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.
Building Blocks are software components. The definition of the term software component follows that of Szyperski [Szy98]:

"Software components are executable units of independent production, acquisition, and deployment that interact to form a functioning system."

The BBM uses the notion of a product family (chapter 8) to cover the market aspect of components. This chapter explains BBs from a technical point of view.

BBs are design and deployment units. Identification of BBs usually goes along the object dimension, that is, a BB is a cluster of objects. However, this is no restriction. A BB can follow the other dimensions as well, or even encapsulate arbitrary parts of the three design dimensions. The main criteria are configurability and incremental integration, as will be outlined in this chapter.

The chapter starts with an overview of the composability design task. Then follows an explanation of the most important ingredient of a BB, its interfaces. The interfaces will be discussed only summarily, but this brief treatment will suffice for our purpose. A more elaborate treatment can be found in [Szy98]. BBs, like other software components, are based on a common component model. The component model defines standard properties for components. Component models will be introduced in the third section, after which layering of BBs will be explained. The roles of BBs, generic or specific, will be dealt with in the following section. A further section will take up the issue of interfaces again and relate it to layering and genericity. Hierarchical components in the BBM will be described then. A section follows which introduces the concept of an architectural skeleton as a way of organising inter-BB relations.

The last section of this chapter is about deployability design. Deployability design deals with possible deployment scenarios of products to the hardware environment. BBs are the minimal deployable units. Deployability design
describes rules for refactoring and grouping of BBs to adapt them to certain deployment scenarios.

If not described differently, we depict BBs by boxes and their directed dependency relation by lines between boxes where BBs located above depend on BBs located below.

7.1 Composability Design Overview

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. This overview covers the application of concepts explained in this and the next chapter.

Composability design consists of the following design steps:

Clustering objects into BBs

There are several criteria for clustering objects into BBs. They are:

_Heuristic 44:_ Cluster objects into BBs such that coupling of objects across BB borders is low and cohesion of objects within a BB is high.

_Heuristic 45:_ Cluster objects into a BB which represent a feature.

_Heuristic 46:_ Cluster objects into different BBs which belong to independently evolvable parts.

_Heuristic 47:_ Cluster objects into BBs such that a BB can be used as a work allocation units for 1 or 2 persons.

Identifying variation points of functionality which belongs to different features

Start with variation points identified in the application domain model.

Select those variations which can be solved by data. Configuration parameters may be supplied by users or via configuration files and passed to the appropriate places.
For variation points requiring code, analyse if the variation point is inside a BB or at an interface of a BB. If the variation point is at the border of a BB, the desired situation, in which variation is modelled by alternative BBs, is already reached.

**Heuristic 48:** If the variation point lies inside a BB, refactor the BB such that the variation point lies at the border of a BB.

Refactoring of BBs affects the object structure and may require refactoring of objects in object design.

**Factoring out of common functionality in separate BBs.**

**Heuristic 49:** Factor out functionality which is present in several BBs in a separate BB.

This may also lead to a refactoring of objects.

**Identifying interfaces of BBs:**

Create abstractions at the interfaces which are implementation-independent and efficiently executable. Often there is a tension between these two requirements.

**Heuristic 50:** Take as main criterion stability under evolution, that is, an interface should be such that it can serve for those implementations and those usages which are likely to happen.

**Designing component frameworks**

A component framework (generic BB) is designed when a BB contains only the generic parts of an implementation. The various specific parts (plug-ins) are located in specific BBs. The specific BBs are extensions of the generic BB. A generic BB can usually not be used without its extensions. The coupling between generic BB and specific BB is usually tight, that is, changes of the implementation of the generic BB will affect the specific BBs. The interface of the generic BB to other BBs should be stable.

**Heuristic 51:** Factor generic implementation parts which are used by several specific parts into a generic BB.

There are examples of BBs which are generic w.r.t. several kinds of extensions. This leads to the introduction of the concept of a generic role (see section 7.5.2).
The introduction of a generic BB will usually lead to a refactoring of objects. But cases where one or more aspects of certain objects are factored out are also possible.

**Identifying system infrastructure generics**

Generic functionality which is to be extended by almost all BBs is put in system infrastructure generics (SIG). A SIG is a special case of a generic BB.

*Heuristic 52: Take the implementation of common aspect functionality as a candidate for a SIG.*

Examples are the initialisation model of the system or the handling of user reports. SIGs are mostly part of the operating system or middleware packages.

**Defining layered subsystems of BBs:**

Start with the layers defined during object design. Place a BB in the lowest possible layer of a layered subsystem (recursive layering) according to the dependency relation with other BBs.

**Designing for incremental integratability and testing:**

Only uni-directional relations between BBs are allowed. Transform mutual dependence into a uni-directional one. The criterion for refactoring is again expected stability under evolution.

*Heuristic 53: Resolve mutual dependence between BB A and BB B in the follow way:
  if A is expected to be more stable than B, then make B depend on A; and vice versa
  if the communication between A and B is expected to be the most stable part, factor the communication out into a new BB and let both, A and B, depend on it.*

Give each BB sufficient functionality for useful tests. This means for a generic BB that it should have useful behaviour (null-behaviour) without any specific BB connected to it. Specific BBs contain extensions of the generic BB.

**Doing detailed design of the BB**

Elaborate the identified interfaces and design the internals of BBs. This may mean refinement of objects, design of data structures and algorithms as part of the object design task and the aspect design task.
The extensive refactoring and factoring into generics leads to a compactness of the code (see section A.5.4) and reduces the size of the system. The detailed description of composability design is split into two chapters. This chapter describes the design of BB, layering and generics. Chapter 8 describes the design of a product family architecture. We will now look at the concepts in detail and start with interfaces of a BB.

### 7.2 Interfaces

First we describe abstraction interfaces and open implementation interfaces. Then we discuss the notion of a connector. Registration and call-back interfaces are introduced next. They are important concepts for achieving uni-directional dependencies. The last section discusses the relation between interfaces, and aspects and threads. In section 7.6 after the discussion of layering and generic BBs we will come back to the topic of interfaces.

#### 7.2.1 Abstraction Interfaces

BBs have a *provides* interface, a *requires* interface and a body which is not visible but only accessible via the provides interface.

*Provides interfaces* define functionality of a BB that can be accessed by other BBs. On the syntactic level, the *provides* interface consists of a (structured) list of method signatures and a description of all their data types. This may be extended with any semantic information which a particular context requires. For example, pre- and post-conditions may be added if necessary. A very common practice is to describe an interface protocol that describes a suggested or required ordering of method calls. However, we do not require such a description because not every context needs it. Provides interfaces should be carefully designed to be independent of implementation detail and consist of abstractions of the functionality which do not prescribe a certain implementation.

The *requires interface* (see figure 34) describes provides interfaces of other BBs that are required for the implementation of a BB.

Note that these interfaces include those explicitly identified during specification of an application as well as additional ones required for the implementation, such as those to infrastructure services.
Requires interfaces consist of a set of references to BBs, a list of those method signatures which are actually used by the BB and the description of all the used data types. The requires interface makes the dependencies of a BB explicit. To be able to use provides interfaces of other BBs, a BB has to reference (import) these interfaces.

**Heuristic 54:** *In the case of embedded systems, use importing of interfaces at compile time if needed for performance reasons. Otherwise use dynamic exploration of interfaces for more flexibility.*

A BB is dependent on the BBs from which it imports interfaces.

For a discussion of the issue of object-oriented programming and stable interfaces we refer to [Szy98]. In particular, this work explains the problems involved in using implementation inheritance across component boundaries (see the fragile base class problems) and discusses alternatives. The BBM relies on object composition instead and avoids these problems, because BB interaction is realised via explicit interfaces.

### 7.2.2 Open Implementation Interfaces

Unlike abstraction interfaces, open implementation interfaces, as introduced by Kiczales [Kic96], do not establish an abstraction for information hiding [Par72],
but are connection points to other BBs which provide implementation alternatives (figure 35). The BBs, which extend a component via its open implementation interface, are extensions of that component. Open implementation interfaces are not intended to remain unchanged if the implementation of the component which offers them changes. Both, abstraction interfaces and open implementation interfaces are provides interfaces. In depicting BBs, we do usually not show the difference between abstraction and open implementation interfaces as done by [Kic96] shown in figure 35. The emphasis is only on the fact that there is a dependency relation between BBs.

Abstraction interfaces and open implementation interfaces may be compared to upper and lower interfaces, respectively, as described for Catalysis [DW99]. However, abstraction interfaces and open implementation interfaces are both provides interfaces. Catalysis, in contrast, uses lower interfaces both for plug-ins and for access to an underlying virtual machine. Plug-ins, however, are optional extensions and should rely on provides interfaces, while access to a virtual machine is part of the requires interface.

7.2.3 Interfaces, Components and Connectors

Some authors recommend modelling a system on the basis of components and connectors [SG96].

Interfaces are collections of methods at the programming language level. An interface is the most trivial connector. It connects two components, one of which implements the interface as a provides interface while the other implements it as a requires interface. More complex connectors are buses, pipes, blackboards, remote method call packages, etc. In fact, a connector can be any abstract data type or other
package which connects two or more components. In the BBM, all but the most simple connectors are implemented as components themselves.

The notion of a connector is useful in top-down modelling. Application entities can be connected abstractly. The precise properties of a connector can be determined later. Product variations may require different connectors. The BBM does not use connectors as a primary concept because the concept of a generic BB can be used instead. Connectors can be modelled as special classes of generic BB (for an example see section 7.5.4).

7.2.4 Registration and Call-Back Interfaces

Call-back interfaces offer a mechanism for making components minimally dependent on the context in which they execute. Besides the services part of the provides interface, a BB provides a call-back interface through which it can call methods of the calling BB. Instead of adapting a service-providing BB to many different interfaces of using BBs, the service-providing BB sets a standard which has to be met by using BBs. The service providing BB itself thus remains independent of the using BBs while adapting its behaviour.

The call-back mechanism is shown in figure 36. BB A defines a service interface, a register interface and a call-back interface. BB B, interested in the service of A, has to register methods, which conform to the call-back interface, with BB A. A call-back occurs when A calls the registered methods of B.

![Figure 36: Call Back Mechanism](image)

However, BB B may have to meet restrictions in the implementation of the call-back methods.

Examples are certain interfaces which may not be called during the call-back from A, and execution time limitations of the call-back (for a broader discussion of these restrictions, see [Cla85]).

The BBM requires that BBs have syntactically only unidirectional relations. Mutual dependencies are not allowed. This establishes a partial ordering of BBs.
The partial ordering of BBs is a necessary condition for integrating a system incrementally. If a design requires bidirectional communication, call-back interfaces have to be used to establish a bidirectional communication. Below, we shall give guidelines for design with unidirectional relations.

### 7.2.5 Interfaces, Aspects and Concurrency

Besides their functional characteristics, interfaces must describe additional information.

*Heuristic 55: Structure interfaces according to aspects.*

They enable the connection of aspect functionality which is distributed over BBs. The interface description of the BB should describe the aspect to which an interface belongs.

Similarly, the restrictions imposed by a concurrency design have to be described with an interface, that is, the assumptions about threading have to be made explicit in the interface description. Examples are information about re-entrance of interfaces, timing constraints and resource usage.

Besides the information which is described per interface of a BB, the overall descriptions of aspect designs and the concurrency design support the understanding of the role of the interfaces.

### 7.3 Component Models

Another facet of BBs is the underlying component model. The component model defines how a component is accessed. In particular, calling conventions for the use of the interfaces are defined by the component model.

Different component models are currently proposed. The most prominent examples are the Component Object Model (COM) of Microsoft, JavaBeans of Sun and CORBA of the OMG. Other component models have been developed for specific applications. To achieve wide applicability, these component models have to demonstrate their usefulness for different application domains, interoperability with existing (legacy) software and bridging to other component models. We shall neither explain nor compare these component models. Szyperski [Szy98] gives a good introduction to and comparison of COM, JavaBeans and CORBA.
The BBM just assumes that a component model is used. Nevertheless, as an example, we describe the component model of the tss system in appendix A.3.5.1. The tss component model is a dedicated component model developed in the mid-eighties to allow flexible design, implementation, loading [Fra97] and testing of BBs. The tss component model shows that components can be implemented with little overhead. This is an important consideration for embedded systems.

7.4 Layering

In section 4.2.1 layering of objects was introduced for separating functionality with different evolution characteristics. Layering for BBs extends and refines the notion of layering of objects. BBs make use of interface abstractions of other BBs. The abstraction serves as an infrastructure for the using BBs. Each BB provides a part of an infrastructure for one or more using BBs. The BBM requires the infrastructure to be independent of the supported BBs. To achieve this independence, mutual relations are transformed in unidirectional relations (see also section 7.4.2).

The BBM uses a *layer* as a clustering of functionality which has unidirectional syntactical relations only, that is, a layer depends on lower layers, and higher layers may depend on it. Parnas identifies layering, besides information hiding, as a desirable property of a system structure [Par72].

*Heuristic 56: Use layering for BBs on two levels. Subsystems, which are collections of BBs, are layered. These layers are based on the classification of layers of domain objects done during object design.*

*Heuristic 57: Individual BBs within subsystems are also layered in relation to other BBs.*

Layering of subsystems will be discussed in this section. Section 7.5 on generic and specific functionality describes layering of individual BBs.

7.4.1 Layering Principles

Structuring of functionality in an infrastructure and its using applications can be based on several principles: layers can have a different scope of visibility, layers can be used conceptually or even in implementation, and layers can provide dif-
different degrees of completeness. The following subsections introduce the different kinds of layering.

**Application versus Technology Layering**

*Heuristic 58:* A common principle for the layering of software is to separate hardware-technology-oriented functionality from application-oriented functionality.

Hardware-technology-oriented functionality is seen as the base on which the application-oriented functionality is implemented (section 4.2.1). This approach is often taken in interactive systems. If the user interaction needs to be regularly adapted or extended, this layering approach (figure 37) provides flexibility since changing the application on the basis of a stable infrastructure restricts the impact of those changes. A variant of this principle is the separation of base technology from application functionality. [BGK*99] and [RE99] use several layers to bridge from base technology to application-specific frameworks.

**Abstraction from HW**

A related layering principle is one in which layers abstract from the concrete HW [Dij68]. Each layer provides abstractions for the higher layers. Layers are also called virtual machines. In figure 38 the lowest layer consists of the HW drivers. Logical drivers abstract HW specific attributes and constitute the second layer. Common
services are a collection of services used by many applications. The user applications, finally, constitute the highest layer.

![Diagram](image-url)

**Figure 38: Abstraction from HW**

*Heuristic 59: Construct layers as virtual machines for higher layers.*

**Generic versus Specific Functionality**

*Heuristic 60: Another way of introducing layers is to distinguish between generic and specific functionality.*

Generic functionality is the infrastructure on which specific functionality is built (figure 39). This principle is used when generic middleware is separated from application programs. More generally, interface abstractions can be encapsulated in a BB. The BB which actually provides the functionality of the interface BB and the BB which uses these interface abstractions to access the functionality are both located in a higher layer. The BB which contains the interface abstractions is more generic than the other ones and therefore resides in a lower layer.

![Diagram](image-url)

**Figure 39: Generic vs. Specific**

*Example: tss Layering Principles*

Within the tss system two principles for layering have been combined, notably abstraction from hardware and generic vs. specific functionality.

The rule that all relations are uni-directional creates some tension between the application of both principles. Generic services are often hardware-independent.
According to the *abstraction from hardware* principle, these services are located in higher layers, whereas according to the generic vs. specific functionality, these services are located in lower layers. For example, an interface between hardware drivers and the remainder of the system is more generic than the specific cases of the hardware drivers. Therefore, the BB containing the interface has to be lower than the specific hardware drivers. However, drivers are closer to the hardware itself. No general rule exists to reconcile these two principles for the actual layering of system functionality. A decision has to be taken for each function separately.

The tss central controller layers (subsystems) (see section 7.4.4) are based on the abstraction from hardware principle. But the system infrastructure generics (see below) are also placed in the lowest layer. Within each layer there is a micro-layering which separates generic from specific functionality (see section 7.5.3).

### 7.4.2 Incremental Layering

A layer is said to be *incremental* if the dependency relation of this layer satisfies the following criteria:

1. The import graph is acyclic, that is, the dependencies are unidirectional.
2. The semantics of layer $L$ may depend only on the semantics of layers from which $L$ imports.

The second criterion especially states that a BB must have well-defined meaning even if call-back methods have not (yet) been registered. This means the BB may neither depend on the presence of registered call-back methods nor on the semantics of these methods.

An incremental layer builds a platform for BBs of the higher layers, that is, it implements a virtual machine. The semantic independence of higher layers requires functional completeness of an incremental layer and the layers below it. We call this functional completeness of a layered subsystem its *platform property*.

Layering is already started during object design. The classification of functionality in layers is a step for getting meaningful functionality per layer from the application domain.

This platform property requires careful design. Techniques for defining a fine-grained platform for a single BB will be described in section 7.5.

The main reason for incremental layering is complexity management. A system with incremental layers can be integrated and tested layer by layer. [Dij68] and [HFC76] mentioned incremental testability as one of the key advantages of incremental layers.
This partial (in)dependence of layers is a compromise between total independence, which is not possible, and total dependence, which is not desirable.

A system is integrated according to its incremental layer structure. They are compiled and linked incrementally. If the target system is equipped with an incremental loader, the layers can be loaded incrementally. Each increment may be developed and tested as soon as enough information on the underlying system is available. System construction does not have to wait until all the increments are present. A small (core) system may be built from initial BBs. The system may be extended repeatedly with new increments until the entire system is ready. Incremental layering is a key concept of the BBM.

7.4.3 More Facets of Layering

Besides the layering principles and the incremental nature of layers, other facets of layering are important. The following subsections discuss several alternative strategies for layering.

7.4.3.1 Conceptual versus Strict Layering

Conceptual layering is a design technique which models functionality in such a way that a particular layer is conceptually independent of other layers.

Strict layering is an implementation technique in which no compile or link time dependencies exist from a layer to a higher layer. This technique makes it possible to build each layer (or rather each increment) independently of the higher layers. Furthermore, a system can be loaded increment by increment.

Strict layering implies conceptual layering, but the reverse does not hold.

An example of conceptual layering only is a middleware layer which is independent of its using applications but has hardwired knowledge of these applications to be able to activate them. In the implementation such a middleware layer needs to be updated for new applications.

The BBM relies on strict layering of BBs. Strict layering is a precondition both for incrementally adding new BBs to the deployed system and for removing BBs from it.

7.4.3.2 Opaque versus Transparent Layering

There are two alternatives to the visibility of layers:
opaque layering: each layer hides the layers below it. A layer may only use functionality provided by the layer directly below it.

transparent layering: layers do not hide the lower layers. A layer may use all lower-level layers.

Other terms for opaque and transparent layering are strict layering (different from our usage of the term) and non-strict layering, respectively, proposed by [Szy98], or closed and open architecture, respectively, proposed by [RBP*91].

Opaque layering requires each layer either to provide an additional abstraction for all the lower-layer services needed in higher layers or to add dummy interfaces. This eases the development of a particular layer, because a layer uses only interfaces of one layer below. Furthermore, restructuring of BBs in a lower layer is shielded from visibility of a higher layer by an opaque layer. On the other hand, updating functionality becomes more complex. Functionality from the lower layers used by a higher layer has to be presented to the higher layer in one form or another. Updates and successive testing are therefore required even if no functionality within a layer itself has been changed.

Transparent layering assumes that each layer provides only its own services. Only if an additional abstraction or an additional service is needed, an interface is introduced in a layer. In the case of transparent layering the implementors of a layer must know in which layer they will find the necessary interfaces. No maintenance effort is necessary in intermediate layers for making new or changed services of the lower layers available.

Opaque layering focuses on the using layer. The usage relations are defined to be exactly those of one layer with the layer immediately below it. Transparent layering focuses on the providing layer. The changes necessary after an update are restricted to the providing layer where a service is located and the layers of its direct users. The users may be from multiple higher layers. No effort in intermediate layers is necessary.

The BBM permits both techniques.

*Heuristic 61: The usage of transparent layers is favourable to the usage of opaque ones.*

*Heuristic 62: Opacity is used for layers that function as facades, such as abstraction layers for hardware, operating system or middleware.*

The facade pattern is described in ([GHJ*94])
7.4.3.3 Partial versus Complete Layering

Sometimes systems are designed with partial layering only. For example, functionality that belongs to an application domain may be layered while functionality that provides computing resources is outside this layering. An example is given in figure 40.

![Partial Layering Diagram](image)

*Figure 40: Partial Layering*

All layers may make use of the computing resources and the computing resources may or may not access any of the layered functionality. The computing resources are a kind of general entity which must always be present and needs to be adapted to entities present in the layers.

The BBM does not allow partial layering. Instead, if, in the example, partial layering were to be restricted so that only layered functionality would access the computing resources and, if necessary, register call-backs, it would be equivalent to a complete layering in which the computing resources are the lowest layer and are visible to all the other layers. In the last situation there would be no bidirectional dependencies.

7.4.3.4 Communication and Layers

*Heuristic 63: Use layers to structure communication in a system.*

Suppose a functional element in the system communicates with another one in the same layer. Communication happens, then, between elements of the same conceptual level. However, if the functional element may only have relations to functional elements of lower layers, the communication may not be implemented directly. Unidirectional dependencies require that such a communication relation be established indirectly. The lower layers are used for the implementation of the
peer-to-peer communication (figure 41). They transport messages from one peer to the other and notify the receiving peer.

![Figure 41: Indirect Peer-to-Peer Communication](image)

This model of peer-to-peer communication introduces semantic relations but no direct syntactic relations between the communicating parties. The ISO-OSI communication model [DZ83] made this communication structuring popular for computer-to-computer communication. Layered subsystems in a distributed system usually also communicate in this way. In BBM-based systems peer-to-peer communication will always be implemented via indirect communication.

For reasons of performance, an optimised version of layered peer-to-peer communication based on transparent layering may be introduced. Layered application entities make direct use of a lower-layer communication facility without intermediate packaging. Available knowledge about the sending environment, the communication channel and the receiving environment is used to minimise intermediate processing. Required functionality of intermediate layers may be implemented by the application layers taking advantage of the application knowledge. The communication within the tss system is based on such optimised peer-to-peer communication (section A.2.3).

### 7.4.4 Example: tss Subsystems

tss layered subsystems have been described in section 4.3. They comprise the extended operating system, equipment maintenance, logical resource management and service management. The subsystems have incremental semantics. The layering is strict, transparent and complete. The operating system provides a facade interface (opaque layering, heuristic 62).
7.5 Generic and Specific Functionality

A very important way of structuring software is by separating generic from specific functionality. Functionality of similar applications is analysed for diversity. This analysis is applied to domain objects, domain functions, algorithms, etc. [CHW98]. The common part is factored out and captured as an abstract concept. The common part is called generic, the diverse parts are called specific.

Separation of functionality into generic functionality and specific functionality is also present in object-oriented modelling and in design patterns [GHJ*94] (see table 3).

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Generic</th>
<th>Specifics</th>
</tr>
</thead>
<tbody>
<tr>
<td>OO modelling</td>
<td>supertype</td>
<td>subtypes</td>
</tr>
<tr>
<td>strategy pattern</td>
<td>data structure</td>
<td>algorithms</td>
</tr>
<tr>
<td>container</td>
<td>algorithm</td>
<td>data structures</td>
</tr>
<tr>
<td>template method pattern</td>
<td>skeleton algorithm</td>
<td>fill-in steps</td>
</tr>
<tr>
<td>bridge pattern</td>
<td>interface</td>
<td>implementations</td>
</tr>
<tr>
<td>observer pattern</td>
<td>event source</td>
<td>event listeners</td>
</tr>
</tbody>
</table>

Table 3: Examples of Separation of Generic and Specifics

The observer pattern which decouples event source from event listeners is an extreme form of generic and specific. The coupling between the generic part and potentially many specific listeners is reduced to events. Essentially, two independent entities are coupled via some events. Common to all examples is that there may be several specifics that are related to one generic.

The specific part is an extension of the generic part. Replacing functionality of the generic part by the specific is not intended. Functional units are built by combining a generic part and a specific part. Achieving stable generic parts for an application domain eases evolution.

The separation of generic and specific functionality is relevant for areas other than software too. Examples are system requirements, domain modelling, hardware, and development and customer documentation. Using the same separation in related areas makes correlating them easier. Domain terminology, for instance, may be extended with those generic concepts.
Note that there may be degrees of genericity. Certain pieces of functions are common to all members of the family, whereas others are common to only a few members of the family. In the latter case we still have generic functionality, but on a smaller scale.

The embodiment of the generic functionality represents a major part of the know-how of an application domain, cf. the domain kernel of Gomaa [Gom95].

**Heuristic 64: Separate common functionality from specific functionality.**

Analysis of functionality in terms of generic and specific functionality is the key concept in architecting a product family [Par76]. Frequently used terms are commonality analysis [CHW98] and diversity analysis. It is common experience that diversity analysis is more effective for identifying generic parts than commonality analysis [KMM96].

**Heuristic 65: Look for the diverse parts in similar functionality for different features.**

### 7.5.1 Generic and Specific BBs

Generic and specific functionality is captured in different BBs. A generic BB contains generic functionality and a specific BB contains specific functionality. A *specific BB* extends the functionality of a generic BB. A *generic BB* is a component framework [Szy98] to which specific BBs, also called plug-ins, are added to extend its functionality. Figure 42 shows the basic inter-BB pattern generated by the BBM.

A generic BB defines standards for its specific BBs similar to a superclass setting standards for its subclasses. However, the specific BB extends the generic BB conservatively, whereas subclasses are often also allowed to overwrite parts of the superclass.

[FS97] call those frameworks black-box frameworks in contrast to object-oriented frameworks, which are called white-box frameworks.

More specifically, the generic BB defines interfaces for its specific BBs. If the interface between the generic BB and the specific BB is based on a domain object, the interface is stable because of its anchoring in the application domain. Usually, however, the interfaces between generic and specific BB depend on the implementation of the generic BB.

Implementation-dependent interfaces are called open implementation interfaces [Kic96] (see section 7.2.2). Call-back methods, also called hook methods in [FS97],
decouple the generic behaviour of an application domain, encoded by the generic BB, from the specific instantiations needed for a particular application, as encoded by the specific BB.

General application processing steps are executed by the generic BB. The invocation of methods of specific BBs customises and extends this generic behaviour to one for the specific application domain. Inversion of control [FS97] means that the generic BB (rather than each specific BB) has control and gives it to specific BBs to do certain actions.

*Heuristic 66: Use inversion of control for designing the functionality of a generic BBs.*

For instance, external event handling is effected in the generic BB, which decides which methods of a specific BB to call in response to those events.

Modelling a generic service requires careful analysis. To become stable, a generic BB usually undergoes several redesigns [Sch97].

*Heuristic 67: A generic BB is stable if new specific functionalities may be based on the generic BB without changing it.*

Of course, generic BBs may be adapted and/or added during the lifetime of the product family. A generic BB embodies the similarities of certain functions. Therefore, changes in a good generic BB are rarely necessary. However, changes
in a generic BB almost always have consequences for all the corresponding specific BBs.

Each specific BB uses the functionality provided by the generic in its own way. Contrary to superclasses, a BB hides its internal structure. The specific BBs do not have direct access to the internal details of the generics. This means that the specific BBs may use only the interfaces provided by the generic to access the generic functionality.

7.5.2 Generic and Specific BB Roles

In practical situations a BB will often be both, generic and specific. It is therefore necessary to use the term generic and specific in relation to some functionality. The terms generic role and specific role of a BB are used to denote that a BB contains generic or specific functionality. A BB can have several generic and / or specific roles.

Most BBs have multiple specific roles. This is similar to a class structure with multiple inheritance in object-oriented modelling. The difference with multiple inheritance is that generic BBs have explicit interfaces and the generic-specific relation is not recursive.

Note that we use the term generic BB without reference to its particular role if it is clear from the context what that role is. Thus, a BB which has at least one generic role may be called a generic BB for short.

7.5.3 Generics and Layering

The BBM uses layering to structure overall functionality (section 7.4). But layering is also used to structure inter-BB relations. Since a specific BB is an extension of a generic one, the specific BB is dependent on the generic BB. The specific BB imports an interface of the generic BB.

Besides syntactical independence of its specific BBs, a generic BB is required to be semantically independent too. Layers have an incremental character (section 7.4.2), but so does a generic BB with respect to its specifics. Semantic independence means that a generic BB without registered call-backs to specific functions has to perform basic functions without leading to any error (section 11.7) to enable incremental testing.

A generic BB is a kind of infrastructure on top of which specific BBs are located. The generic BB is located in a layer below the specific BBs.
This may seem counter-intuitive from the point of view of object-oriented modelling in which super classes are usually drawn above subclasses. In contrast, the BBM uses the notion of an extensible platform upon which further BBs are constructed (see figure 43).

For instance an operating system can to a great extent be modelled to consist of generic services which are used by almost all the application.

Specific BBs register call-backs with their generic BBs (section 7.2.4). In fact, BBs with generic roles are the only BBs that provide a registration interface. The BBs with specific roles are the only BBs that use the registration interface and have a call-back interface. In figure 43 a generic BB is depicted with its specifics, together with the registration and use interfaces of the generic. The arrows indicate method invocations.

7.5.4 Classification of Generic BBs

Generic BBs can be classified according to patterns. The idea is that the very general concept of separation between generic and specific functionality leads to a number of domain-specific types of generics. We describe four different types as part of the core BBM. The list is not exhaustive but can be extended with new ones by specialised versions of the BBM.
**Abstraction Generic**

An abstraction generic is the basic form of a generic (see figure 44).

*Heuristic 68: Use an abstraction generic to implement an abstract concept which is to be extended by specific BBs.*

The distribution of functionality may range from a very thin generic to a very thick one. A thin generic means that the generic implements almost exclusively an interface and the rest of the functionality resides in the specifics. A thick generic means that the specifics add only minor variant data or functions.

![Diagram](image)

*Figure 44: Abstraction Generic*

An example of a thick generic is one which implements a template method pattern [GHJ*94] in which the factor-out steps are small.

An example of a thin generic is a generic which only forwards calls to its specifics. Ethos [Szy92a] uses *directory objects* to access basic services in an operating system which should be extensible. A file system, for instance, is not directly accessed to create a new file but the access goes via a directory object. A file system can be updated in a running system without its clients noticing. A somewhat thicker generic would be a file handling generic which provides the interface to the client, does general file administration but has the handling of different file types, such as sequential or indexed files, factored out in specifics.

All variations between the extremely thin and the extremely thick ones are possible. Examples given by Kiczales [Kic96] such as an application-specific paging strategy that can be coupled to a generic memory handler would probably rank between the extremes.
Connectable Resource Generic

Connectable resource generics are defined to manage connectable resources which are supplied via HW boards. Changing the configuration of those boards also changes the number of connectable resources.

*Heuristic 69: Use a connectable resource generic to manage connectable resources which are supplied by HW boards.*

![Resource Supplier](image1) ![Resource Generic](image2) ![Resource Customer](image3)

*Figure 45: Connectable Resource Generic and Resource Flow*

Examples of tss are boards which provide a number of connectors for other boards, or timeslots to be used by some application. These connectors or timeslots are resources supporting system configurability. The generic has the task to administer these resources.

Figure 45 shows a connectable resource generic with one supplier and one customer. Note that the arrows in the figure show the resource provisioning from supplier to customer and not the dependency relation. The dependency relation is from the specifics, that is the resource supplier and the resource customer, to the generic.

The connectable resource generic handles the resource abstraction and has two classes of interfaces. The first class is for supplying and allocating resources, the second for communication between resource supplier and customer. The connectable resource generic has an allocation table which relates instances of sup-
pliers to instances of customers. The communication interfaces have one of the following characteristics:

- a customer instance signals some event to its supplier instance,
- a supplier instance notifies all the related customer instances (broadcast) about some event, and
- a supplier instance notifies a specific customer instance about some event on the basis of some selection criteria.

Suppliers and customers of a connectable resource generic may evolve independently. A connectable resource generic bears some resemblance to a bottleneck interface [Szy98]. A bottleneck interface couples two independently extensible abstractions and is itself not extensible. A connectable resource generic also allows independent extension of suppliers and customers via their respective specific BBs.

**System Infrastructure Generic**

System infrastructure generics provide an operating infrastructure for application BBs. System infrastructure denotes a collection of services which is part of the operating system, standard UI functionality, or generic and domain-specific middleware.

_Heuristic 70: Design a system infrastructure generic for functionality which provides an operating infrastructure for almost all application BBs._

System infrastructure generics (SIG) are a means for encapsulating these services. The characteristic of a SIG is that all application BBs are potential specifics of it. The operating infrastructure provides the basic framework whereas individual BBs only provide increments.

SIGs may administer system resources such as processor time and memory pools, and standardise system functionality such as the user interface, a recovery strategy or error handling.

_Heuristic 71: System Infrastructure Generics must provide interfaces for application BBs for indicating their resource requirements._

The SIG will either have service interfaces to provide data directly or registration interfaces to register call-backs which the SIG uses to retrieve the data from the application BBs.
Generic parts of aspects (chapter 5) which can share a common implementation are factored out into SIGs. This leads to an instantiatable aspect infrastructure.

A SIG for recovery handling is such an example. A generic report handler may be used by the configuration management aspect and the fault management aspect. An internal message handler may be used by more aspects.

Since these generics offer very generic functionality, they are located low in the system hierarchy of layers, i.e. they can be seen as part of an extended operating system. Layering them low in the system is an example of the generic-versus-specific functionality layering principle. An application BB has a specific role for many of the SIGs. Figure 46 shows a BB together with three SIGs.

![Diagram of Building Block and SIGs](image)

*Figure 46: System Infrastructure Generics*

Standardisation of SIGs may go so far that specific functionality is reduced to specific data instances. All the relevant algorithms reside in the SIG and the specific data instances reside in the specific parts. In such a case BBs do not access these data directly, but through a SIG.

An example of such a SIG is one which handles exceptions. The SIG defines standardised functionality such as the handling of severity of exceptions, writing to a log file and communication to a user interface, whereas the actual severity, output formats and reporting texts are located in the application BB.
Using a SIG to implement system functions such as exception handling is an alternative to a coded case statement with each case alternative representing a specific instance of an exception. Maintaining lists of case alternatives for an evolving system leads to frequent changes in the source code of the exception handler. This is a potential source of errors. In contrast to this a SIG is not changed. The list of different exceptions in a specific product is automatically built by the configuration of selected BBs in the product. Consider the addition of a BB to the system. No recompilation, relinking or reloading of, for instance, exception handling is necessary. Instead, the BBs define their own exceptions and make them known to exception handling. After adding the Building Block, the system will have an adapted list of exceptions. There are, hence, no separate configuration files or common include files in the system.

The use of the SIG can reduces the code of the specific BBs considerably. Furthermore, this standardisation allows automatic code generation. Specific functionality of a SIG may be reduced to parametrised data structures. The collection of specific BBs together with the SIG implement a configurable table in the running system whereby all access methods are implemented in the SIG itself. The parts that extend the table can be generated automatically.

The tss system used a specific database-based tool (section 11.8) for data generation. Because of their relevance for the whole system, data can easily be reviewed and changed without entering an implementation phase. Part of the production of a BB is the code generation for all tool-supported SIGs. An implementor of a BB may not even be aware that some of the SIG-specific data of his BB form part of the Building Block.

Often when diagrams with relations between BBs are drawn, the relations to SIGs are not shown because all the BBs are specifics of at least some of the SIGs.

**Layer Access Generic**

Layer access generics provide access to a layer’s functionality. They restrict the visibility of the structure within a layer for higher layers, i.e. references are channelled through a number of specifically designed indirections. The advantages of layer access generics are, first, that higher-layer BBs only have to know about layer access generics of the lower layers, and, second, that they provide for configurability of BBs in lower layers without any need to update BBs in higher layers.

*Heuristic 72: Design a layer access generic to restrict the visibility of the structure of a layer for higher layers.*
A specific pattern is used to achieve this purpose. In contrast to normal interfaces, which are often conceived as being on top of a BB, layer access generics are located below the BBs they provide access to. The BB register themselves with the layer access generic (see section 7.2.4). Figure 47 shows import relations of BBs in a lower and a higher layer. Configurable BBs (cBB) of lower layers have no provides interface. They can be replaced without affecting higher layers. Also, changes within the lower layer are less likely to propagate to higher layers. Layer access generics are brokers for layer functionality.

**Example: tss Generics**

The tss system has around 50 generics. We shall mention some of them below without going into much detail.

Examples of Abstraction Generics from the OS subsystem include file handling and I/O device handling. File handling includes the operations open, close, write and read. Specifics of file handling have specialised handlers for different file types such as sequential or indexed files. I/O device handling includes the operations mount, dismount, assign and deassign. I/O device handling has separate specifics for handling different device types such as PCs, printers and disks.

Examples of abstraction generics from the equipment management subsystem include two for handling two classes of peripheral cards. Peripheral card handling recovers and supervises peripheral cards in the central controller. The protocols and data structures are standardised for all cards and they belong to the generic part. Spe-
cific parts are reduced to peripheral card-specific data structures and parameters. For each peripheral card type, there is a separate specific part.

Examples of abstraction generics from the logical resource management subsystem include generics for administering generic classes of analog and digital subscriber lines and trunk lines. Digit analysis and automatic audible responses are other abstraction generics. Digit analysis has as its input digit strings and has specifics for different destination types such as analog or digital subscribers or PABXs. Audible response generic has specifics for single and periodic announcements.

Examples of abstraction generics in the service management subsystem include the call-handling generic which implements basic call handling such as call set-up, call phase and call release, which are valid for (almost) all call types. It can be extended to different signalling handlers. Another generic handles the generic call forwarding feature. Specifics are for call forwarding on absent, on busy or on other conditions.

The equipment management subsystem contains four connectable resource generics for administering hardware (-related) resources. Two generics administer internal bus slots, the so-called time slot (TS) generic and the so-called universal peripheral slot (UPS) generic. External connections are administered by the so-called circuit generic, which is based on a 64kbit communication channel abstraction. A fourth connectable resource generic establishes pools of dynamically allocatable communication channels for a group of subscribers, so-called concentration groups. Figure 48 shows how Equipment Maintenance structuring is based on mirroring of the peripheral hardware structure to a central controller BB structure.

Three generics of equipment management also have the role of a layer access generic. The circuit generic, the TS generic and the concentration group generic function as a layer interface to the higher layers LRM and SM.

The OS contains a number of SIGs. They are persistent data handling, process handling, memory management, user interface presentation data handling, recovery handling and exception handling.

7.6 Interfaces Revisited

After the sections on layers (section 7.4) and generic BBs (section 7.5), we shall take another look at interfaces. In section 7.2 we made a distinction between provides and requires interfaces. From the discussion of incremental layering it is clear that a BB may depend only on BBs in lower layers. As a consequence a
Figure 48: System Structure with HW Mirroring in EM

requires interface relates to BBs in lower layers only and a provides interface relates to higher layers only.

There are, however, two types of interfaces, call-back interfaces and so-called first-access interfaces (see section A.3.3.4), of which it is not so clear whether they are provides or requires interfaces. One may say that a BB requires certain processing to be done by a calling BB which knows the precise content of a data structure or that a BB provides a first-access interface to be used by the recovery manager. Note that in both cases the activation goes from the lower BB to the higher BB. In the first case, a using BB has registered call-back methods, whereas in the second case a standardised entry in a BB header allows the recovery manager to activate initialisation methods.
However, contrary to the use of require and provide suggested above, call-back interfaces are part of the provides interface and the first-access interface is part of the requires interface, because the incrementality of a BBM-based system is the leading concern for the definition of provides and requires interfaces. Even if a BB has only limited processing capabilities in a certain situation, the BB may not depend on the presence of a call-back method. A BB always has to implement some default behaviour if no call-back method is present. Call-back interfaces are in a sense optional.

A similar argument as for call-back methods applies to the first-access methods. A BB requires the recovery manager to call the BB’s initialisation methods to be initialised. Therefore, first-access methods are part of the requires interface of the BB. Figure 49 shows the provides interface at the top and the requires interface at the bottom of a BB to indicate how interfaces support incrementality.

**7.7 Grouping of BBs**

Hierarchical components are components which consist of components themselves. This is a useful means for abstraction. The BBM does not support hierarchical BBs. A BB is a black box, that is, its internal design is not visible, and it is deployed as one unit. Recursive deployment does not seem to be a useful concept.
However, collections of BBs can be grouped to a white box component, that is, a component that does not hide its internals. Examples of such groupings are layers (section 7.4), a feature set (see section 8.3), that is, a set of BBs which implements a certain feature, or managed object related BBs (controlling and controlled BB) (see section 10.2.1). The BBs of a feature set may belong to different subsystems.

**Heuristic 73:** Apart from a BB itself, the collection of BBs of a white box component can be packaged as unit of deployment.

Packaging of BBs into larger deployment sets is described in section 7.9. In a running system only deployment units are recognisable.

**Substitutability**

Two BBs are substitutable if they have the same provides and requires interfaces and if they are fulfilling the same role in the architecture.

However, there may be cases where technical substitutability as defined above is not enough. If both BBs were planned and one of them is not just an improvement of the other, difference in features is involved (see section 8.3.4). Depending on the feature set of the product, one of them will be chosen.

**Heuristic 74:** If two substitutable BBs are to be present in the same product, the BBM requires that there must also be some generic which switches between the two.

## 7.8 Architectural Skeleton

An architectural skeleton is the result of structuring the software in incremental layers which themselves consist of generic and specific BBs. An architectural skeleton is defined as the collection of generic BBs in the different layers. The specific BBs that implement the wanted functionality constitute the "meat". The skeleton, however, provides a structured platform at different levels, which supports the rapid creation of products. It is an incrementally layered component framework. The skeleton implements the stable structure in a domain. It is a generic domain architecture. Abstracting from internal structure, an architectural skeleton is also called a product platform [ML97].
Figure 50: Architectural Skeleton

Figure 50 gives an example of an architectural skeleton together with the sets of specific BBs. The architectural skeleton consists of the shaded BBs only. They are represented in two types of generic BBs. First, there are generics in each of the three layers of the system. Secondly, there are two SIGs in the lowest layer (transparent layering). Note that the relations of the system infrastructure generic with their specifics are not indicated, because potentially all the Building Blocks are specifics of a system infrastructure generic.

Another important point in this example is that the interfaces of the subsystems (or layers) are represented by generics only. These so-called layer access generics are brokers of the layer functionality towards higher layers. They allow configurability within a layer without affecting the higher layers.

Note that the system is incremental in two ways. First, the system can be built subsystem by subsystem. Secondly, BBs can be added in each subsystem.

Heuristic 75: Choose generic BBs in such a way that stability of architectural skeleton increases.
Another important point is that BBM-based systems are always open. A new BB may always be added to generics using them as its relative infrastructure. Such a new BB has a specific role but may also have generic roles. Thus, a completed system can always be extended if it provides interfaces to access its functionality. This can be realised either by removing a BB which has no provides interface and substituting it by one that has a provides interface, or by using the provides interface of a BB and adding new functionality.

A self-describing component is a component that may be introduced into a system without requiring adaptations in other parts in the system.

**Heuristic 76:** Make a BB is a self-describing component by letting it communicate its characteristics such as its resource requirements to the infrastructure.

In particular, the infrastructure should not be adapted because of requirements imposed by a component. In this way the exchange or adaptation of a component does not have any direct consequences for the rest of the system. Aspect completeness (section 5.6.1) is a way to make BBs self-describing.

A system integration approach has a low integration complexity if it can be based on lists of components without requiring adaptations in these components for the sake of integration. In particular, if the infrastructure provides all facilities to make components self-describing, the complexity of resource management decreases considerably because it is separated into a requesting and an administering part. In particular, the requirements of resources are part of the components themselves. The BBM uses such an integration approach (see section 11.7).

---

### 7.9 Deployability Design

Deployability design is about possible deployment scenarios of the products. Address spaces (section 6.1) are also used within deployability design.

The input for deployment scenarios may come from the customer, from product management or from technology assessment requiring a certain HW partitioning. Geographic distribution, if required, will often come directly from the customer.
BBs are the minimal units of deployment. However, for commercial or other reasons, BBs may be packaged to larger sets. Delivery will be in statically or dynamically linked code libraries. Since the library structure is important for the flexibility of software upgrading care should be taken when BBs are packaged.

As described earlier, address spaces are boundaries for objects, BBs and threads (see section 6.1). Address spaces may also be used for fault containment and recovery (see below). Furthermore, the flexibility to move BBs across HW instances is based on the presence of BBs implementing all requires interfaces of the moved BBs at the target location.

Deployability design consists of the following steps:

**Determining fault containment units**

Fault containment units are chosen to confine the consequences of failures and to be able to do a recovery from the failure. Fault containment units should not cross HW boundaries. Design has to take care that a fault containment unit can always synchronise with the rest of the system.

*Heuristic 77: Select a set of objects in such a way that the set may be independently recoverable when an error occurs.*

This may require refactoring in object design.

**Determining possible deployment scenarios**

Deployment scenarios have to take the following factors into account:

- different geographic locations,
- HW partitioning and
- different SW address spaces.

*Heuristic 78: Align BB-, thread-, fault containment unit- boundaries to HW instances*

This may lead to refactoring in the respective design task.
Packaging BBs to deployment sets

BBs are deployment units. They may be packaged for commercial or practical deployment reasons into deployment sets. Product management may package BB, taking into account:

- sellable units (basic product may be one unit), that is, a commercial component may be different from a technical component (visibility to the customer), and
- independent evolvability.

*Heuristic 79:* Package BBs to deployment sets such that independent selling and evolution remains possible.

Deployment sets consist of executables and libraries.

White-box collections of BBs such as subsystems or feature sets (see section 7.7) are examples of useful deployment sets.

Generating data files

Data files are used amongst others for initial or default configuration values, persistent data, international strings, font libraries, sounds, tones and announcements.

The creation of a tss product is described in section A.4.

Heuristics Summary of BB Design

*Heuristic 44:* Cluster objects into BBs such that coupling of objects across BB borders is low and cohesion of objects within a BB is high.

*Heuristic 45:* Cluster objects into a BB which represent a feature.

*Heuristic 46:* Cluster objects into different BBs which belong to independently evolvable parts.

*Heuristic 47:* Cluster objects into BBs such that a BB can be used as a work allocation units for 1 or 2 persons.

*Heuristic 48:* If the variation point lies inside a BB, refactor the BB such that the variation point lies at the border of a BB.
Heuristic 49: Factor out functionality which is present in several BBs in a separate BB.

Heuristic 50: Take as main criterion stability under evolution, that is, an interface should be such that it can serve for those implementations and those usages which are likely to happen.

Heuristic 51: Factor generic implementation parts which are used by several specific parts into a generic BB.

Heuristic 52: Take the implementation of common aspect functionality as a candidate for a SIG.

Heuristic 53: Resolve mutual dependence between BB A and BB B in the follow way: if A is expected to be more stable than B, then make B depend on A; and vice versa if the communication between A and B is expected to be the most stable part, factor the communication out into a new BB and let both, A and B, depend on it.

Heuristic 54: In the case of embedded systems, use importing of interfaces at compile time if needed for performance reasons. Otherwise use dynamic exploration of interfaces for more flexibility.

Heuristic 55: Structure interfaces according to aspects.

Heuristic 56: Use layering for BBs on two levels. Subsystems, which are collections of BBs, are layered. These layers are based on the classification of layers of domain objects done during object design.

Heuristic 57: Individual BBs within subsystems are also layered in relation to other BBs.

Heuristic 58: A common principle for the layering of software is to separate hardware-technology-oriented functionality from application-oriented functionality.

Heuristic 59: Construct layers as virtual machines for higher layers.

Heuristic 60: Another way of introducing layers is to distinguish between generic and specific functionality.

Heuristic 61: The usage of transparent layers is favourable to the usage of opaque ones.
Heuristic 62: Opacity is used for layers that function as facades, such as abstraction layers for hardware, operating system or middleware.

Heuristic 63: Use layers to structure communication in a system.

Heuristic 64: Separate common functionality from specific functionality.

Heuristic 65: Look for the diverse parts in similar functionality for different features.

Heuristic 66: Use inversion of control for designing the functionality of a generic BBs.

Heuristic 67: A generic BB is stable if new specific functionalities may be based on the generic BB without changing it.

Heuristic 68: Use an abstraction generic to implement an abstract concept which is to be extended by specific BBs.

Heuristic 69: Use a connectable resource generic to manage connectable resources which are supplied by HW boards.

Heuristic 70: Design a system infrastructure generic for functionality which provides an operating infrastructure for almost all application BBs.

Heuristic 71: System Infrastructure Generics must provide interfaces for application BBs for indicating their resource requirements.

Heuristic 72: Design a layer access generic to restrict the visibility of the structure of a layer for higher layers.

Heuristic 73: Apart from a BB itself, the collection of BBs of a white box component can be packaged as unit of deployment.

Heuristic 74: If two substitutable BBs are to be present in the same product, the BBM requires that there must also be some generic which switches between the two.

Heuristic 75: Choose generic BBs in such a way that stability of architectural skeleton increases.

Heuristic 76: Make a BB is a self-describing component by letting it communicate its characteristics such as its resource requirements to the infrastructure.
Heuristics Overview of Deployability Design

*Heuristic 77:* Select a set of objects in such a way that the set may be independently recoverable when an error occurs.

*Heuristic 78:* Align BB-, thread-, fault containment unit- boundaries to HW instances

*Heuristic 79:* Package BBs to deployment sets such that independent selling and evolution remains possible.