one. This is an important point for real-time data applications. SPLICE makes processes and components identical.

Architectural styles constitute single-view architectural models [Ben97]. All views coincide in one overall view. This makes systems easy to understand but also limits the flexibility of architectural design. Their adequacy depends on the kind of system under design.

They are a first step towards explicit architectural modelling. They should be considered part of a system architect's handbook. A system architect may use them together with other architectural and design patterns, and with an architectural method (see section 3.5.2).

9.3.2 Soni

Through the analysis of 15 systems at Siemens, Soni et al. [SNH95] came to recognise certain structures explicitly or implicitly available in all of the systems.
Figure 56 gives an overview of the different views. A conceptual architecture describes the system at the highest level of abstraction. It contains a decomposition of the system into its main components and the connectors relating them. A module architecture describes the static partition of the software into modules and the dependency relation between modules. A code architecture describes files, directories and libraries which are used by the module architecture. The execution architecture describes resources of the operating system used for execution and communication, and the assignment of elements from the afore-mentioned architectures to them. Execution elements reside on a hardware architecture. The hardware architecture describes processing nodes and network connections.

The conceptual architecture is comparable with the domain model used by the BBM. However, a domain model is totally in terms of externally observable behaviour, and does not contain a high-level partition of the system. The module architecture is comparable with the object dimension and the BBs. The execution architecture is comparable with the thread dimension. The hardware architecture is comparable with the deployment model of the BBM.
9.3.3 4+1 Model

Kruchten [Kru95] presents a model which uses four views (figure 57) to describe a software architecture. A logical view describes the system’s functionality in terms which the customers and end users can understand. The development view describes the system’s development units. The process view describes the use of execution units. The physical view describes the allocation and allocatability to hardware instances. Scenarios are used as methodical means for specifying functionality within and between the view descriptions. Kruchten explicitly assigns the views to certain stakeholders.

![Diagram of 4+1 Architectural Model]

An example of a model derived from Kruchten’s model is described in [MHM98]. It uses the four views object view, layered view, task view and scenarios.

9.3.4 Comparison with Soni and 4+1

In this section we shall compare the model developed by Soni [SNH95] and Kruchten [Kru95] with the BBM. The architectural models of Kruchten and Soni et al. make a distinction between object and thread dimension for their implementation structuring. Kruchten uses the terms development view and process view, while Soni et al. use module interconnection architecture and execution architecture. The examples given by Soni et al. indicate that the conceptual architecture provides a functional decomposition which is also the top-level
Comparison With Other Methods

structure for the module interconnection architecture and the execution architecture. Kruchten's logical view provides no constraints for the further structuring in the development and the process view. The BBM by relying on the application domain model uses a similar approach to that proposed by Kruchten.

Kruchten’s model is object-oriented and recognises the independence of the modelling of processing resources from development units. It has no aspect dimension, that is, there is no support for functional structuring. Soni et al. work with a functional structuring in the conceptual architecture and distinguish between development units and processes only on the next level. The functional structuring is dominant, object-oriented structuring may be used on a micro-level. Perhaps this is the case because their model is more a reverse-architecting model than an architecting model.

Neither Kruchten’s nor Soni’s model work with components. Kruchten’s development units and Soni’s modules are traditional decomposition structures. No separate modelling for flexible integration and composition is addressed. However, such modelling would be a quite natural extension to their approaches.

The code view is only explicitly present Soni’s model. The BBM assumes that the code is structured per BB. Because that is not the focus of the BBM, no additional support is provided. If necessary, a project may add a separate model to describe an independent code view.

The application domain model used by the BBM is close to the logical model developed by Kruchten [Kru95]. The BBM additionally is based on a product feature dependency model resulting from commercial product design. This is an important input because it presents the commercial perspective on the system. Evolution of a system will be via new or updated features.

Table 5 list the models of the three compared approaches.
## Comparison of Architectural Meta-Models

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Soni</th>
<th>Kruchten (4+1)</th>
<th>BBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Model</td>
<td>Conceptual Architecture</td>
<td>Logical View</td>
<td>Application Domain</td>
</tr>
<tr>
<td>Feature Model</td>
<td>-</td>
<td>-</td>
<td>Product Feature</td>
</tr>
<tr>
<td>Object Model</td>
<td>-</td>
<td>part of Development</td>
<td>Object</td>
</tr>
<tr>
<td>Functional Model</td>
<td>part of Conceptual</td>
<td>-</td>
<td>Aspect</td>
</tr>
<tr>
<td>Process Model</td>
<td>Execution Architecture</td>
<td>Process View</td>
<td>Thread</td>
</tr>
<tr>
<td>Development Unit Model</td>
<td>Module Architecture</td>
<td>Development View</td>
<td>BB Dependency</td>
</tr>
<tr>
<td>Distribution Model</td>
<td>Hardware Architecture</td>
<td>Physical View</td>
<td>Deployment</td>
</tr>
<tr>
<td>Code Model</td>
<td>Code Architecture</td>
<td>part of Development</td>
<td>part of BB</td>
</tr>
</tbody>
</table>

We can conclude that the BBM has more comprehensive means for structuring the architecture. Especially the product feature model provides important guidance for the structuring in BBs. This is important because a product family architecture needs to support the implementation of the required feature in each of the products.
Comparison With Other Methods