4 Object Design

Object design is about the design in the object dimension. Software objects, as defined by object-oriented modelling, encapsulate data and operations on these data. Objects are connected to other objects through relations.

Objects are used in the context of the BBM at four different levels. *Application domain objects* describe entities of the application domain. *Hardware domain objects* describe elements of the hardware system. They are both part of the first level. On the second level we have *domain-induced objects*. They are a replication of the objects of the first level into the software design space. They are generated from inputs of the BBM. On the third level we have *design objects*. They are refined and refactored due to the design tasks of the BBM. On the fourth level we have *implementation or programming language objects*. They are mostly a mirroring of design objects. However, specific implementations may use objects, which are more fine-grained.

Object design uses the application domain model defined by application domain modelling (see section 2.6.2) as input. The application domain model (first level) is used to derive an internal software model. There are several sources for objects from outside of the software, which will be described in the first section. The other design tasks, aspect design, concurrency design, composability design and deployability design, also lead to a refinement of the object structure (from second to third level). The second section describes those design objects.

4.1 Domain-Induced Objects

There are several sources for identification of objects outside of the software. The most obvious one is the application domain consisting of the application itself and the operational context of the application. Others are induced by sys-
tem qualities and by hardware. We call them domain-induced objects. We describe the design steps in detail.

### 4.1.1 Domain Object Model

The main source for objects is the application domain object model. In a first step we mirror the external application domain model into the internal software model. The initial decomposition of the functionality is the one obtained from the application domain.

**Heuristic 1:** Use application domain objects and relations to generate an initial object model of the product family by mirroring them in the software system.

Figure 24 shows the mirroring of the domain model into the systems software.

![Figure 24: Mapping of Domain Model to Software](image)

Note that this step is often done implicitly in other methods.

Mirroring the domain object model also mirrors the object interactions. External actors will interact with their mirrored domain objects and with other objects described by the object interactions. Object interactions form the basis for behavioural modelling of the systems.

Inheritance, often seen as an essential feature in object-oriented modelling, may be used during the modelling of domain objects (see section 2.6.2). The inheritance relation is one of the relations between objects. Inheritance relations are transformed during object design into object composition relations. The sub-
class is explicitly calling the superclass for functionality it needs, while the superclass is delegating calls explicitly to its subclass. Object composition relations are easier to handle between different BBs because of explicit interfacing. A further reason for transforming inheritance relations is that inheritance at the programming language level is often only a compile-time concept, whereas BBs are deployment units [Szy98]. We shall not discuss inheritance any further (see for instance [RBP*91] or [Szy98]).

4.1.2 System-Relevant Functionality

The internal software model contains only the functionality, which is relevant for the system to be built. Remember that the application domain model may contain more than is actually required by the systems to be built. Therefore the required functionality has to be selected from the application domain model.

Heuristic 2: Remove objects, attributes and relations which do not describe required system functionality.

The issue is the relation between the domain model and the precise requirements for the systems to built. For instance, the system may only do a certain kind of processing whereas the domain model is wider in scope. For instance, the system may have a control perspective or a recording perspective with respect to their real world counterparts

Heuristic 3: Adapt the functionality of domain-induced objects to the required perspective of the system.

This adaptation may take only parts of objects and leave other objects completely outside of the system.

Sometimes application domain modelling does already adapt its model to be an internal software model. That means that the domain model reflects the actual requirements. Such a domain model looses some of its power since it will not be stable under changing requirements. The reason why domain modelling as a separate activity was introduced was to achieve exactly such a stability by being independent from requirements for a specific system.

During aspect design we will execute an architectural concern analysis (see section 5.2) which may lead to the identification of functionality not identified in the domain model.
4.1.3 System Qualities

Besides the functionality defined in the application domain model, additional functionality may be necessary to achieve the required system qualities. During aspect design an architectural concern analysis and an analysis of quality specifications (see section 5.2) will be performed. This analysis may result in extended system functionality and lead to additional objects.

Examples of objects induced by system qualities are a database to save persistent values to survive system crashes and an encryption package to achieve security of communicated data. High availability may be supported by an administration of loadable modules which allows to upgrade a system and to fall-back in case of failure.

4.1.4 Hardware-Implemented Functionality

The next step is to factor out functionality which is implemented in HW.

An example is the processing of signals which may be completely implemented in HW.

The integration of different types of hardware into the system may lead to a distributed HW architecture. Software then has to be distributed over different HW instances (see section 3.2.5).

4.1.5 Hardware-Managing Objects

Specific processing HW has to be handled by software. Specific HW-managing objects are introduced to manage the HW functionality. Flexible HW boards will also need a specific object to manage the type and state of the board.

Heuristic 4: Create one object per replaceable HW unit.

On a standard computer platform this is already done for standard devices in the operating system. Also communication interfaces and channels are usually handled by the operating system. If there is no operating system or it handles only part of HW, the remaining HW resource handling has to be modelled in new objects.
In figure 25 external sources for domain-induced objects are shown.

\[ \text{software functionality} \]

- domain functionality
- application objects
- infrastructure objects
- hardware objects

**Figure 25: Examples of Sources of Objects**

### 4.2 Design Objects

Domain-induced objects are refactored into design objects by the BBM design tasks. First we describe the refactoring to place design objects in layers (section 4.2.1). There are a number of cases for refactoring the object structure as a result of aspect design, concurrency design, composability design or deployability design. We shall describe examples of refactoring below.

More examples of refactoring are described in [DMN*97], where for each axis of variability one additional object is recommended to hide the variations.

#### 4.2.1 Classification of Functionality in Layers

Layering is very common in modelling functionality of large software-intensive systems. Layering need not be inherent in the functionality but is a way of introducing structure. The purpose of layering is to achieve separation of concerns and management of complexity. Having a layer for a certain kind of abstraction guides the identification of similar abstractions throughout the design.

Layers are specifically introduced to achieve portability and ease of evolution. Interface abstractions between two layers are chosen such that certain functionality can be executed on different hardware or operating systems platforms.

Layering is extensively used in the design of electronic systems. The main reason is to separate the concerns of hardware handling and of application functionality. Hardware technology and application domain functionality have different evolution speed. The functionality realised by the hardware is part of the application domain and
evolves with the application domain. The hardware technology changes faster than the application domain functionality. Hardware handling software abstracts from hardware specifics to abstract concepts on which the application functionality is based. The selection of functionality, its partitioning and its implementation technology change over time. The abstract nature of software makes the coupling of application functionality and solution technology a loose one.

Layers should be decided on from an engineering perspective rather than purely on the basis of the logical nature of the used abstractions, that is, a layer is a means to deal with a large amount of functionality. This means that the precise number and nature of the layers are not fixed, but are subject to system evolution, i.e. extending the functionality may necessitate the introduction of additional layers to handle complexity.

In the following we describe a possible rationale for the introduction of several coarse layers of objects.

Two initial classes of software are obtained by factoring out supporting functionality from application functionality. Objects are put in layers such that a layer may only use a lower layer but not vice versa. The supporting layer is called infrastructure and will be refined later. Two layers are shown in figure 26. An example is libraries for communication between application objects.

<table>
<thead>
<tr>
<th>Application Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure Objects</td>
</tr>
</tbody>
</table>

*Figure 26: Initial Two Layers*

**Heuristic 5:** Refactor domain-induced objects to objects of an application layer and an infrastructure layer.

On the basis of these two layers, which contain objects representing the application functionality and support functionality, additional layers may be appropriate.
A further layer may be appropriate if application functionality extends significantly. An application-specific platform encapsulates basic application abstractions. Various advanced applications make use of this platform. (figure 27).

![Three Layers with Basic and Advanced Applications](image)

**Figure 27: Three Layers with Basic and Advanced Applications**

**Heuristic 6:** Refactor large collections of application objects to objects of a basic application layer and an advanced application layer.

Infrastructure functionality such as the operating system and other infrastructure services which should be used by all the application objects may be grouped in again a different layer, i.e. the lowest layer (figure 28) In such a case the application infrastructure is often called middleware.

![Four Layers with Operating Infrastructure](image)

**Figure 28: Four Layers with Operating Infrastructure**

**Heuristic 7:** Design objects which will be implemented by an operating system layer independent from an additional middleware layer.

Layering refactors domain-induced objects. The design objects are arranged within the layers and similar abstractions are used per layer. Basic application objects are separated from advanced application objects and HW managing objects are separated from application objects.
In this section, a basic call object deals with the originator of the call, the dialled number, the destination indicator and the call state. Advanced call objects are call objects containing various features such as follow-me and automatic ring-back on busy.

### 4.2.2 Communication Objects

To communicate between threads and processes, communication objects are introduced. There are three types.

*Heuristic 8:* Design messages which are sent between threads and processes in separate objects.

*Heuristic 9:* Design objects which hold message objects such as mailboxes, buffers, queues as separate objects.

*Heuristic 10:* Design protocol implementations as objects.

### 4.2.3 Interface Objects

Designing BBs to be independent of each other leads to extra interface objects.

*Heuristic 11:* Group interfaces of several domain-induced objects to one interface abstraction.

*Heuristic 12:* Limit the visibility of attributes and operations of domain-induced objects behind interface objects.

### 4.2.4 Registry Objects

In designing for configurability in a product family, domain objects may represent different alternative or parallel features. Domain objects representing the variable functionality must be able to register themselves to some registry object to achieve configurability of BBs.

*Heuristic 13:* Model registration functionality as a separate design object.

If the variation is in an algorithm to be configured, the strategy pattern [GHJ*94] may be used for the implementation.

Alternatively, a design object containing the common part of the variation may be extended to function as a registry. Then, variation objects must register themselves to it. However, a separate registry object is preferable because it can be used for all registrations.
4.2.5 Container Objects

Handling all or many instances of a class in a similar way can either be implemented as part of the class functionality, or it can be done via container objects such as lists, queue, etc.

The advantage of container objects is that the uniform functions are implemented by the container. The container then holds objects that need not be uniform. This allows for easier evolution.

*Heuristic 14: Use container objects for explicitly handling instances of a class in lists and queues.*

4.2.6 Functional Objects and Object Functions

The design for an aspect may be expressed in terms of aspect-specific objects. How do aspect-specific objects then relate to design objects? There are two ways of dealing with aspect-specific objects:

- aspect-specific objects may be transformed into common attributes and methods of all related design objects, or
- aspect-specific objects are new design objects.

Which alternative should be used depends on the amount of functionality which is modelled. The first alternative may be used for aspects with small functionality, while the second gives more structure for large aspects having separate data structures. The second possibility is also more natural with respect to object-oriented programming (see chapter 5).

*Heuristic 15: Use separate objects to model aspects with large amount of functionality.*

A further question is whether data can be separated per aspect or whether data is global to all aspects. Experience indicates that a domain object may be reduced to a minimal design object which represents its identity and state. This minimal object is accessed from all aspects. All other data is modelled in design objects belonging to one or more aspects. The domain objects are then represented by a number of design objects. The above mentioned reduced domain objects consisting of identity and state only can be viewed as a separate data aspect. This will not be elaborated any further here because the practical consequences are limited.
4.3 Example: Layers and Some Objects of tss

The central controller SW of the tss system (see section A.3) is based on layers similar to those that have been developed over the years for most telecommunication infrastructure systems (for example [KBP*95]). These coarse layers are major chunks of functionality and are therefore called layered subsystems. Typical layers are the extended operating system, equipment maintenance, logical resource management and service management. Figure 29 shows the four tss layered subsystems. We describe typical objects within the layers.

The service management subsystem (SM) comprises all the services of the application. Its main purpose is the provision of the system's intended functionality, that is, call signalling (heuristic 10) and call facilities.

The logical resource management (LRM) subsystem manages the data resources for the higher-layer subsystem. The LRM subsystem deals with data for signalling, lines (e.g., analog/digital subscriber line, basic and primary access, trunk lines) and facility data (e.g. call forwarding, follow me).

The separation into SM and LRM results from an application of heuristic 5. Internally, SM is further separated according to heuristic 6. The data-oriented objects of LRM and the service-oriented objects of SM result from the application of heuristic 1, heuristic 2 and heuristic 3.

The equipment maintenance (EM) subsystem consists of the control layer for the peripheral hardware and its interconnection structure, as controlled by the central controller. It deals with aspects of e.g. recovery and fault management of controlled equipment, and data distribution to the controlled equipment. All the instances of equipment are domain objects (heuristic 4). An abstract 64 Kilobit/s channel abstrac-
tion serves as a registry and interface object for different line types of LRM (heuristic 13).

The extended operating system (EOS) (heuristic 7) comprises, for instance, process handling, timer services, exception handling, data base, recovery mechanisms, administration of BB executables, file handling, memory management.

For more information about the SW architecture of tss see section A.3.

4.4 Explicit Transition

Objects are a useful concept for the execution of different tasks. Application domain modelling often uses objects to describe key domain concepts and their relations. Object-oriented design makes use of objects. Object-oriented analysis also uses objects, often in a mixture of application domain modelling and high-level design. And finally, the implementation is done via object-oriented programming.

In spite of the fact that in all the tasks of the architecting model (see section 2.6) the concept of an object is used, these tasks have a quite different character. The result is therefore that the semantics of an object is different in these tasks. For instance, in application domain modelling an object describes an entity of the application domain such as a system, or an instrument, or thing or a human being. It is not interpreted as a computational entity having state and operations such as in the case of object-oriented programming.

The object design task of the BBM makes the transition between objects in these different tasks explicit. Application domain objects are identified outside of the BBM and used as input for architectural design. Domain-induced objects are a replication of application and technology domain objects in the software design space. Aspects are identified as orthogonal functionalities to domain-induced objects. The core of the object design tasks consists of the transformation of domain-induced objects into design objects. Implementation objects are a refinement of design objects.

Heuristics Overview

Heuristic 1: Use application domain objects and relations to generate an initial object model of the product family by mirroring them in the software system.
Heuristic 2: Remove objects, attributes and relations which do not describe required system functionality.

Heuristic 3: Adapt the functionality of domain-induced objects to the required perspective of the system.

Heuristic 4: Create one object per replaceable HW unit.

Heuristic 5: Refactor domain-induced objects to objects of an application layer and an infrastructure layer.

Heuristic 6: Refactor large collections of application objects to objects of a basic application layer and an advanced application layer.

Heuristic 7: Design objects which will be implemented by an operating system layer independent from an additional middleware layer.

Heuristic 8: Design messages which are sent between threads and processes in separate objects.

Heuristic 9: Design objects which hold message objects such as mailboxes, buffers, queues as separate objects.

Heuristic 10: Design protocol implementations as objects.

Heuristic 11: Group interfaces of several domain-induced objects to one interface abstraction.

Heuristic 12: Limit the visibility of attributes and operations of domain-induced objects behind interface objects.

Heuristic 13: Model registration functionality as a separate design object.

Heuristic 14: Use container objects for explicitly handling instances of a class in lists and queues.

Heuristic 15: Use separate objects to model aspects with large amount of functionality.