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

Citation for published version (APA):
The goal of the BBM is to support the architecting of product families. This chapter describes how the BBM concepts are used to achieve this goal. The first section introduces commercial product features in the scope of product diversity. The second section summarises means for implementing diversity. The third section discusses the concept of a product family architecture. The last section places the concept of a product family architecture in the context of managing the evolution of products.

8.1 Product Diversity and Features

Nowadays, electronic products usually are developed as sets of related products aimed at covering a segment of a specific market. The commonality between these products is complemented by diverse parts, which makes them different.

In describing a product and its advantages to a prospective customer, one usually refers to its features. A feature is a characteristic capability or attribute of a product which is believed to be of importance to the customer, or alternatively important for distinguishing it from a competitive product.

A set of related products which share a common set of features is also called a product line or product family. The development of a product family is a strategic decision for a specific market. We shall not discuss the business and technical prerequisites which justify the development of a product family (see [MSG96], [DKO*97] and [ML97]).

Product diversity is a term broader than what is covered by features. For example, if the only difference between two cars were to be their colour, one would not speak about different features of the cars. However, what a product precisely is, and what degree of difference is necessary to speak about a different product will depend on the market. It is not a technical question. Diversity can
exist between different products as well as between different instances of one product. We shall use the term feature to define functionality of a product and not to distinguish different instances of a product.

Note that the creation of feature descriptions and their dependencies is not part of the BBM but is done as part of the commercial product design (section 2.6.3). The BBM uses the feature descriptions and feature dependencies as input for the creation of a product family architecture.

8.1.1 Feature Description

A feature has been loosely defined above as a capability of a product. The terminology is borrowed from the application domain model (see section 2.6.2). The domain model describes functionality not only of one product, but of a whole range of possible products. The precise form of a feature description therefore depends on the domain model. If the domain model is complete, a feature description can be restricted to references to the domain model. Otherwise feature descriptions complement descriptions of the domain model.

8.1.2 Feature Relations

Features are not described in isolation. Features will usually make reference to related features. A feature relation graph describes relations between features. The relations are:

*Dependence:* One feature may be dependent on the presence of another feature. Application features are usually dependent on infrastructure features. But within an application too, features may rely on other features. A special case of dependency is when two features are mutually dependent.

*Exclusion:* The exclusion relation describes features which may not be present together in one product.

*Selection from a set:* A number of features may form a set, only a certain maximum of which is allowed to be present in a product.

Further relations to indicate business considerations may be defined to aid the development people in making the right design decisions. This is not worked out any further.

*Heuristic 80:* Use a feature relation graph to describe relations between features.
8.1.3 Feature List

A collection of features is called a feature list. A feature list is used as a high-level description of a product. The BBM uses this approach to define products (see section 8.3).

The feature list can only describe an entire product if it is complete. A feature list is complete if all the required features are either directly mentioned in the feature list or are implied because of the feature relation.

Example: Typical tss Customer Features

As an example we describe typical features of the tss family. Different tss family members differ in type and number of the peripheral cards, different types of subscribers, service functions and network connections, etc. Additionally, products differ in kind of signalling handlers and call facilities (call forwarding on busy, automatic call answering if absent). After system delivery a customer may ask for more concentrated subscriber cards, additional service functions, updated signalling handlers when the signalling standard was updated, and/or additional call facilities. This illustrates that a product may evolve into a different one after delivery to the customer.

Product Features and Feature Modelling

Features as used by the BBM are different from features used for feature modelling. Feature modelling [CE00] is a variant of conceptual modelling and used throughout all development phases to describe domain and system functionality. Features are used as alternatives to objects in creating a domain model. The notion of an object as consisting of state and behaviour is artificial for describing, for instance, a table. Feature modelling, in contrast, claims to make more natural descriptions by just describing necessary and optional features of a table. Feature modelling concepts are described in feature diagrams consisting of a hierarchy of features and subfeatures.

A feature as used by the BBM is an independently sellable unit of functionality, that is, a feature is a unit from a commercial point of view. It is an outcome of the commercial design activity. Features have a dependency relation with the meaning that if a particular feature is needed also features it depends on are needed. In the BBM, features are not described in hierarchies. A product is described as a set of mandatory and optional features. The mandatory set determines the distinguishing characteristic of a particular product while the optional features allow for variation. Besides the dependence between features, the implementation of the features may introduce additional dependencies between BBs which results in certain features being a must with certain products. Furthermore, late changes in the feature relations by product management may lead to an implementation which restricts product configurability beyond feature dependence.
An example of this was the request for low-end switching systems without HW redundancy. This request could not be met without major redesigns. The problem was not so much the SW where visibility of redundant HW was easily localised but the design of the HW itself.

8.2 Implementation Means for Diversity

The BBM supports the creation of a product family on the architectural level. This means that BBs are designed so that each product can be configured from a list of BBs. Several concepts of the BBM are used to model architectural diversity. Layering of BBs is used to limit the effects of changes; for example, hardware-related changes are separated from application changes (see section 7.4.1). Generic BBs implement common functionality with variation points for specific BBs. Specific BBs connect themselves at variation points [DW99] to a framework via call-backs (see section 7.5.1). Feature orientation, to be described below, relates features of a product to BBs [MHM98]. Product variability is obtained through selection of BBs. From a product family point of view, the diversity between different products is most important.

Diversity in a product family can also be implemented in a non-architectural manner. In such a case we speak of a product family without a product family architecture.

Examples of non-architectural means for product diversity are to be found in four different areas:

Control-flow-driven diversity concepts of programming languages for changing a program’s control flow such as IF or CASE statements.
Data-driven diversity such as property values, i.e. the diversity occurs via some initialised constant or dynamic value; configuration database, i.e. a systematic implementation of property values; and configuration bits, i.e. a bit list represents different functions which are usable or not.
Development-environment-supported diversity, such as conditional compilation and conditional linking.
Customer-enabled diversity is a means for customisation such as setting specific initial or default values, enabling or disabling certain functions. It is a variant of data-driven diversity.
Relying only on non-architectural diversity for implementing a product family leads to a situation where the product family becomes a monolith. Maintenance for specific products always affects all products.

The tss system uses, besides programming language constructs, a configuration database to store data such as the hardware configuration, the logical function configuration, the subscriber and network configuration, and various other parameter values. Each table of the database belongs to a BB. A BB may have zero, one or more tables. These configuration data may be different per product installation.

The tss system does not use conditional compilation because conditions are difficult to maintain and the programs are difficult to understand. Configuration bits are not used either, because the products are configured to a minimum, i.e. unnecessary functionality is omitted.

*Heuristic 81: Design a system infrastructure generic to handle data-driven diversity.*

---

### 8.3 Product Family Architecture

A product family architecture is an architecture which covers a product family through architectural diversity. A product is obtained through selection of software components.

#### 8.3.1 Feature Orientation

A product family is not developed in one step. Evolution and extension are the most important development tasks after the initial development of the family. A family’s architecture is one of its critical success factors. A good architecture should allow to develop and update independently those parts which are independently present in products. Those independent parts are encapsulated in different BBs [Par76]. The BBs obtained in this way are called feature BBs. That is, the identification of BBs is based on features [MHM98]. The relation between customer features and feature BBs should be simple.

A feature that is sold to a customer does not necessarily have a direct relation with SW, however, many will require SW functionality. Only those that require SW functionality are considered here.

Feature orientation means that a system’s architecture is used primarily to support the system’s evolution. Evolution and extension which can be limited to one
or a few BBs are less costly to develop than those which are spread over large parts of the system. In the most simple case a change is realised by exchanging a BB, and an extension by introducing an additional BB; in [Ben97] this is called "hard component software". Such an approach requires a good understanding of the nature of a system and its evolution. Often this understanding will itself evolve. Feedback from first products improves the development of the product family.

Feature orientation requires a good, stable infrastructure and a BB granularity which matches features. Feature orientation may be difficult to realise in an initial version of a product. However, as a system and its functionality are better understood, it becomes easier to take evolution into account.

### 8.3.2 Family Patterns

![Figure 51: Basic Pattern for Diversity](image)

The basic pattern for diversity used to build a family architecture is a combination of a generic BB with its specific BBs (see figure 51). Specific BBs can register with the generic BB if they obey the abstractions provided by the generic BB.
The family architecture is based on the architectural skeleton introduced in section 7.8. The family architecture shown in figure 52 can be built from the basic diversity pattern and layer access generics which handle the access to the next layer above (see section 7.5.4). Layer access generics make it possible to configure BBs in lower layers, because specific BBs have no direct relations to higher layers.

8.3.3 Extensibility

Extensibility of systems is one of the major challenges during the development of a product family. Extensions may be conservative or non-conservative. Conservative extension does not change BBs in the system. Non-conservative extension extends a system by affecting existing parts.
New features are easiest to handle if they can be developed and incorporated through new components only. Existing systems are not affected and no new versions of components already deployed need be developed. The BBM supports this through the development of generic components in different layers of the architecture. The restriction to uni-directional dependencies clearly limits the knowledge of BBs of their environment. Higher-layer BBs can be substituted without the need to make changes in a lower-layer BB [Fra97]. The skeleton of generics (see section 7.8) is extensible in all layers. Generics are the mechanism for achieving conservative extensibility.

There are many different forms of non-conservative extension. Extended versions of existing BBs are the easiest. The most difficult extensions are those which require changes to the architectural skeleton, that is, changes have to be made in many generics. In such a case the system must be updated almost in its entirety.

Heuristic 82: A desirable non-conservative extension is the refactoring of a BB to a generic BB.

Forcing extensions into new BBs only, without updating existing BBs, degrades the quality of the architecture and limits the possibility of realising further extensions. In such a case parts of the system will be redundant in slightly different versions. Short-term considerations and long-term architectural conceptual integrity have to be carefully weighted, since the architecture is the basis for a long-lived product family. Compromise will eventually destroy its integrity.

Refactoring BBs into generic and specific ones is a way to remove duplicate implementations. However, a stable generic is rarely developed in one step. Feedback from implementation is necessary. If variation is only conceptually known, designing stable interfaces between a generic BB and its specific BBs is difficult.

Besides the ability to extend a system with new functionality, it is also important to be able to reduce it. It should be possible to remove functionality which is not needed or outdated from a system. The minimisation of functionality through BB configuration under the constraint that all the required features are supported is called configuration to minimum. Configuration to minimum is a desirable characteristic for a system to support the management of evolution [Par79]. It means that each product contains exactly the BBs which are necessary for it, and no more.
8.3.4 Feature Mapping

Heuristic 83: A good family architecture is one whose BB structure resembles the feature structure, i.e. a good family architecture is feature-oriented.

Indeed, a feature-oriented architecture localises foreseeable evolution and extension in small sets of BBs. Development, then, takes place as a development of increments to a stable base.

Product managers and architects are jointly responsible for a product family’s features. Product managers translate customer wishes into features. Architects contribute features from a technical perspective. Any other stakeholder may also suggest features (see section 11.1).

Product features and their relations are an input for architects. To achieve feature orientation, the architects model the system so that a feature is realised by a small set of BBs. Dependent features may result in dependent BBs. Mutually dependent features can, if they are small enough, be implemented in common BBs. Features, that are not necessarily present together in a product, should be implemented in separate BBs. We define, therefore, the following relations (see figure 53) for the set $F$ of features $f$ and the set $B$ of Building Blocks $bb$:

- the feature dependency relation ($FD$):
  
  \[ FD := \{ (f_1, f_2) \in F \times F \mid f_1 \text{ depends on feature } f_2 \}; \]

  often the dependency relation contains only direct dependencies. $FD$ can be build by the transitive closure over the direct dependency relation;

- the feature exclusion relation ($FE$):
  
  \[ FE := \{ (f_1, f_2) \in F \times F \mid f_1 \text{ and } f_2 \text{ are not both part of the same product} \}; \]

- the BB dependency relation ($BBD$):
  
  \[ BBD := \{ (bb_1, bb_2) \in B \times B \mid bb_1 \text{ depends on } bb_2 \} \]

  note, that the $BBD$ is a partial order (reflexive, transitive, anti-symmetric) as defined by the BBM and, therefore, $BBD$ is transitively closed;

- the feature implementation relation ($FI$):
A feature $f$ may be directly implemented by a set of BBs, that is, each of the BBs of this set partially implements $f$.

From the above defined relations we can construct the relation $FID$ of independent features:

$$FID := \{ (f_a,f_b) \in F \times F \mid (f_a,f_b) \not\in FD \} ,$$

$FID$ is the complement of $FD$

the set of feature a particular feature $f$ depends on:

$$FD(f) := \{ f_2 \in F \mid (f,f_2) \in FD \}$$

the set of directly implementing feature BBs of a certain feature $f$:

$$FBB(f) := \{ bb \in B \mid (f,bb) \in FI \}$$

the set of BBs a particular BB $bb$ depends on:

$$BBD(bb) := \{ bb_2 \in B \mid (bb,bb_2) \in BBD \}$$

and the set of completely implementing BBs of a certain feature $f$:

$$CIBB(f) := \{ bb \in B \mid bb \in FBB(f) \lor \exists bb_1 \in FBB(f) : bb \in BBD(bb_1) \}$$

Starting from one feature one can construct a set $FD(f)$ consisting of the feature and all its dependent features. Following the implementation relation for all elements of $FD(f)$ leads to the respective feature BBs

$$FDBB(f) := \{ bb \in B \mid \exists f_2 \in FD(f) \land bb \in FBB(f_2) \}.$$  

These BBs have a dependency relation to dependent BBs. Thus, starting from one feature one can obtain a consistent partial system that implements that feature.
PS(f) := \{bb \in B \mid f_2 \in FD(f) \land bb \in CIBB(f_2)\}.

By using a feature list of a product one obtains a complete collection of BBs which realise this product.

Figures 52 and 53 can be combined to obtain figure 54. Four application features are shown in their mapping to specific BBs.

![Diagram](image_url)

*Figure 54: Application Feature Implementation Relation*

The relations, described above, are used as a basis for assessing the quality of a family architecture. A good family architecture is one in which independent features can evolve independently. In such a case feature adaptation leads to adaptations in Fl-related BBs only.

Excluding features are implemented by different BBs, if and only if

\[ \forall(f_1, f_2) \in FE \rightarrow \{bb \mid (f_1, bb) \in Fl\} \cap \{bb \mid (f_2, bb) \in Fl\} = \emptyset \]

Independent features are independently configurable, if and only if
\( \forall (f_1, f_2) \in FID \rightarrow [ (FBB(f_1) \cap CIBB(f_2)) = \emptyset \land \\
(CIBB(f_1) \cap FBB(f_2)) = \emptyset ] \)

In a good architecture, BB dependency relations follow the feature dependencies FD.

A product family architecture which is feature-oriented allows configuration of a product with minimal functionality consisting only of the functionality of its required features. We call this configuration to minimum.

8.3.5 Example: tss Feature BBs

We shall give some typical examples of features of the tss product family.

**Card Maintenance BBs**

BBs in the EM layer (section ) which are responsible for the maintenance and supervision of the peripheral hardware cards are structured so that each peripheral hardware card type has a corresponding maintenance and supervision BB. Changes in the PCs lead only to changes in the corresponding maintenance and supervision BBs (figure 48). Since a large part of the functionality for maintenance and supervision has been standardised for all the peripheral cards, a separate generic BB, PC-G, implements this standard functionality (Figure 55). During the design a careful analysis has been performed to determine which SW parts can be standardised and which must be specific per PC. A customer feature of a new PC type leads as a result of HW mirroring to an extra PC-specific BB in the central controller SW. Figure 55 shows the PC-G with five card specific BBs.

![Figure 55: Peripheral Card Maintenance](image)

The card maintenance BBs define recovery actions, error handling, configuration parameters of the peripheral cards. Peripheral cards and their maintenance BBs are self-describing (see section 7.8).

**Line Handling**

The line-handling BBs are logical resources which handle all attributes and properties of lines as defined by ITU, i.e. subscribers (analog, digital, ISDN). ITU describes
logical resources as managed objects. BBs are structured according to these object definitions.

**Announcement Handling**

The tss system also offers automatic announcement services. Examples are automatic time announcement, operator console number announcement and automatic subscriber number announcement for directory services. The BB announcement handling in the service management layer receives ASCII strings containing the time, operator console number or the subscriber number. It has to translate the ASCII string into a number of speech fragments. A special service function card plays the announcement for a caller. Each actual announcement consists of a fixed part ("the current time is", "hours", "minutes", "and", "seconds") and a flexible part ("twelve","thirteen","fourteen"). However, the translation will vary for each type of announcement. The subscriber number string "213719" is translated into the speech fragments "two", "one", "three", "seven", "one", 'nine". The same string for time announcement translates to "twentyone", thirtyseven", "nineteen". Each translation has been encapsulated in one BB. Adding announcements in different languages, for example, only requires loading the corresponding speech fragments and adding a new translation BB.

**Call Facilities**

The call facilities in the service management layer are encapsulated in a separate BB per facility. Facilities are state machines which are active during the call. That is, a facility such as call forwarding on busy is a type of state machine which is instantiated per call. A generic BB implements the basic call model. Facilities extend this basic model, that is, facility BBs are coupled to the generic BB. A customer requiring a new facility receives just the corresponding BB.

**8.4 Managed Evolution**

The functionality of almost all products is constantly being changed. The technology used to realise those products is also changing. Evolution is a very important issue for successful product development. The reason for that is that evolution affects all areas of development. Each analysis and design task has to be considered from the perspective of evolution.

The important point when preparing for evolution is that one addresses those facets of evolution which can be handled without attempting to address the unpredictability of the future. An architecture’s future-proofness (see section
2.2) therefore is relative to those issues which are foreseeable. Through consequent usage of the available knowledge about likely changes, evolution may be dealt with in product development in a controlled manner.

Perry [Per94] characterises evolution in three dimensions: relevant domains, experience and process. He emphasises that these dimensions capture the nature of evolution much better than performed changes like corrections, improvements and enhancements.

Several of the concepts of the BBM address evolution. The three design dimensions - object, aspect and thread - separate design issues and support independent handling of them. Evolution should preferably even be limited to a single dimension, the object dimension (see section 3.3). BBs and layers support evolution through the incrementality of layers (see section 7.4.2) and generic BBs (see section 7.5.3)

The family architecture, then, is focused on features. This allows flexible definition of products as members of a product family. Moreover, since features change over time and new features are conceived, different products are defined via different feature sets, and a product may over time evolve into a different one. Conservative and non-conservative extensions (see section 8.3.3) are both necessary to keep an architecture balanced with respect to conceptual integrity and development efficiency.

The concept of a product family is very useful because of its mixture of properties. It is a business concept as well as a technical one ([MF93], [ML97]). On the business side a product family is viewed as a collection of parallel and consecutive products made out of sets of features. On the technical side a product family architecture provides a collection of BBs from which different products are configured.

**Heuristics Overview**

*Heuristic 80: Use a feature relation graph to describe relations between features.*

*Heuristic 81: Design a system infrastructure generic to handle data-driven diversity.*

*Heuristic 82: A desirable non-conservative extension is the refactoring of a BB to a generic BB.*

*Heuristic 83: A good family architecture is one whose BB structure resembles the feature structure, i.e. a good family architecture is feature-oriented.*