Dealing with the complexity of:

Introduction into Software Engineering

Prof. Bernd Brügge, Ph.D. (TU München, SS 2010)

This summary contains the main topics of course *Introduction into Software Engineering* given by Prof. Bernd Brügge, Ph.D. (SS2010, TU München). Text and figures are based on the book *Object-Oriented Software Engineering* (Brügge, Dutoit, 3rd ed., 2010) and the slides.

Included are lectures 5-20, except for a detailed view into OCL. Please report any fault, failure or erroneous state to e.steger at mytum.de.

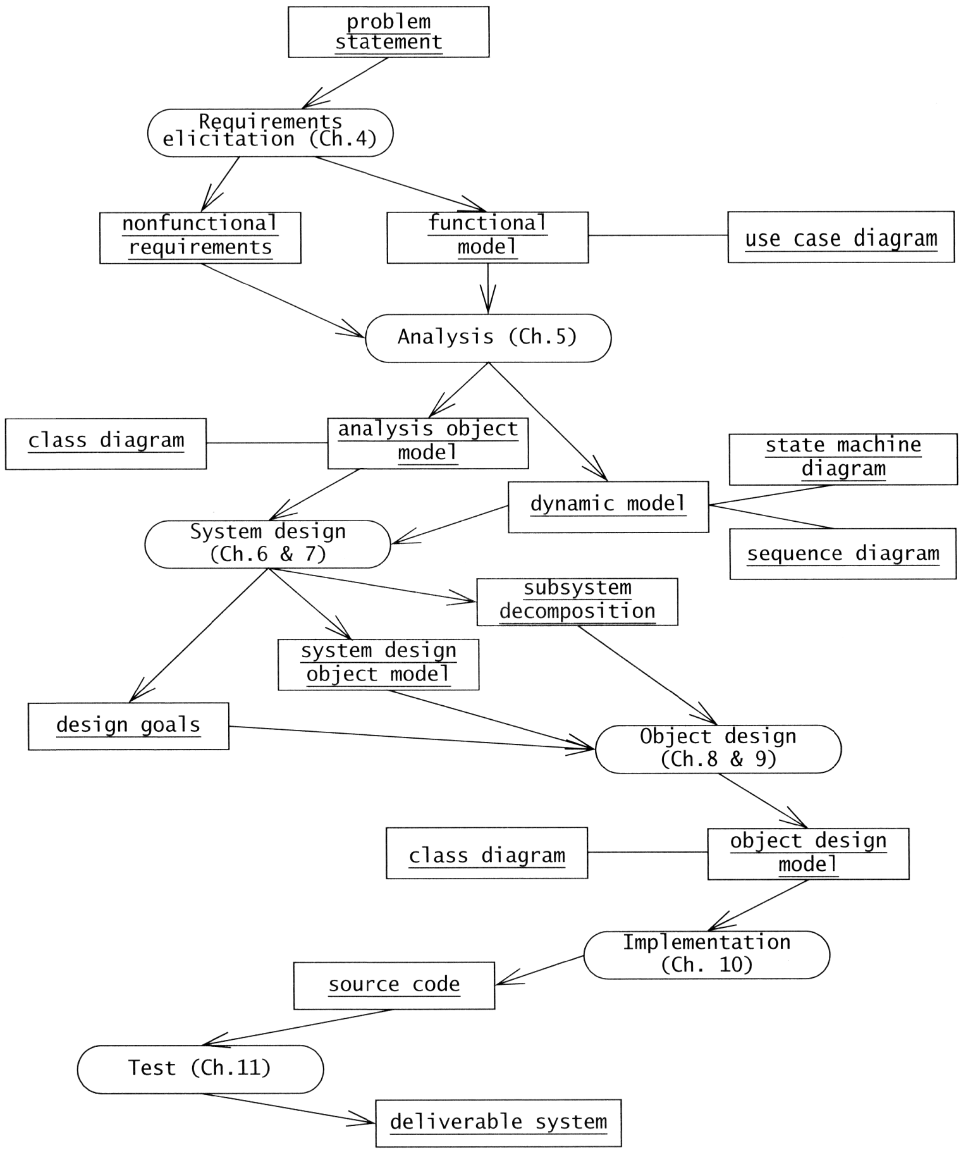
Emanuel Steger

V 2011-05-08 (Spelling mistakes fixed)

Contents

0. Overview of Activities

The diagram below depicts an overview of Object-Oriented Software Engineering (OOSE) development activities and their products. Since OOSE is iterative, these activities can occur in parallel and more than once.



1. Requirements Elicitation

During requirements elicitation, the client and developers define the *purpose of the system.* The main activities are *identifying actors and scenarios.* Developers observe users and create a set of detailed scenarios for typical functionality the future system needs to provide.

A scenario is a concrete example of the future system in use, it includes a series of interactions between the user and the system.

Once developers and users agree on a set of scenarios, developers derive from these scenarios a set of use cases that completely represent the future system.

A use case is an abstraction that describes all the possible actions between an actor and the system.

The result of requirements elicitation is a definition of the system, written in natural language that the client understands. Such a definition is called **requirements specification**.

The requirements specification is structured and formalized during **analysis** to produce an **analysis model**, also called *technical specification.* The analysis model uses a (semi-)formal notation like UML. The **requirements process** is the combination of requirements specification and analysis.

1.1. Functional Requirements

Functional requirements describe the interactions between the system and its environment independent of its implementation. The environment includes the user and any other external system with which the system interacts.

Example: “An operator must be able to define a new game.”

1.2. Non-functional Requirements

Non-functional requirements describe aspects of the system that are not directly related to the functional behavior of the system. They can be categorized into **quality requirements** and **pseudo requirements / constraints**.

1.2.1. Quality Requirements

- **Usability** is the ease with which actors can perform a system function (measurable, e.g. "number of steps needed to purchase a product").
- **Dependability** includes:
  - **Reliability:** The ability to perform the required function under stated conditions for a specified period of time.
  - **Robustness:** The ability to maintain a function when the user enters a wrong input or when there are changes in the environment.
  - **Availability:** The percentage of time that the system can be used.
  - **Fault tolerance:** The ability to operate under erroneous conditions.
  - **Security:** The ability to withstand malicious attacks.
  - **Safety:** A measure of the absence of catastrophic consequences to the environment.
- **Performance** concerns with quantifiable attributes of the system, such as **response time**, **throughput**, **availability** and **accuracy**.
- **Supportability** concerns with the ease of changes to the system after deployment, it includes:
  - **Adaptability:** The ability to change the system to deal with additional application domain concepts.
  - **Maintainability:** The ability to change a system to deal with new technology/conventions or to fix defects (including **extensibility** and **modifiability**).
  - **Portability:** The ease with which a system can be transfered from one hardware or software environment to another.

1.2.2. Pseudo Requirements / Constraints

- **Implementation** requirements are constraints on the implementation of the system, including the use of the specific tools, programming languages, or hardware platforms.
- **Interface** requirements are constraints imposed by external systems, including legacy systems (older systems) and interchange formats.
- **Operations** requirements are constraints on the administration and management of the system in the operating setting.
- **Packaging** requirements are constraints on the actual delivery of the system (e.g. "system must be delivered on floppy disks").
- **Legal** requirements are concerned with licensing, regulation etc.

1.3. Requirements Validation

Validation is an activity that ensures that the system addresses the client's needs (we are building the right system). In contrast, **verification** attempts to detect faults (makes sure that we are building the system right).

The requirements specification is:

- **complete** if all possible scenarios through the system are described, including exceptional behavior,
- **consistent** if it does not contradict itself,
- **unambiguous** if exactly one system is defined (it is not possible to interpret the specification two or more different ways, "clarity"),
- **correct** if it represents accurately the system that the client needs and that the developers intend to build.

Furthermore the requirements specification is:

- **realistic** if the system can be implemented within constraints,
- **verifiable** if, once the system is built, repeatable tests can be designed to demonstrate that the system fulfills the requirements specification,
- **traceable** if each requirement can be traced throughout the software development to its corresponding system functions, and if each system function can be traced back to its corresponding set of requirements.

1.4. Types of Engineering

Requirements elicitation activities can be classified into three categories:

Greenfield engineering is the development started from scratch (no prior system exists), so the requirements are extracted from the users and the client.

Reengineering is the redesign and reimplementation of an existing system triggered by technology enablers or by business processes. Sometimes, the functionality of the new system is extended, but the essential purpose remains the same. The requirements are extracted from the existing system.

Interface engineering is the redesign and reimplementation of the user interface of an existing system. The rest of the legacy system is left untouched.

1.5. Activities

Identifying Actors

The first step of requirements elicitation is the identification of actors (external entities that interact with the system). This serves both, to define boundaries of the system and to find all the perspectives from which the developers need to consider the system. Actors are role abstractions and do not necessarily map to certain persons. And the same person can fill more than one role at different times.

During the initial stages, it is hard to distinguish actors from objects (e.g. a database subsystem that can at times be an actor, in other cases part of the system). At this point, we assume that any external software system using the system to be developed, is an actor.

Questions for identifying actors can be:

- Which user groups execute the system?
- With what external hardware or software will the system interact?

Once the actors are identified, the next step is to determine the functionality that will be accessible to each actor, that means identifying scenarios.

Identifying Scenarios

A scenario is a concrete, focused, informal description of a single feature of the system from the viewpoint of a single actor. In particular, scenarios cannot contain descriptions of decisions (example see Table 1).

Scenario name	`warehouseOnFire`
Participating actor instances	`bob, alice:FieldOfficer`; `john:Dispatcher`
Flow of events	1. Bob notices smoke coming out of a warehouse. Alice activates the `Report Emergency` function... 2. Alice enters the address of the building and a brief description of its location. 3. John is alerted to the emergency, reviews the information submitted by Alice and allocates a fire unit. 4. Alice receives the acknowledgment and the ETAs.

Table 1: An example scenario for a use case.

Questions for identifying scenarios can be:

- What are the tasks that the actor wants the system to perform?
- Which external changes does the actor need to inform the system about?
- Which events does the system need to inform the actor about?

Once developers have identified and described actors and scenarios, they formalize scenarios into use cases.

Identifying Use Cases

A use cases specifies all possible scenarios for a given piece of functionality and is initiated by an actor (example see Table 2). After initiation, a use case may interact with other actors, as well. Generalizing scenarios and identifying use cases that the system must support, enables developers to define the scope of the system. By examining the entry and exit conditions of use cases, developers can determine if there may be missing use cases.

Use case name	`ReportEmergency`
Participating actors	Initiated by `FieldOfficer` Communicates with `Dispatcher`
Flow of events	1. The `FieldOfficer` activates the "Report Emergency" function of her terminal. 2. `FRIEND` presents a form to the `FieldOfficer` 3. The `FieldOfficer` fills out the form by... 4. `FRIEND` receives the form and notifies the Dispatcher. 5. The `Dispatcher` reviews the submitted information... 6. `FRIEND` displays the acknowledgment and selected response to the `FieldOfficer`.
Entry condition	The `FieldOfficer` is logged into `FRIEND`
Exit condition	The `FieldOfficer` has received an acknowledgment and the selected response form the `Dispatcher`, OR the `FieldOfficer` has received an explanation indicating why the transaction could not be processed.
Quality requirements	The `FieldOfficer`'s report is acknowledged in 30 seconds.

Table 2: An example of a use case. Steps 2, 4 and 6 are related to the system.

Heuristics for identifying use cases are:

- The name of a use case should be a verb phrase denoting what the actor is trying to accomplish (e.g. "Report Emergency" - perspective of the actor).
- Actors should be named with noun phrases (e.g. `FieldOfficer`).
- The boundary of the system should be clear
(distinguish between steps of the actor and the system).
- Exceptions should be described separately.

Refining Use Cases

The focus of this activity is on completeness and correctness. Developers identify functionality, not covered by scenarios and refine or rewrite the use case. This includes for example, specifying *access rights* or *exceptional handling.*

Identifying Relationships among Actors and Use Cases

Communication relationships between actors and use cases (solid lines) represent the flow of information during the use case. The «`initiate`» stereotype denotes the initiation of the use case by an actor, the «`participate`» stereotype denotes that an actor (who did not initiate the use case) communicates with the use case.

Redundancies among use cases can be factored out using «`include`» relationships (dashed line with an arrow pointing at the use case that is included). Usually, included use cases are common system functions that can be used in several places. Each including use case (e.g. `OpenIncident`) must specify where (within the flow of events) the included use case (e.g. `ViewMap`) should be invoked.

Conversely, a behavior that can happen anytime should be represented as «`extend`» relationship. Exceptional behavior can more easily be specified as an entry condition of the extended use case (e.g. `ConnectionDown`) than at any point of the base use case (`ViewMap`). Moreover, additional exceptional situations can be added without modifying the base use case.

Identifying Initial Analysis Objects

Often, users (or clients) and developers use different terminologies, or the same terms in different context. To establish a clear terminology, developers identify *participating objects* for each use case.While Identifying/naming objects, developers should always use application domain terms. Usually, they are looking for real-world entities (e.g. `FieldOfficer`) and processes (e.g. `EmergencyOperationsPlan`) that the system must track, use cases (e.g. `ReportEmergency`) or artifacts with which the user interacts (e.g. `Station`). Identifying objects in more detail is done during analysis.

2. Analysis

Analysis results in a model of the system, describing the *application domain*, that the developers can unambiguously interpret. The **application domain** represents all aspects of the user's problem. However, the **solution domain** is the modeling space of all possible systems. Modeling in the solution domain represents the system design and object design activities.

In the same sense, objects identified during *analysis* and (system/object) *design* are distinguished: **Application objects**, also called *domain objects*, represent concepts of the domain that are relevant to the system. **Solution objects** represent components that do not have a counterpart in the application domain (*not* identified during analysis), such as persistent data stores, user interface objects or middleware.

The analysis model further aims to be correct, complete, consistent and verifiable. Requirements elicitation and analysis are iterative and incremental activities that occur concurrently.

The **analysis model** is composed of three individual models (see Figure 1):

- The **functional model** is represented by **use cases** and **scenarios**. Both are usually identified during requirements elicitation and refined during analysis.
- The **analysis object model** focuses on the structure of the system and the individual concepts that are manipulated by the system, their properties and relationships. It is represented by **class** and **object diagrams**, including associations (and multiplicities), their attributes and operations.
- The **dynamic model** focuses on the behavior of the system. There are two types of **interaction diagrams**:
  - **Sequence diagrams** represent the interactions among a set of objects during a single use case over time.
  - **Communication diagrams** depict the same information as sequence diagrams by numbering the interactions (but do not show the time). They help identifying senders and receivers, and event dependencies and concurrency.
  - **State machine diagrams** represent the behavior of a single object (or a group of very tightly coupled objects).

  The dynamic model serves to assign responsibilities to individual classes and to identify new classes, associations and attributes to be added to the analysis object model.

It is essential to remember that these models present user-level concepts, not actual software classes or components. For example, classes such as `Database`, `Subsystem` or `SessionManager` should not appear in the analysis model (see Table 3 for more examples). Most of the classes in the analysis object model will correspond to one or more software classes in the source code (with many more attributes and associations).

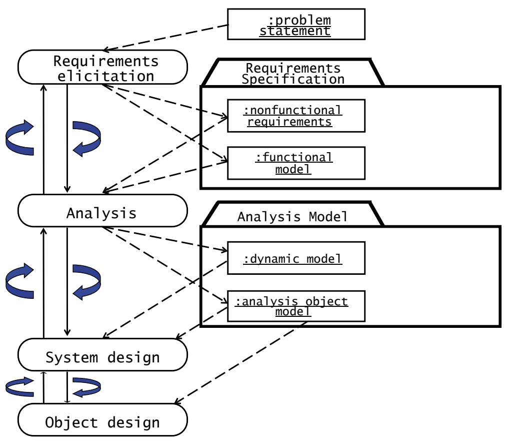


Figure 1: Activities and products during analysis and design.

Domain concepts, that should be represented	**Software classes, that should not be represented**
`UniversalTime`	`TimeZoneDatabase` (Refers to how time zones are stored.)
`TimeZone`	`GpsLocator` (Denotes to how location is measured.)
`Location`	`UserId` (Refers to internal id.-mechanism.)

Table 3: Good and bad examples for analysis class names.

2.1. Types of Objects

Within the analysis object model, we distinguish between three different types of objects (see Figure 2):

- **Entity objects** represent the persistent data tracked by the system.
- **Boundary objects** represent the actors-system-interactions.
- **Control objects** are in charge of realizing use cases.

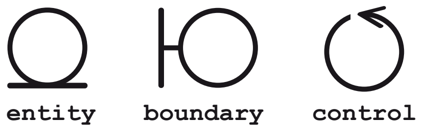


Figure 2: Icons for entity, boundary and control objects.

This distinction leads to models that are more resilient to change. By separating the system into interface (boundary objects) and basic functionality (entity and control objects), we are able to keep most of a model untouched, when for example, only the user interface changes.

Modeling in UML, we can attach stereotypes (i.e. «`entity`», «`boundary`» and «`control`») to objects. In addition, we may also use naming conventions for clarity. Control objects may have the suffix `Control` and boundary objects matching suffixes like `Form`, `Button`, `Display` or even `Boundary` appended to their name.

2.2. Inheritance

Inheritance enables us to organize concepts into hierarchies.

- **Generalization** identifies abstract concepts from lower-level ones.
- **Specialization** identifies more specific concepts from a higher-level one.

2.3. Activities

2.3.1. Object Modeling

While identifying objects/classes we should not forget that objects are inside and actors are outside the system boundary. Picking up a use case (flow of events) and using Abbott's heuristics (see Table 4), we can map parts of speech to model components. This seems to be a very intuitive way of finding objects, but it should be mentioned that first, the quality of the object model depends highly on the style of writing of the analyst (e.g. consistency of terms used) and second, there are many more nouns than relevant classes.

Part of speech	**Model component**	**Example**
Proper noun	Instance	Alice
Common noun	Class	Field officer
Doing verb	Operation	Creates, submits, selects
Being verb	Inheritance	Is a kind of
Having verb	Aggregation	Has, consists of
Modal verb	Constraints	Must be
Adjective	Attribute	Incident description

Table 4: Abbott's heuristics.

Identifying Entity Objects

Participating objects form the basis of the analysis model. Identifying them means looking for:

- *real-world entities* the system needs to track (e.g. `FieldOfficer`),
- *real-world activities* the system needs to track (e.g. `EmergencyOperationPlan`),
- terms that developers/users need to clarify in order to understand the use case,
- *recurring nouns* in use cases (e.g. `Incident`) and
- *data sources* or *sinks* (e.g. `Printer`).

Identifying Boundary Objects

Boundary objects represent the system interface, designed for actors. In each use case, an actor interacts with at least one boundary object which collects information from the actor and translates it into a form that can be used by entity and control objects.

Heuristics for identifying boundary objects are:

- Identify *user interface controls* that the user needs to initiate the use case (e.g. `ReportEmergencyButton`).
- Identify *forms* the users need to enter data into the system (e.g. `ReportEmergencyForm`).
- Identify *notices and messages* the system uses to respond to the user (e.g. `AcknowledgmentNotice`).
- Identify *actor terminals* if multiple actors are involved (e.g. `DispatcherStation`).

Identifying Control Objects

Control objects are responsible for coordinating boundary and entity objects. They usually do not have a concrete counterpart in the real world. A control object is usually created at the beginning of a use case and is responsible for collecting information from the boundary objects and dispatching it to entity objects.

Heuristics for identifying control objects are:

- Identify one control object *per actor* in the use case.
- The life span of a control object should cover the extend of the use case.

Identifying Associations

Making relationships between objects explicit clarifies the analysis model. It enables developers to discover boundary cases associated with links. For example, it is intuitive to assume that a `EmergencyReport` is written by a `FieldOfficer` but should the system support `EmergencyReports` written by more than one?

Associations have several properties:

- a **name** that describes the association
  (attached to the line that links both classes - e.g. `writes`),
- a **role** at each end, identifying the function of each class
  (e.g. `author` and `document`) and
- a **multiplicity** at each end, identifying the possible number of instances
  (e.g. `*` attached to `EmergencyReport`-class indicates a `FieldOfficer` may write zero ore more `EmergencyReports`).

Identifying Aggregates

There are two types of aggregation:

- **composition aggregation**
  (the existence of the parts depends on the whole - depicted by solid diamond),
- **shared aggregation**
  (the whole and the part can exist independently - depicted by hollow diamond).

Identifying Attributes

Attributes are properties of individual objects, consisting of a **name** and a **type**. When identifying properties of an object, only attributes relevant to the system should be considered. Properties that are (represented by) objects are not attributes and should be represented by associations.

2.3.2. Dynamic Modeling

The dynamic model describes the components of the system that have interesting behavoir. **Interaction diagrams** describe the communication among a set of interacting objects (via messages). **State machine diagrams** describe the sequence of states, an object goes through in response to external events.

Mapping Use Cases to Objects with Interaction Diagrams

There are two types of UML interaction diagrams: *sequence diagrams* and *communication diagrams.*

A sequence diagram ties use cases with objects. It shows how the behavior of a use case is distributed among its participating objects and helps finding missing objects. Good heuristics for drawing a sequence diagram are:

- the first column is the *actor* and
- the second column is the *boundary object*
  (that the actor used to initiate the use case).
- The third column then is a *control object* that manages the rest of the use case (e.g. interacts with other control objects).
- Control objects are created by boundary objects
  (e.g. `ReportEmergencyButton` creates `ReportEmergencyControl`) and
- boundary objects are created by control objects
  (e.g. `ReportEmergencyControl` creates `ReportEmergencyForm`).
- Entity objects are accessed by control a boundary objects and
  never access boundary or control objects.

The time span when an object can receive messages is represented by a dashed line. Once it has received a message, this kind of activation is represented by a rectangle from which other messages can originate. The length of the rectangle represents the time the operation is active.

We distinguish between two structures for sequence diagrams (see Figure 3):

- Within a **fork diagram**, the dynamic behavior is placed in a *single object* (usually control object) that knows all the other objects.
  Choose this *centralized* control structure, if:
  - the operations can change order and
  - new operations are expected to be added as a result of new requirements.
- The **stair diagram** distributes the dynamic behavior, each object delegates responsibility to another one.
  Choose this *decentralized* control structure, if:
  - the operations have a strong connection and
  - the operations are always be performed in the same order.

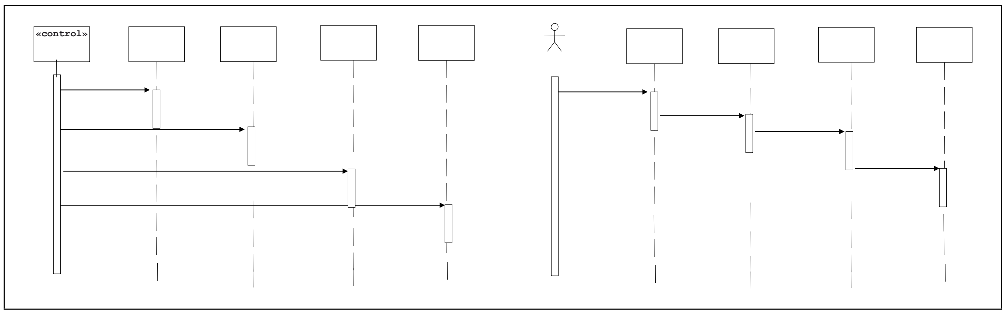


Figure 3: Comparison of fork and stair diagram.

A communication diagram is based on a usual class/object diagram. Figure 4 depicts one example of a sequence and communication diagram.

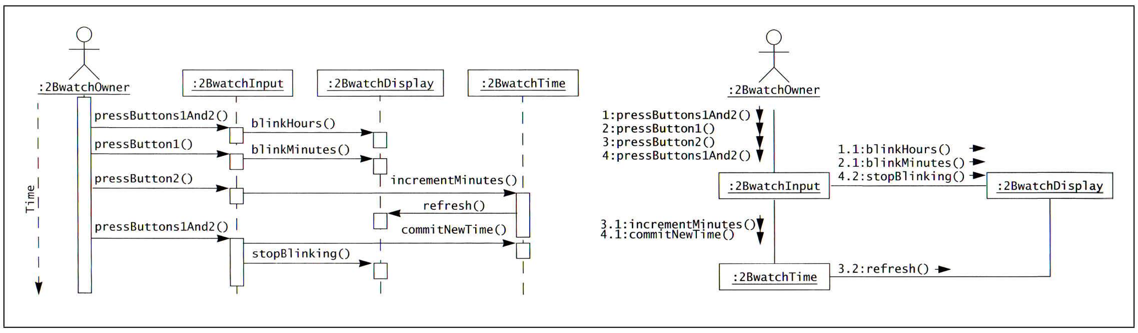


Figure 4: Comparison of sequence and communication diagram.

A communication diagram visualizes the interactions between objects as a flow of messages (i.e. *events* or *calls* to operations). It describes the static structure as well as the dynamic behavior.

Modeling State-Dependent Behavior of Individual Objects

Sequence diagrams represent the behavior of the system from the perspective of a single use case, whereas **state machine diagrams** represent behavior from the *perspective of a single object* and help finding missing use cases. They show the state transitions for the dynamically interesting objects (see Figure 5). Only objects with an extended lifespan and state-dependent behavior are worth considering, this is almost always the case for control objects less often for entity objects and almost never for boundary objects. Furthermore, actions in state machine diagrams are candidates for public operations in classes.

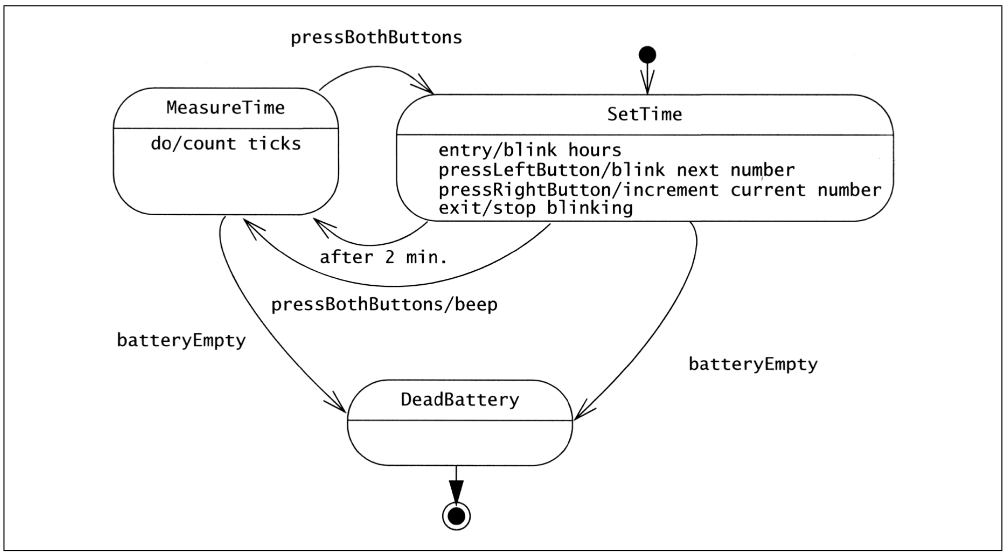


Figure 5: Example for a state machine diagram.

3. System Design

System Design is the transformation of an analysis model into a system design model, considering the solution domain. During requirements elicitation and analysis, we concentrated on the purpose and the functionality of the system.

During system design, we focus on the processes, data structures and software/hardware components necessary to implement it. System design reduces the gap between the problem and the existing machine. We define the design goals of the project and decompose the system into smaller subsystems, to reduce the complexity of the solution domain. These subsystems can be realized by individual teams.

3.1. Design Goals

Design goals are derived from the *non-functional requirements* and describe the qualities of the system that developers should optimize. These criteria are organized into five groups:

- **performance** (response time, throughput, memory),
- **dependability** (robustness, reliability, availability, fault tolerance, security, safety),
- **cost** (development, deployment[1], upgrade, maintenance, administration),
- **maintenance** (extensibility[2], modifiability[3], adaptability, portability, readability[4], traceability of requirements) and
- **end user** (utility, usability).

When defining design goals, only a small subset of these criteria can be simultaneously taken into account. Common trade-offs are:

- space vs. speed
- delivery time vs. functionality
- delivery time vs. quality
- delivery time vs. staffing

3.2. System Decomposition

System decomposition is the identification of subsystems, services and their relationship to each other. The way how we will decompose the system is influenced by the *analysis functional model.*

[1]Cost of installing the system and training the users.
[2]How easy is it to add functionality or new classes to the system?
[3]How easy is it to change the functionality of the system?
[4]How easy is it to understand the system from reading the code?

3.2.1. Managing Subsystems

A subsystem is a replaceable part (logical or physical component) of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes. It is characterized by the services it provides to other subsystems.

A service is a set of related operations that share a common purpose. Operations that are available to other subsystems form the **subsystem interface**.

Later, in object design, we will focus on refining and extending the subsystem interfaces, this will result in the **Application Programmer Interface (API)**. In Unified Modeling Language (UML), provided interfaces of a subsystem can be depicted with ball-icons, required interfaces with socket-icons.

Coupling measures the dependencies between classes of two subsystems. If two subsystems are loosely coupled, the are relatively independent, so modifications to one of the subsystems will have little impact on the other.

Cohesion/coherence measures the dependencies among classes within one subsystem. If a subsystem contains many objects that are related to each other and perform similar tasks, its cohesion is high.

Ideal subsystem decomposition should *minimize coupling* and *maximize cohesion* (see Figure 6). In general, there is a trade-off between cohesion and coupling.

A good heuristic is that developers can deal with 7±2 **concepts** at any one level of abstraction, means if there are for example more than nine subsystems at any given level of abstraction, you should consider revising the decomposition.

A **hierarchical decomposition** of a system yields an ordered set of layers for different levels of abstraction.

A layer is a grouping of subsystems *providing related services*, possibly realized using services from another layer.

Each layer can depend only on lower level layers and has no knowledge of the layers above it.

In a **closed architecture**, each layer can access only the layer *immediately* below it (e.g. OSI model). That means classes within one layer can only access classes in the same layer, or in the layer immediately below them (example see Figure 7). Closed architectures lead to low coupling between subsystems and subsystems can be integrated and test incrementally.

In an **open architecture**, a layer can also access layers at deeper levels.

Another approach to deal with complexity is to **partition** the system into peer subsystems, each responsible for a different class of services. Each subsystem depends loosely on the others, but can often operate in isolation.

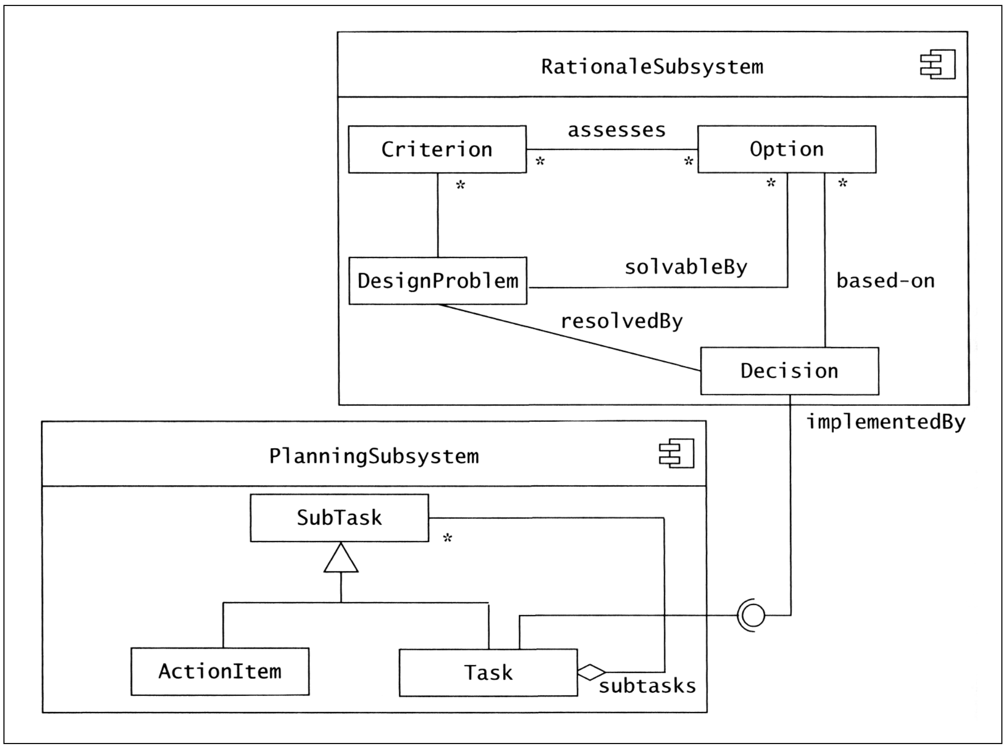


Figure 6: Component diagram (subsystems) with high cohesion and low coupling.

3.2.2. Architectural Styles

As the complexity of systems increases, the specification of system decomposition is critical. The architecture style describes the subsystem decomposition in terms of responsibility, dependencies, hardware mapping and major policy decisions such as control flow, access control and data storage.

Denoting a layer as a *type* (e.g. class, subsystem), a **tier** accordingly is an instance (e.g. object, hardware node). *Layer* and *tier* are often used interchangeably.

The client/server architectural style is a special case of the layered architectural style. One or more *servers* provide services to instances of subsystems, called *clients*.

Each client calls a service offered by the server, the server performs the service and returns the result back to the client. The client knows the interface of the server but the server does not know the interface of the client.

The *end user* interacts only with the client.

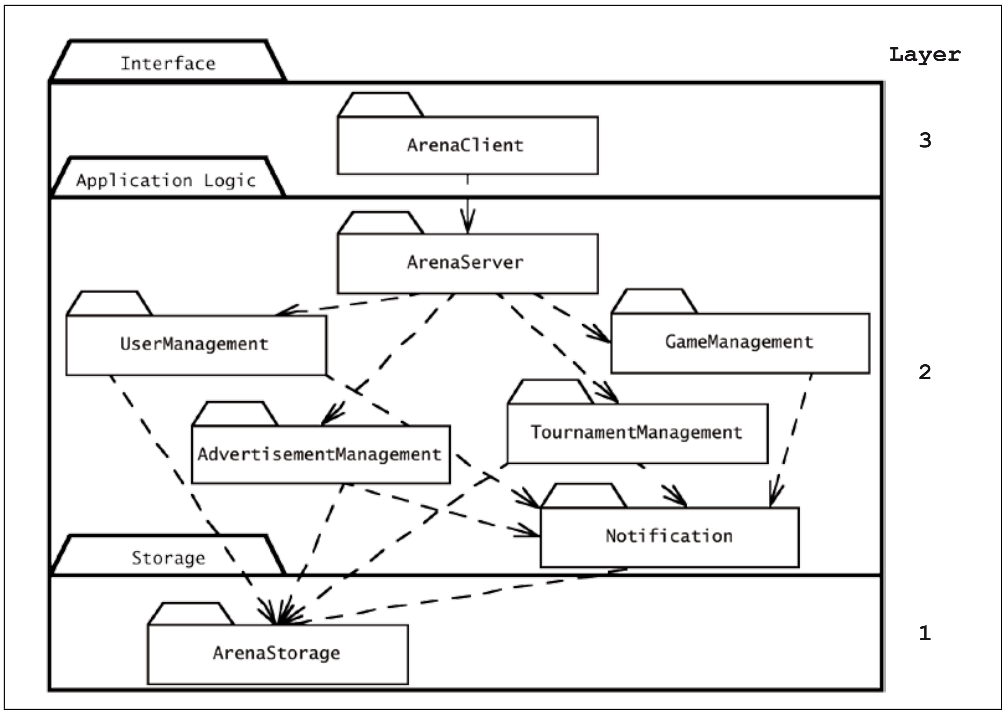


Figure 7: Subsystem decomposition into three layers (depicted as packages).

This architectural style is often used in the design of database systems:

1. **Client:** user application (front-end)
2. **Server:** database access/manipulation (back-end)

Design goals for this architecture are:

1. Portability: many operating systems and networking environments.
2. Location-transparency: server might self be distributed, but provides a single "logical" service to the user.
3. High performance: optimized for CPU-intensive operations.
4. Scalability: server should handle large number of clients
5. Flexibility: the user interface supports a variety of end devices
6. Reliability: server should survive client and communication problems.

The peer-to-peer architectural style is a generalization of the client/server architecture in which subsystems can act as client or as servers, in the sense that each subsystem can request and provide services. It allows a (database) server additionally to communicate with the clients. For example, a database must process queries from `Application1` and should be able to send notifications to `Application2` when data has changed. Hence, a peer is a *requester* (client) as well as a *provider* (server).

The three-tier architectural style allocates the three different layers on three separate hardware nodes:

L3: **Interface**, including all boundary objects that deal with the user (e.g. web browser)

L2: **Application logic**, including all control and entity objects (realizing processing, rule checking etc. required by application - e.g. web server)

L1: **Storage**, realizing the data storage, retrieving and querying persistent objects

The four-tier architectural style is a three-tier style in which the *interface layer* is decomposed into a *presentation client* and *presentation server* layer.

L4: **Client interface** (e.g. web browser)

L3: **Server interface** (e.g. html `form`)

L2: **Application server**, realizing processing and session management

L1: **Storage**

The repository architectural style is characterized, owing to the fact that subsystems access and modify a single data structure called the central repository. Subsystems are relatively independent and interact only through one repository (high coupling), which only ensures that concurrent access is serialized.

The Model/View/Controller (MVC) architectural style (see Figure 8) classifies its subsystems into three different types:

- **Model** subsystems maintain domain knowledge,
- **View** subsystems display it to the user and
- **Controller** subsystems manage the sequence of interactions with the user.

This style decouples data access (entity objects) and data presentation (boundary objects) so that change in one subsystems has less or no impact to the other subsystem.

In action: the *controller* gathers input from the user and sends messages to the *model* which maintains the central data structure. The *views* display the *model* and are notified whenever the *model* is changed.

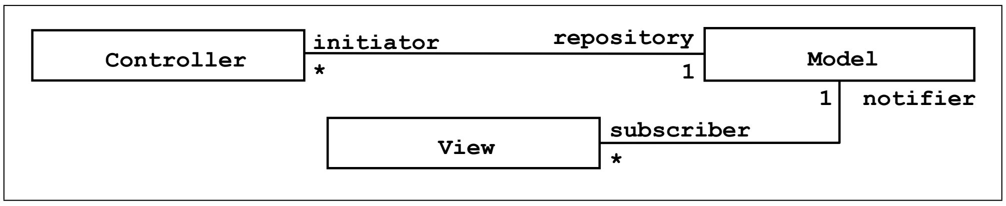


Figure 8: MVC architectural style (depicted by classes).

The MVC architectural style is *non-hierarchical* (triangular), in contrast to the hierarchical three-tier architectural style (linear) in which the presentation layer never communicates directly with the data layer.

The pipe and filter architectural style consists of two subsystems:

- A **filter** does processing steps,
- A **pipe** is the connection between two processing steps.

Each filter has an input and an output pipe. The data from the input pipe is processed by the filter and then moved to the output pipe.

3.3. Activities

3.3.1. Decomposing the System

Identifying Subsystems

The initial subsystem decomposition should be derived from the functional requirements. Subsystems should be related to their functionality.

Heuristics for grouping objects into subsystems are:

- Assign objects identified in one use case into the same subsystem.
- Create a dedicated subsystem for objects that move data among subsystems.
- Minimize the number of associations crossing subsystem boundaries.
- All objects in the same subsystem should be functionally related.

The *Facade design pattern* allows us to further reduce dependencies between classes, by *encapsulating* a subsystem with a simple, unified interface. This allows access only to the public services offered by the subsystem and hides all other details, effectively reduces coupling between subsystems.

Reducing Complexity

To communicate a complex model to each other, developers use *navigation* and *reduction of complexity.*

To navigate the model means starting with the key abstractions (e.g. considering a class `phoneNumber`) and then decorating the model with further details (e.g. consists of `areaCode`, `mainNumber` and `extension`). To reduce complexity, we place subsystems in separate *packages.*

Considering Figure 9, we can rely on four techniques for identifying packages:

1. **Looking for key abstractions**
   `Project` package consists of five classes:
   `Project`, `Outcome`, `Schedule`, `Work` and `Resource`.
2. **Identifying patterns**
   `Outcome`, `Work` and `OrganizationalUnit` are *Composite patterns*, each can be depicted as own package.
3. **Identifying (application domain) taxonomies**
   `WorkProduct`, `Activity`, `Staff` and `Resource` all have subclasses, so each of them can be depicted as own package.

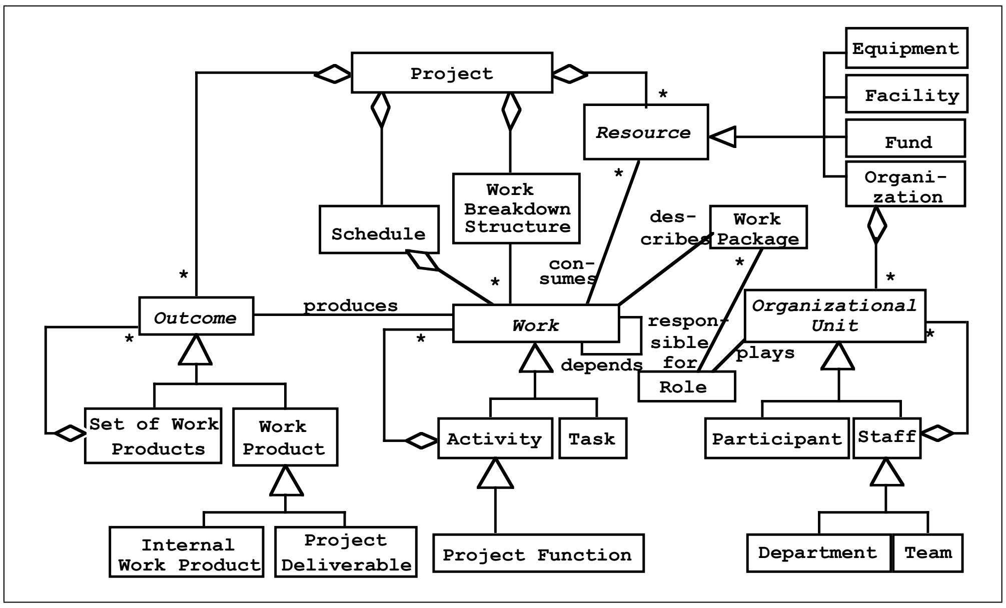


Figure 9: Example: Modeling a project (class diagram).

3.3.2. Addressing Design Goals

Concurrency

To address non-functional requirements like *performance*, *response time*, *latency* or *availability*, we use concurrency in system design. Two objects are **inherently concurrent**, if they can receive events at the same time without interacting (e.g. objects in sequence diagrams that are independently interacting with other objects). To figure out these objects, we consider our *analysis dynamic models.*

A **thread of control** is a path through a set of state diagrams where only a *single object is active* at any time. A thread remains within a state diagram until the object sends an event to a second object, which then becomes active, the first objects then waits for another event. The results of *thread splitting* are two (or more) active objects and hence, two (or more) concurrent threads of control.

But concurrent threads can lead to *race conditions* and to non-deterministic results (example: two customers accessing one bank account concurrently). To prevent this problem, we synchronize threads, that means we protect objects from being accessed by more than one threads simultaneously.

Within a sequence diagram, we can depict different types of messages by using three types of arrows (see Figure 10).

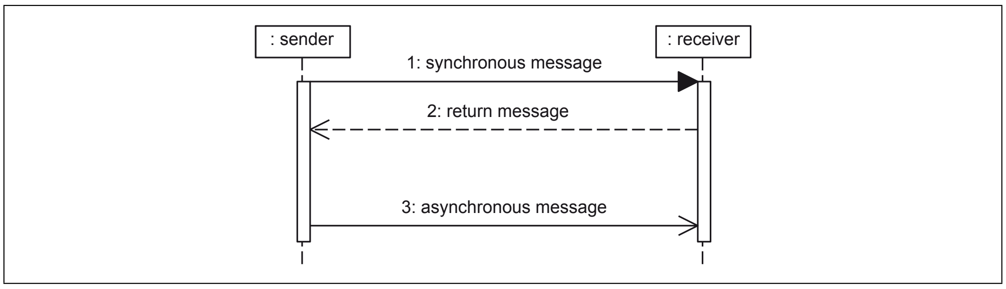


Figure 10: Three types of messages within a sequence diagram.

- A **synchronous message** means, that the sender waits until the receiver has finished processing the message and continues after receiving a message back.
- An **asynchronous message** means, that the sender continues immediately and will not wait for a return message.

Questions for identifying concurrency are:

- Does the system provide access to multiple users?
- Which entity objects of the object model can be executed independently from each other?
- What kinds of control objects are identifiable?
- Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel?

There are two ways for implementing concurrency: *physical* and *logical.*

Hardware/Software Mapping

During this activity, we will take a look at our *analysis object model* and will answer these two questions:

- Should a subsystem be realized with hardware or software?
- How do we map the object model onto the chosen hardware and software?

Mapping the object model onto the hardware means:

- **Objects** correspond to processors, memory and I/O devices.
- **Associations** correspond to network connections.

Modeling in UML, we can use **deployment diagrams** (see Figure 11) to illustrate the distribution of components at run-time. A **node** (represented by boxes, e.g. `Server`) is a physical device or an environment in which components (e.g. `LoadBalancer`) are executed. Furthermore, a node can contain another node, for example, a device can contain an execution environment.

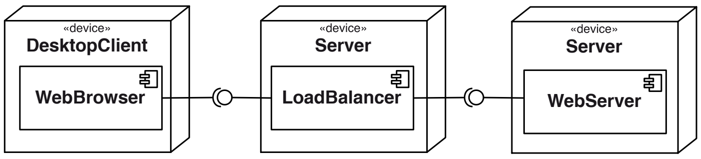


Figure 11: Example of a deployment diagram.

Mapping objects onto hardware:

- **Control objects** correspond to processors.
- **Entity objects** correspond to memory.
- **Boundary objects** correspond to I/O devices.

Mapping associations onto connectivity:

- **Physical connectivity** describes which associations in the object model are mapped to physical connections.
- **Logical connectivity** describes the associations between subsystems, that do not directly map into physical connections.

Informal connectivity drawings often contain both types of connectivity, that sometimes leads to confusion.

(Persistent) Data Management

Some objects in a system model need to be **persistent**, that means values of attributes must have a lifetime beyond a single execution.

Candidates for **persistent objects** are *entity objects* within our *analysis object model*, as well as some (attributes of) *boundary objects* that store e.g. user interface preferences.

We may use a *file system*, if we assume that the data will be used by multiple readers but a single writer. A *database system* ensures access to concurrent readers and writers.

Mapping objects to a relational database means that *classes* will be mapped to *tables*, class *attributes* will correspond to *columns* and *objects/instances* will be represented by *rows*.

Object-oriented databases provide services similar to relational databases, but store data as objects and associations.

Global Resource Handling (Access Control)

This activity describes access rights for different classes of actors and how object can be protected against unauthorized access.

In multi-user systems, different actors have access to different functionality and data. During analysis, we modeled these distinctions by associating different use cases to different actors (*analysis dynamic model*). During system design, we model access by determining which objects are shared among actors and by defining how actors can control access.

Access on classes is modeled with an **access matrix**. *Rows* represent *actors*, *columns* represent *classes* whose access we control. An entry in the access matrix is called an *access right* and lists the operations (e.g. `view()`, `edit()` or «`create`») that can be executed on instances of the classes by a certain actor.

We can represent the access matrix using one of three different approaches:

- A **global access table** represents explicitly each cell in the matrix as a *(actor,class,operation)* tuple, hence allows or denies access for an actor.
- An **access control list** associates a list of *(actor,operation)* pairs with each class to be accessed. Every time an object is accessed, its access list is checked for the corresponding actor and operation.
- A **capability** associates a *(class,operation)* pair with an actor. It allows an actor to access an object of the class, described in the capability.

Questions concerning about **authentication** within a system are:

- What is the authentication scheme?
  - user name and password: *access control list*
  - tickets: *capability-based*
- What is the user interface for authentication?
- Does the system need a network-wide name server?
- How is the authentication known to the rest of the system?
  - At runtime, compile time, by port or by name?

Access control can be realized by using the *Proxy design pattern*. Implemented in *Java* for example, the proxy object may protect the real object by checking `if(user instanceof AllowedUserGroup)`, at the begin of relevant methods.

Software Control

Control flow is the sequence of actions in a system. Sequencing actions includes deciding which operations should be executed and in which order. Considering our *analysis dynamic model* will help us during this activity.

There a two main system design choices concerning software control: implicit or explicit.

- **Implicit control**
  - **Rule-based control**
  - **Logic programming**
- **Explicit control**
  - **Centralized:** One control object/subsystem (*spider*) controls everything.
    + *Pro:* Change in the control structure is very easy.
    – *Con:* The single control object is a possible performance bottleneck.
    - **Procedure-driven:** Control resides within the program code. Hence, operations wait for input whenever they need data from an actor.
      - **Threads** are the concurrent variation procedure-driven control.
    - **Event-driven:** Control resides within a dispatcher, calling other methods via *callbacks*. Means, a main loop waits for an external event (e.g. user input), that is then dispatched to the appropriate object.
  - **Decentralized:** Control is distributed to several control objects.
    + *Pro:* possible speed up by mapping the objects on different processors.
    – *Con:* increased requires communication, responsibility is spread out.

A very easy way to figure out, if rather centralized or decentralized control should be used is, to take a look at the sequence diagrams and control objects from the analysis model. A *fork diagram* will lead to centralized control, a *stair diagram* to decentralized control.

Boundary Conditions

At this point, we still need to examine the boundary conditions of the system. Boundary conditions strongly depend on (hardware) design decisions and not on requirement decisions. Furthermore, many system functions can be inferred from everyday user requirements. So, it is common that they are not specified during analysis and treated separately.

Boundary conditions are categorized into three groups:

Initialization concerns about how to bring the system from a *non-initialized* state to a *ready* state, including:

- Data that needs to be accessed at startup time,
- Services that have to be registered and
- Configuring the user interface.

Termination deals with how resources are cleaned up and how other systems are notified upon termination, including questions like:

- Are single subsystems allowed to terminate?
- Are subsystems notified if a single subsystem determines?
- How are updates communicated to the database?

Failure leads to **exception handling** that specifies how the system should react in case of, for example bugs, errors or external problems. Exception handling deals with questions like:

- How does the system behave when a node or communication links fails?
- How does the system recover from failure?

Good system design foresees fatal failures and provides mechanisms to deal with them.

Boundary conditions are best modeled as **boundary use cases** or **administrative use cases**, hence they extend our *functional model.* Interesting use cases are the start up of several subsystems, as well as of the full system, its terminations and errors and failure within subsystems or components.

A concrete example for a boundary use case could be an `Administrator` who invokes the use case `ManageServer`. `ManageServer` then will `«include»` other use cases, like `StartServer`, `ShutdownServer` and `ConfigureServer`.

4. Object Design

During *analysis*, we identify application objects (entity) that are independent from a specific system, and solution objects (boundary/control), that are visible to the user (e.g. forms). More solution objects are identified during *system design*, in terms of hardware/software mapping.

During *object design*, we refine and detail both, application and solution objects and identify additional solution objects needed to bridge the gap between a given problem and an existing machine (see Figure 12).

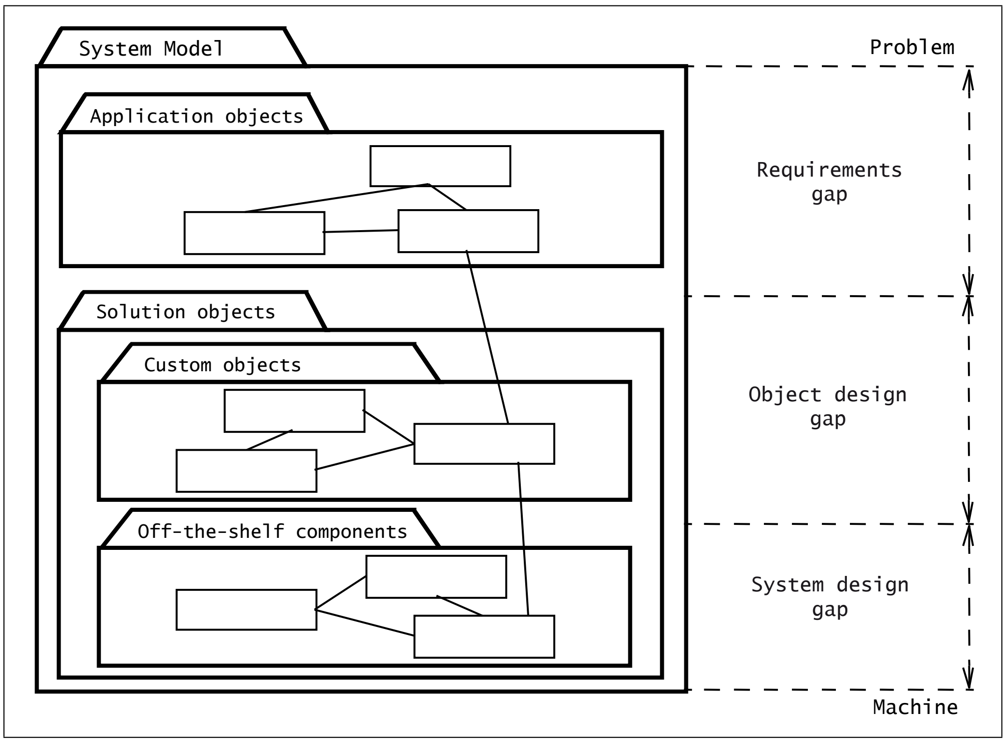


Figure 12: Object design closes the gap between application objects and off-the-shelf components.

4.1. Code Reuse

During analysis, we use inheritance to classify objects into taxonomies. This allows us to differentiate the common behavior of the general case, the **superclass**, from the behavior that is specific to specialized objects, the **subclass**. The focus of inheritance during object design is to reduce redundancy (and the risk of inconsistencies) and enhance extensibility. Figure 13 shows the inheritance meta-model.

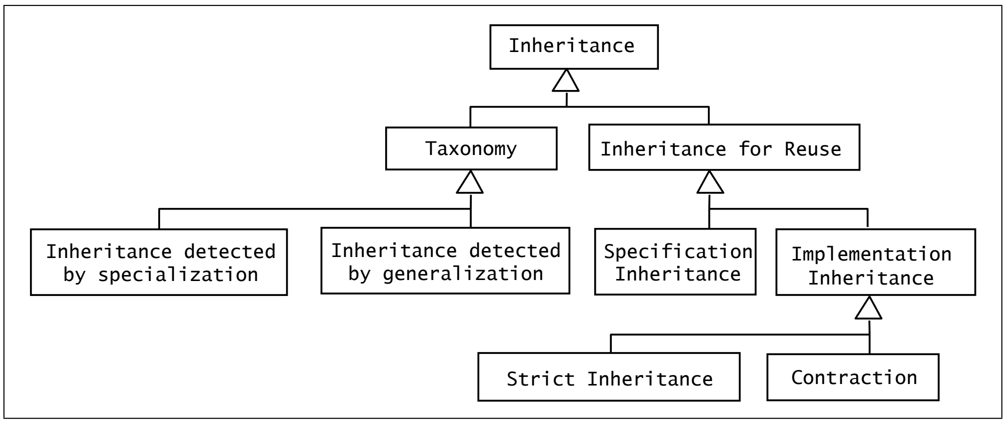

Inheritance
Taxonomy
Inheritance for Reuse
Inheritance detected by specialization
Inheritance detected by generalization
Specification Inheritance
Implementation Inheritance
Strict Inheritance
Contraction

Figure 13: Inheritance in analysis and design.

In terms of reusing code, we distinguish between two types of inheritance:

Implementation inheritance is a mechanisms that describes the sole reuse of code by subclassing an existing class and refining its behavior.

Specification inheritance (also called *interface inheritance*) is the classification of concepts into hierarchies. Often superclasses (and their predefined operations) are abstract.
Strict inheritance allows to only add new methods to the *subclass*, but not to override existing methods of the *superclass*.
Seen from another perspective, this allows us to invoke methods, written in terms of a superclass `T`, on any instances of subclasses of `T`, which is the essence of the **Liskov Substitution Principle**.

Implementation inheritance often leads to a problem: extending a class, merely to reuse some behavior (e.g. certain methods), unavoidable pretends some kind of specification. That means, other developers may consider, that methods of the *subclass* provide at least the same functionality as those in the *superclass*. But in fact, overridden methods may have changed their behavior or even do nothing if their body has become empty (called *contraction*). An alternative of implementation inheritance is *delegation*.

Delegation is a mechanism for code reuse in which an operation resends a message to another class to accomplish the desired behavior.
(`Client` calls `Receiver` which delegates the `Delegate` / `LegacyClass`).

This addresses the problems mentioned before: extensibility and subtyping. Delegation is also called *composition* or *blackbox reuse*, whereas the use of inheritance is also called *whitebox reuse*.

4.2. Design Patterns

Design patterns are template solutions that developers have refined over time to solve a range of recurring problems.

A design pattern has four elements:

1. A unique **name**,
2. a **problem description**, describing when the pattern can be used,
3. a **solution** stated as a set of collaborating classes and interfaces, and
4. a set of **consequences**, describing trade-offs and alternatives to be considered.

These four elements are not explicitly mentioned, while describing a couple of design patterns on the following pages.

Classes in design patterns often use inheritance and delegation, they are usually denoted as:

- **client class** (accesses the pattern),
- **pattern interface** (visible to the client class, often abstract class or interface),
- **implementor class** (provides lower-level behavior), and
- **extender class** (specializes an implementor class to provide a different implementation or extended behavior of the pattern).

We distinguish between three types of design patterns:

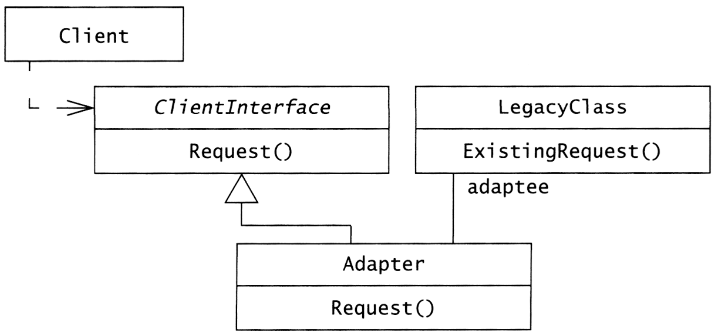
- **Structural Patterns** (e.g. Adapter, Bridge, Composite, Facade)
  - reduce coupling between classes,
  - introduce an abstract class to enable future extensions,
  - encapsulate complex structures.
- **Behavioral Patterns** (e.g. Observer, Strategy, Template Method)
  - allow a choice between algorithms,
  - allow the assignment of responsibilities to objects,
  - model complex control flows that are difficult to follow at runtime.
- **Creational Patterns** (e.g. Abstract Factory)
  - allow to abstract from complex instantiation processes,
  - make the system independent from the way its objects are created, composed and represented.

Design patterns help us to design our system more modifiable and extensible in case of change like new vendor/technology, implementation, views, complexity of the application domain or errors.

4.2.1. Adapter (S): Wrapping around Legacy Code

The Adapter pattern converts the interface of a `LeagcyClass` into a different interface expected by the `Client`, so that both can work together without changes.

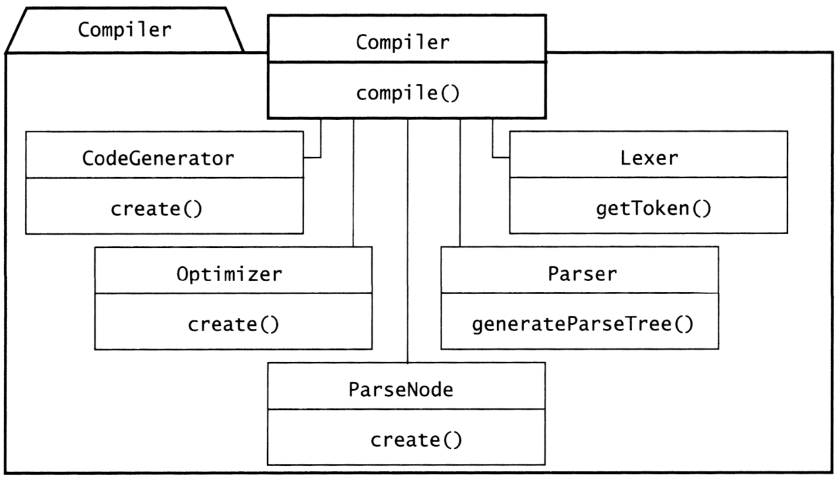
The `Adapter` class implements the `ClientInterface`, expected by the `Client`. The `Adapter` then delegates requests from the `Client` to the `LeagcyClass` and performs any necessary conversion.



4.2.2. Facade (S): Encapsulating Subsystems

The Facade pattern reduces coupling between a set of related classes (subsystem) and the rest of the system, while providing a high-level interface.

The `Facade` consists of a set of public operations, each of them delegated to one or more operations of classes behind (within) the `Facade`.



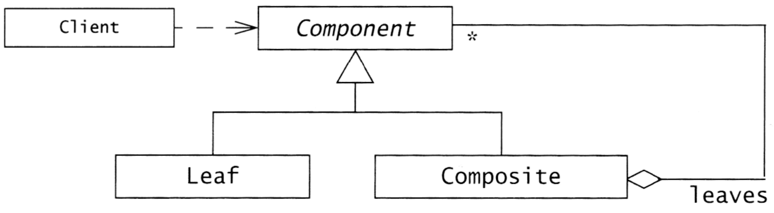
It is recommended to offer a Facade for each subsystems that provides a set of services.

4.2.3. Composite (S): Representing Recursive Hierarchies

The Composite pattern represents a hierarchy of variable width and depth.

`Leaves` and `Composites` (containers) can be treated uniformly through a common interface. That means, the `Client` can treat (invoke methods on) individual objects and compositions of these objects in the same way and without knowing which specific class each object belongs to.

The `Component` interface specifies the services (operations) that are shared among `Leaf` and `Composite`. `Leaf` and the `Composite` contain the concrete implementations. Usually, there is more than just one `Leaf`, and `Leaves` might also have subclasses.



Example Drawable elements (e.g. lines, circles) can be organized in groups that can be moved and scaled uniformly. Note that groups can also contain other groups (variable depth). The `Composite` may contain operations like `addComponent()` or `moveComponent()`. When a new element, a `Leaf` (e.g. rectangle) will be added (variable width), we only have to implement the certain operations (e.g. `draw()`, `move()`) and do not have to change anything within the `Component` interface.

Another example is a hierarchical file system in which a `FileSystemElement` can either be a `Directory` or a `File`. A `Directory` (`Composite`) can again contain a `FileSystemElement`.

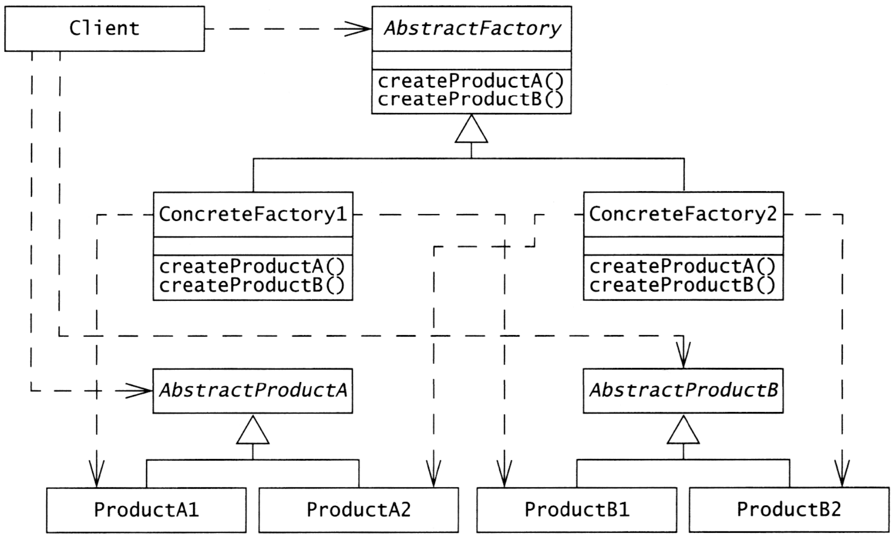
In the same way, a `HtmlTag` may for instance be an `img`-tag, or it is a `div`-tag that again can contain `HtmlTags`.

4.2.4. Abstract Factory (C): Encapsulating Platforms

With help of the Abstract Factory pattern, we can get rid of `if-then-else` statements, usually needed to instantiate a `Product` that is produced by different manufacturers, while using dynamic binding.

The `AbstractFactory` defines the interface that both `ConcreteFactories` have to implement to create `Products`.

So once, the `Client` has picked a `ConcreteFactory` that should create all `Products`, a call `createProductA()` or `createProductB()` on the `AbstractFactory` is enough to have a consistent set of different products from the same manufacturer.

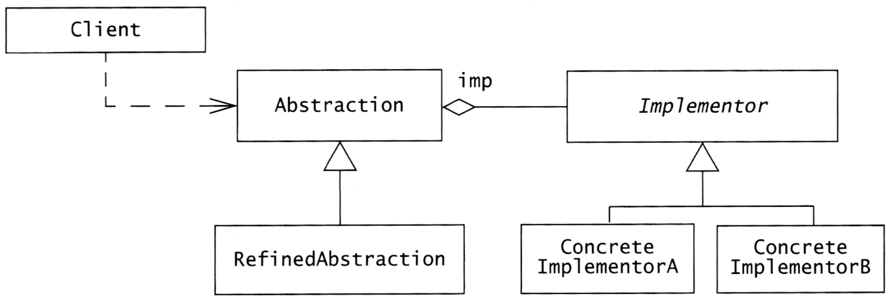


An `AbstractProduct` is for example a `LightBulb` or a `Blind`. A concrete `Product` then could be `PhilipsBulb` or `OsramBulb` both having different implementaions of `switchOn()` and `switchOff()`.

Initiation associations are depicted as dashed lines, that means that for example `ConcreteFactory2` initiates the associated `ProductB2` and `ProductA2` at runtime.

4.2.5. Bridge (S): Allowing for Alternative Implementations

The Bridge pattern allows our system to be more flexible. This pattern can be used to provide multiple implementations under the same interface and allows to delay the binding between the interface and a concrete implementation to the start-up time of the system.



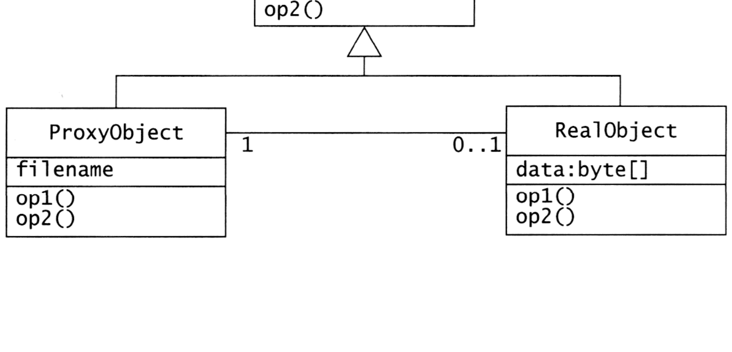
An example for different implementations can be the storage of persistent data. A `StoreImplementor` defines the interface to the `Client`, while `ImplXml` and `ImplDatabase` provide the concrete implementation for storing data.

Adapter and Bridge pattern are similar in purpose and structure. Both decouple an interface from an implementation. But, seen from the `Client`, the Bridge pattern uses delegation first and then inheritance, which allows us to add new `ConcreteImplementations`. The Adapter pattern first uses inheritance and then delegation to the `LegacyClass`.

4.2.6. Proxy (S): Encapsulation Expensive Objects

The Proxy pattern improves the performance or the security of a system by delaying expensive computations, using memory only when needed, or checking access before loading an object into memory (Remote, Virtual, Protection P.).

The Client always invokes operations on instances of the `ProxyObject`. This operations use delegation to access the corresponding operations in the `RealObject` (if needed or allowed).



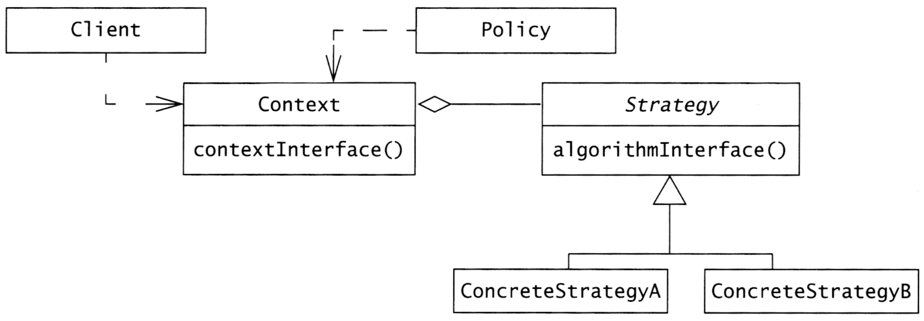
4.2.7. Strategy (S): Encapsulating Algorithms using Delegation

The Strategy pattern allows a system to switch between different algorithms for a specific task at *runtime*, depending on a certain context.

The abstract `Strategy` class describes the interface (operations visible to the `Client`) that is common to all `ConcreteStrategies`. Each `ConcreteStrategy` contains a concrete algorithm.

Since the choice between those strategies depends on "external circumstances", a new class `Context` is added, that fulfills two functions. For one thing, it selects the `ConcreteStrategy` to be used and for another thing, it provides the `Strategy` interface to the `Client`.

The decision which, `ConcreteStrategy` the `Context` has to use (binding of a `ConcreteStrategy`) is made by the `Policy` object. That means, when the `Client` invokes a certain operation, the `Policy` decides which `ConcreteStrategy` is the best, given the current circumstances.



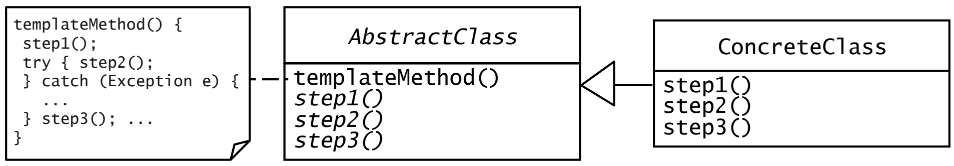
Example A `NetworkConnection` (`Context`) can be realized using different strategies. A `NetworkInterface` (`Strategy`) for instance provides `Ethernet`, `WaveLan` and `Umts` (`ConcreteStrategies`).

The `LocationManager` (`Policy`) then picks one of these strategies, depending on which is available and the fastest for the moment. The `Application` (`Client`) then just invokes `open()`, `send()` or `close()` and does not care about the technology used to establish the connection.

Bridge and Strategy pattern look almost identical. The key difference between both is, that the `ConcreteImplementations` in the Bridge pattern are usually created at initialization time, while `ConcreteStrategies` in the Strategy pattern are usually created and substituted several times during run time.

4.2.8. Template Method (B): Encapsulating Algorithms using Inheritance

The Template Method pattern uses an `AbstractClass` to define a template for related algorithms (`ConcreteClass`/es) as a set of ordered and small steps in terms of abstract operations.

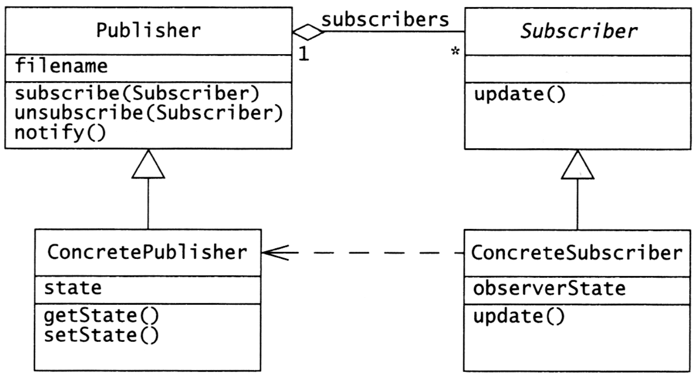


The `templateMethod()` ensures that all operations are invoked in the right order. All abstract operations are implemented by the `ConcreteClass`(es).

4.2.9. Observer (B): Decoupling Entities from Views

The Observer pattern maintains consistency across (redundant) views, whenever the state of the observed object changes.

The `Publisher` (also called `Subject`) represents the *entity object*, maintaining some state. Since it is the `Publisher`, that notifies all `Subscribers`, whenever the state has changed, it can also be seen as a control object in the model. Hence, we add a new class `ConcretePublisher`, that now undertakes the role of being the entity object and only maintains the state.



The `Subscribers` is the superclass of all different views, called the `Concrete-Subscribers`.

Whenever the state of `Publisher` changes, it invokes the `notify()` method, which iteratively invokes each `Subscriber.update()` method.

The Observer pattern can be used for realizing subscriptions and notification in a MVC architecture.

4.3. Frameworks

An application framework is a reusable partial application that can be specialized to produce custom applications. In contrast to class libraries, frameworks are targeted to particular technologies (e.g. processing) or applications domains (e.g. user interface). The key benefits of frameworks are reusability and extensibility.

Frameworks can be classified by their position in the software development process:

- **Infrastructure frameworks** aim to simplify the software development process (used internally).
- **Middleware frameworks** are used to integrate existing distributed applications and components (e.g. CORBA).
- **Enterprise frameworks** are application specific and focus on domains.

They can also be classified by the techniques used to extend them:

- **Whitebox frameworks** rely on inheritance and dynamic binding.
- **Blackbox frameworks** support extensibility by defining interfaces (integrated via delegation).

Design patterns vs. frameworks. Frameworks focus on reuse of concrete designs, algorithms and implementations whereas design patterns reuse abstract designs and small collections of cooperating classes.

Class libraries vs. frameworks. Class libraries are less domain specific, provide a smaller scope of reuse and are passive (they do not constraint the control flow). Frameworks are active (they control the flow of control) and they often use class libraries themself. Their classes cooperate for a family of related applications.

Components vs. frameworks. Components are self-contained instances of classes that are plugged together to form complete applications. In terms of reuse, a component is a blackbox that defines a cohesive set of operations and can even be reused on the binary code level (advantage: the entire application does not has to be recompiled when certain components change). Frameworks are often used to develop components and components can be plugged into blackbox frameworks.

4.4. Interface Specification

During interface or service specification, we precisely describe each class interface, including operations, arguments, type signatures and exceptions. The result of service specification is a complete interface specification for each subsystem (often called API). That means, we describe the conditions under which an operation can be invoked and those under which it raises an exception.

4.4.1. Points of View

During object design, developers play different roles. We differentiate them based on their point of view:

- **Class implementors** design the internal data structures and implement the code for each public operation, and therefore the interface specification.
- **Class users** invoke public operations during the realization of another class (client class).
- **Class extenders** focus on specialized versions of the same services by using public and protected operations.

During implementation a developer may be class implementor as well as user and extender. Which classes this developer implements or uses, depends on the part of the system (e.g. a certain package) (s)he is working on.

4.4.2. Class Details

Since the developer roles (described above) are assigned during object design we must refine our object model and add details to attributes and operations. The **type** of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute. The **signature** of an operations consists of the types of parameters and the type of the return value (e.g. `acceptPlayer(Player):void`).

Because a *class user* for example, should not be able to see the internal data structures of the class, we need to add information about the **visibility** to attributes and operations (see Table 5).

	Visibility	**Access by other classes**	**Intended for**
-	private	only within the same class	implementor
#	protected	also subclasses and classes of the package	extender
+	public	by any class	user
∼	package	by any class of the package	

Table 5: The four levels of visibility, defined in UML.

4.4.3. Contracts with OCL

Contracts specify constraints that the class user must meet before using the class as well as constraints that are ensured by the class implementor/extender when used.

Contracts include three different types of constraints:

- An **invariant** of a class is a predicate that is *always* true for all of its instances.
- A **precondition** is a predicate that must be true *before* an operation is invoked.
- A **postcondition** is a predicate that must be true *after* an operation is invoked.

Preconditions correspond to *rights* and postconditions to *obligations*. Constraints on model elements (e.g. attributes, operations, classes) or groups of elements (e.g. associations and participating classes) can be expressed in **Object Constraint Language (OCL)**.

A **constraint** is expressed as boolean expression, returning the value `true` or `false` and can be depicted as a note attached to an UML element by a dependency relationship (see Figure 14).

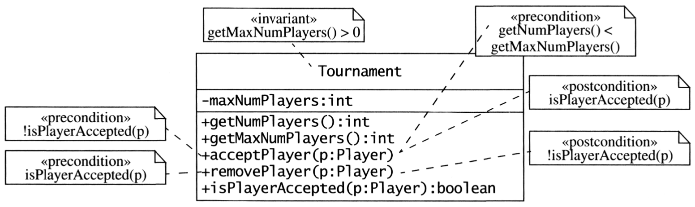

«invariant» getMaxNumPlayers() > 0
«precondition» getNumPlayers() < getMaxNumPlayers()
Tournament
-maxNumPlayers:int
+getNumPlayers():int
+getMaxNumPlayers():int
+acceptPlayer(p:Player)
+removePlayer(p:Player)
+isPlayerAccepted(p:Player):boolean
«postcondition» isPlayerAccepted(p)
«postcondition» !isPlayerAccepted(p)
«precondition» !isPlayerAccepted(p)
«precondition» isPlayerAccepted(p)

Figure 14: Constraints in OCL attached as notes to a class diagram.

Another way to express constraints in OCL is in textual form. Considering the class `Tournament`, depicted in Figure 14, the **invariant** requiring the attribute `maxNumPlayers` to be positive is written as follows:

```
context Tournament inv:
  self.getMaxPlayers() > 0
```

The `context` keyword (line 1) indicates the entity to which the *expression* (line 2) applies. The keywords `inv`, `pre` and `post` correspond to «`invariant`», «`precondition`» and «`postcondition`». The keyword `self` (for attributes and operations) denotes all instances of the class, but can be omitted if there is no ambiguity (if there is not operation with the same name in another class). The whole construct is called *predicate*.

For pre- and postconditions, the context of the expression is an operation. Considering the following **precondition**:

```
context Tournament::acceptPlayer(p:Player) pre:
  !isPlayerAccepted(p) and
  getNumPlayers() < getMaxNumPlayers()
```

The variable `p` within the constraint (line 1) refers to the parameter `p` passed to operation (line 2). We can write several preconditions for the same operation, that all must be `true` before the operation can be invoked.

Using **postconditions**, we often need to refer to the value of an attribute or returned by an operation, before and after the execution of the operation.

```
context Tournament::acceptPlayer(p:Player) post:
  getNumPlayers() = self@pre.getNumPlayers() + 1
```

The suffix `@pre` denotes the value returned by `getNumPlayers()` before invoking `acceptPlayer()`.

Collections

See section 9.3.5 in the book.

Quantifiers

See section 9.3.6 in the book.

4.5. Activities

Object design includes four groups of activities (see also Figure 15). Usually, **reuse** activities and **interface specification** occur first. Once, the object design model for the subsystem is relatively stable, **restructuring** and **optimization** activities occur next.

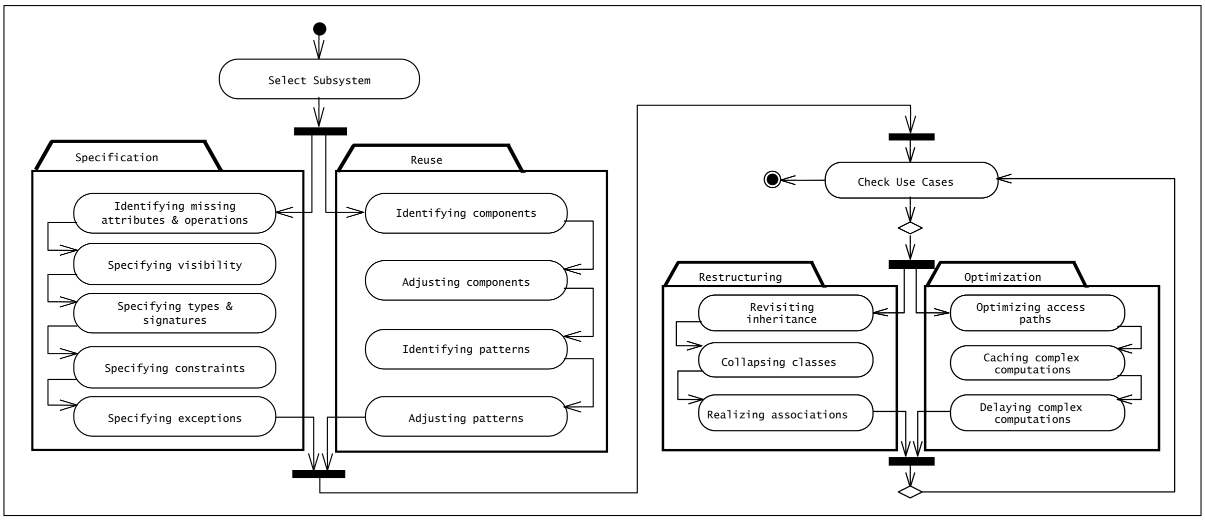


Figure 15: Activities during object design (activity diagram).

4.5.1. Reusing Pattern Solutions

During this activity, we identify off-the-shelf components and design patterns, to make use of existing solution.

Similar to Abbott's heuristics, key phrases in the Requirements Analysis Document (RAD) or System Design Document (SDD) can be used to identify candidate design patterns (see Table 6).

Phrase	**Design Pattern**
Manufacturer/platform independence	Abstract Factory
Must comply with existing interface/reuse legacy component.	Adapter
Must support future protocols.	Bridge
All commands should be undoable, all transactions logged.	Command
Must support aggregate structures, or hierarchies of variable depth/width.	Composite
Policy and mechanisms should be decoupled. Must allow different algorithms to be interchangeable at runtime.	Strategy

Table 6: Natural language heuristics for identifying design patterns.

4.5.2. Specifying Interfaces

Identifying Missing Attributes and Operations

During this activity, we examine the service description of the subsystem and identify missing attributes and operations. These details are related to the system and independent of the application domain, hence, they have been ignored when constructing the object model.

Example: The number of concurrent tournaments and matches within the `ARENA` project is the main consumer of server resources. Now, we focus on possible misuse of the system in which player attempt to play several matches concurrently. In this case, we need an additional operation that checks, if the tournament a player wants to apply for overlaps with one, the player has already been accepted for.

Specifying Types, Signatures and Visibility

Specifying types refines the object model in two ways. First, we specify the range of each attribute (e.g. `+start:Date` may represent the time as `YYYY-MM-DD`). Second, we map classes and attributes to built-in types (e.g. `String`, `Map` or `List`) and can use all the operations provided to manipulate the values.

Within a class diagram, examples for attributes and operations could be:

- `-maxNumPlayers:int`
- `+name:String`
- `+start:Date`
- `+getMaxNumPlayers():int`
- `+isPlayerAccepted(p:Player):boolean`

Good information hiding heuristics are:

- Carefully define the public interface for classes and subsystems
  (use Facade design pattern for subsystems).
- Apply the "need to know" principle (set *public* only if access is needed).
- The fewer details a class user has to know:
  - the easier the class can be changed and
  - the less likely the class user will be affected by any change.
- Make attributes always *private*.

Specifying Constraints

See sections 9.4.3-5 in the book.

4.5.3. Optimization and Restructuring

The optimization of the object model addresses performance criteria (e.g. response time or memory utilization). Typical activities include:

- changing of algorithms to respond to speed or memory requirements,
- reducing multiplicities in associations to speed up queries,
- adding redundant associations for efficiency,
- rearranging execution orders
- adding derived attributes to improve the access time to objects, and
- opening up the architecture
  (adding access to lower layers because of performance requirements)

The restructuring of the object model addresses the increase of code reuse and design goals such as maintainability, readability and understandability of the system model. Typical activities include:

- transforming N-ary into binary associations,
- implementing binary associations as references,
- merging two similar classes from different subsystems into a single class
- collapsing classes with no significant behavior into attributes,
- splitting complex classes into simpler ones, and/or
- rearranging classes and operations to increase inheritance and packaging.

Activities that refer to the optimization and restructuring of the object model are explained in the next section in more detail.

5. Mapping Models to Code

Working on an object model involves many transformations that are error prone. For example, developers perform local transformations to the object model to improve its modularity and performance, they transform the associations of the object model into collections of object references, etc.

In each transformation, small errors usually creep in, resulting bugs and test failures. We now focus on a set of techniques to reduce the number of such errors.

Since we have two perspectives on our system, the *object model* and the *source code* (see Figure 16), change in one has impact on the other. There are two ways to maintain consistency between both:

Forward Engineering improves the consistency of the source code with respect to the object model.

Reverse Engineering discovers the object model from the source code (in case the model became out of sync with the code or even lost).

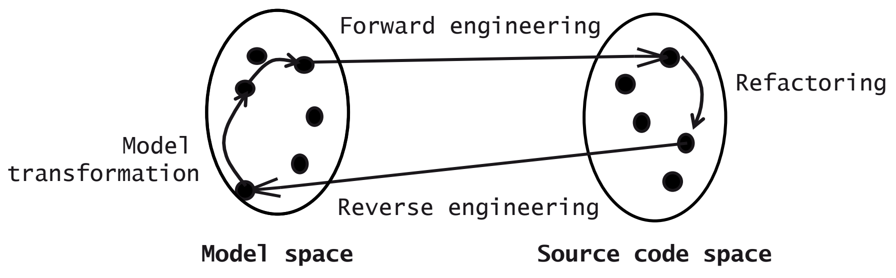


Figure 16: The four types of transformation.

5.1. Model Transformation

The purpose of object model transformation is to simplify or optimize the original model, bringing it into closer compliance with a design goal. A transformation may add, remove or rename classes, operations, associations or attributes, or even add or remove information.

Considering a model that consists of three classes: `LeagueOwner`, `Advertiser` and `Player`, all having an attribute `email` in common. Although, there is no need for generalization in order to make the model more understandable, we may reduce redundancy by moving the `email` attribute to a new superclass that we call `User`.

In principle, the whole development process can be thought as a series of model transformations, starting with the analysis model and ending with the object model.

5.2. Refactoring

A refactoring is a transformation of the source code that improves its readability or modifiability without changing the behavior of the system. To ensure that refactoring does not change the behavior of the system, the refactoring is done in small incremental steps, that are interleaved with tests.

For example, the object model transformation above corresponds to a sequence of three refactorings:

1. **Pull up field:** moves `email` field from the subclass to the superclass `User`. *Important:* ensure that the `email` field in all three classes is equivalent.
2. **Pull up constructor body:** moves the initialization code to the superclass. *Important:* add the call `super(email)` to each constructor of the subclasses.
3. **Pull up method:** moves the methods manipulating `email` to the superclass. *Important:* Examine methods of the subclasses that use the `email` field. Only methods that do not use any fields or operations, specific to that subclass can easily be moved to the superclass.

5.3. Transformation Principles

To avoid introducing new errors, each transformation:

1. must address a single criteria,
2. must be local (only a few methods of classes at once),
3. must be applied in isolation to other changes, and
4. must be followed by a validation step.

5.4. Java Keywords

- `SubClass extends Class`: Inheritance
- `final method()`: Strict inheritance
- `abstract Class`, `interface Interfaceable`: Specification inheritance

5.5. Activities

5.5.1. Optimizing the Object Model

Repeated associations traversals

Operations that are invoked frequently should not require many traversals, but should have a direct connection between the querying object and the queried object. If that direct connection is missing, an association between these two objects should be added. Inefficient access paths can be identified with the help of sequence diagrams.

Collapsing Classes

During analysis, many classes are identified that turn out to have no interesting behavior. If most attributes in such a class are only involved in `set()` and `get()` operations, we should consider folding these attributes into the calling class and removing the called class from our model.

Delaying Expensive Computations

Often, specific objects are expensive to create. If their creation can be delayed until their actual content is needed, we ca can use the *Proxy design pattern.*

5.5.2. Mapping Associations to Collections

Associations are UML concepts that denote collections of *bidirectional* links between two or more objects. Object-oriented programming languages, however, do not provide the concept of association. Instead, they provide *references*, in which one object stores a handle to another object and *collections*, in which references to several objects can be stored and possible ordered. References are *unidirectional* and take place between two objects.

One-to-one associations correspond to a simple reference from one class to another. If only one class has a reference to the other, this association is *unidirectional.* If both classes need to know each other (both invoke methods of each other), the association has to be *bidirectional*, hence both have references.

One-to-many associations are realized using a collection of references. We may use for example `java.util.List` or `Set`, depending on the constraints on the association (e.g. allow duplicates, order items).

Many-to-many associations are realized using collections of references in both classes and methods to keep these collections consistent.

Considering two classes `Tournament` and `Player`. Each `Tournament` object has a collection (e.g. `List`) that refers to all involved `Player` objects. In the same way, each `Player` object has references to the `Tournaments` he will play at. In case, an administrator drops one `Player` from a certain `Tournament` by updating the `List` of `Players`, the `Player` needs to be informed and his `List` of `Tournaments` needs to be updated as well (see Java code below).

```
class Tournament {
  List players = new List();
  void addPlayer(Player p) {
    if(!players.contains(p)) {
      players.add(p);
      p.addTournament(this);
    }
  }
}
```

```
public class Player {
  List tournaments = new List();
  void addTournament(Tournament t) {
    if(!tournaments.contains(t)) {
      tournaments.add(t);
      t.addPlayer(this);
    }
  }
}
```

5.5.3. Mapping Contracts to Exceptions

Since Java does not provide built-in support for contracts, we need to use the exception mechanism as building blocks for signaling and handling contract violations (using `try`, `catch` and `throw/s`).

A simple mapping would be to add code within the method body to check invariants, pre- and postconditions:

- Check **preconditions** at the beginning, if one is false, raise an exception.
- Check **postconditions** at the end of the method, if more than one postcondition is not satisfied, only the first detection is reported.
- Check **invariants** at the same time as postconditions.
- The checking code should be encapsulated into separate methods that can be called from subclasses.
- If checking slows down the program: omit checking for private and protected methods or even omit checking for postconditions and invariants. Focus on contracts for components with the longest life.

5.5.4. Mapping Object Model to a Persistent Storage Schema

For storing persistent objects in relational databases, we need to map the object model to a storage schema and provide infrastructure for converting from and to persistent storage.

The database **schema** describes the valid set of data records that can be stored in the database. Relational databases store both, the schema and the data in terms of tables. The *columns* of a table represent the *attributes*, each *row* represents one *data record*, with each *cell* representing a certain *value*.

A **primary key** of a table is one attribute (or a set of attributes) that uniquely identifies a certain data records (row). The set of attributes that could be used as a primary key are called **candidate keys**. A **foreign key** is an attribute that references the primary key of another table.

Mapping Classes and Attributes

We map each class to a table with the same name, for each attribute we add a column and each data record corresponds to an instance of the class. When mapping attributes, we need to select a data type for the (SQL) database column that completely stores the data of the corresponding Java type. For example, limiting the length of an attribute `name` to 25 characters enables us to use a column of type `VARCHAR(25)`.

A good way to assign the primary key of a table is to add an `id` column (unique integer value for each row). Since values of primary keys are references (foreign keys) stored in relating tables, change of a primary key value is problematic. The advantage of using an additional `id` column is, that this value does not need to be changed (in contrast to values of attributes like `name` or `email`).

Mapping Associations

The mapping of associations depends on the multiplicity of the association.

Buried associations implement one-to-one and one-to-many associations. *One-to-one* associations can easily implemented, using a foreign key in each table. For *one-to-many* associations, we add a foreign key to the table representing the class on the "many" end (see Figure 17).

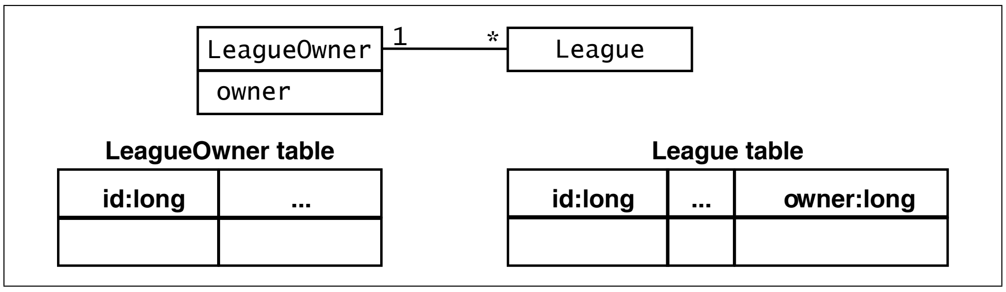


Figure 17: Buried association: only table `League` stores foreign keys.

Separate tables implement *many-to-many* associations, using a separate two-column table (association table) with foreign keys for both classes of the association. Note that one-to-one and one-to-many associations could be realized with an association table instead of buried association (that results in a database schema that is modifiable).

Mapping Inheritance Relationships

Relational databases do not directly support inheritance, but there are two main options for mapping an inheritance relationship to a database schema. Considering two subclasses `LeagueOwner` and `Player`, both inherit from the (abstract) superclass `User`, there are two ways to store these subclas objects.

Vertical mapping uses individual tables for superclass and subclasses. The superclass table includes a column for each attribute defined in the superclass. The subclass tables include columns for attributes that are only defined in the subclasses. That means, attributes of a `Player` instance are stored in the `User` table as well as in the `Player` table, depending on where they are defined. Both tables use the same `id` for this instance and the superclass has an additional column (e.g. called `role`) denoting the subclass that corresponds to the data record.

Horizontal mapping only uses tables for the subclasses. Hence, there is no table for `User`, but `Player` and `LeagueOwner` tables, that store all attributes of an object.

The trade-off between using a separate table for superclasses and *duplicating columns* in the subclass table is between *modifiability* and *response time.*

6. Testing

Testing is the process of finding differences between the expected behavior specified by system models and the observed behavior of the implemented system. The goal is to maximize the number of discovered faults, which then allows developers to correct them and increase the reliability of the system.

- **Failure** is any deviation of the observed behavior from the specified behavior.
- An **erroneous state** (error) means the system is in a state such that *further processing* by the system will lead to a *failure*, which then causes the system to deviate from its intended behavior.
- A **fault** (bug/defect) is a design or coding mistake that may *cause an erroneous state.*

Typical faults and errors are:

- faults in the user interface specification,
- algorithmic faults (missing initialization, incorrect branching condition),
- mechanical faults (operating temperature), or
- errors (wrong user input, `null` reference errors)

Testing is one way of dealing with faults. Fault handling, in more general, includes three techniques:

- **Fault avoidance** tries to prevent the insertion of faults into the system (before release). It includes *development methodologies*, *configuration management* and *verification.*
- **Fault detection** tries to identify erroneous states and find the underlying faults (before release). It includes *testing* and *debugging.*
- **Fault tolerance** techniques assume that a system can be released with faults and that system failures can be dealt with by recovering from them at runtime. It includes *exception handling* and *modular redundancy.*

In this section, we focus on fault detection techniques, including reviews (static analysis) and testing (dynamic analysis).

A **review** is the manual inspection of (parts of) the system without actually executing it. There are two types of reviews: *code walktroughs* (informal presentation of the API, code etc. to others) and c*ode inspections* (formal presentation of the component to others).

Testing is a fault detection technique that tries to create failure or erroneous states in a planned way. Different testing methods are described in *6.4 Activities.*

6.1. Test Cases

A test case is a set of inputs and expected results that exercises a test component with the purpose of causing failures and detecting faults.

A test case has five attributes:

- a **name** (e.g. `Test_UseCaseName`, `Test_CompACompB`),
- a **location** (path name of executable),
- an **input** (data or commands),
- an **oracle** (expected test results), and
- a **log** (output produced by the test).

Test cases can be modeled as object diagrams and relationships can be identified. *Aggregation* is used when a test case can be decomposed into a set of subtests. Two tests are related via the `precede` association when one test case must precede another one.

Test cases are classified into:

- **blackbox tests** (focus on the input/output behavior of the component and do not deal with the internal aspects or structure) and
- **whitebox tests** (focus on the internal structure of the component, independently form the particular input/output behavior).

6.2. Test Stubs and Drivers

A **test component** is a part of the system that can be isolated for testing (i.e. one or more objects or subsystems)

A **test driver** simulates the part of the system that *calls* the component under test. A **test stub** simulates a component that *is called* by the tested component. It must provide the same API as the method of the simulated component and must return a value compliant with the return result type of the method's type signature.

6.3. Model-Based Testing

The *System under test (SUT)* is a part of the **system model** we want to test. Since we can not test the SUT in isolation (it collaborates with other objects in the system model), our **test model**, derived from the the SUT, furthermore consists of test doubles.

Test doubles are *dummy objects* (fill parameter lists but are actually never used), *fake objects* (working implementation, but "shortcut"), *stubs* (provide always the same answer when invoked) and *mock objects* (mimic the behavior of the real object).

6.4. Activities

6.4.1. Unit Testing

Unit testing focuses on the individual components of the system (i.e. classes and subsystems). Unit testing reduces complexity (focus on smaller units of the system), makes it easier to pinpoint and correct faults and allows parallelism in the testing activities.

The minimal set of objects to be tested should be the participating objects in use cases. Subsystems should be tested only after each of the including classes have been tested.

The most important unit testing techniques are as follows.

Equivalence testing is a blackbox testing technique that minimizes the number of test cases by partitioning possible inputs into equivalence classes and selecting a test case (certain input) for each equivalence class.

Example: Considering an operation `getNumDaysInMonth(int month, int year)`, returning an integer value. We find three equivalence classes for the `month` parameter, there are months with 31 days (i.e. 1, 3, 5, 7, 8, 10, 12), months with 30 days (i.e. 4, 6, 9, 11) and February with 28 days. Since February can also have 29 days, we identify two equivalence classes for the `year` parameter, leap years (i.e. multiples of 4) and non-leap years. Hence, there are six possible combinations of equivalence classes for both parameters. Non-positive integers and integers lager than 12 are invalid values.

Boundary testing is a special case of equivalence testing that focuses on the conditions at the boundary of the equivalence classes. In our example, we forgot that years that are multiples of 100 are not leap years, unless they are multiple of 400, which results in two more classes. Other boundary cases include the months 0 and 13, which are at the boundaries of the invalid equivalence classes.

Path testing is a whitebox testing technique that exercises all possible paths through the code at least once. The starting point is the *flow of control*.

A flow graph is constructed by mapping (decision) statements (e.g. `if/else` statements, `while` loops) to nodes. Complete path testing consists of designing test cases such that each edge is traversed at least once. This is done by examining the conditions associated with each branch point and selecting one input for the `true` and one for the `false` branch. The minimum number of tests, necessary to cover all edges is equal to the number of independent paths through the flow graph[5]. In this case, *path coverage* amounts 100%.

[5]Defined as cyclomatic complexity (CC = edges - nodes + 2).

Since it is nearly impossible to reach 100% path coverage in non-trivial systems, there are three other metrics that discover less code but certain problems:

Statement coverage measures the percentage of exercised statements (`if-else` and `loop` statements are only traversed once).

Branch coverage measures the percentage of exercised branches (tests not all conditions in one branch).

Condition coverage measures the percentage of all atomic conditions that where set `true` and `false`.
- **Single:** every atomic condition must be once `true` and once `false`.
- **Multiple:** every combination of `true` and `false` for all atomic conditions.
- **Minimum:** every condition (atomic or composite) must be at least once true and once false.

Since these whitebox testing methods can detect only faults resulting from exercising a path in the program, faulty statements like:

```
if(month==1 || month==3 || month==5 || month==7 || month==10 || month==12)
```

are not detected, if there is no appropriate input, causing failure. In this example, the expression `month==8` is missing.

State-based testing compares the resulting state of the system with the expected state. Test cases are derived from state machine diagrams for the class. For each state, a representative set of stimuli is derived for each transition (similar to equivalence testing). The attributes of the class are then instrumented and tested after each stimuli has been applied to ensure that the class has reached the specific state.

Polymorphism testing addresses failure that may occur due to dynamic binding. When applying the path testing technique to an operation that uses polymorphism, we need to consider all dynamic bindings, one for each message that could be sent.

For example, an operation `Network.send()` can be bound to either `Ethernet.send()`, `WaveLan.send()` or the `Umts.send()` method (each class `extends Network`) depending on the certain class of the object on which the `send()` methods is invoked.

6.4.2. Integration Testing

Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Two or more components are integrated and tested (double, triple or quadruple test), and when no new faults are revealed, additional components are added to the group.

Horizontal Integration

Horizontal integration testing strategies originally were devised by assuming that the system decomposition is hierarchical and that each of the components belongs to layers ordered with respect to the "call" association. These strategies, however, can be easily adapted to non-hierarchical system decompositions.

Figure 18 depicts an example of a hierarchical system decomposition with three layers

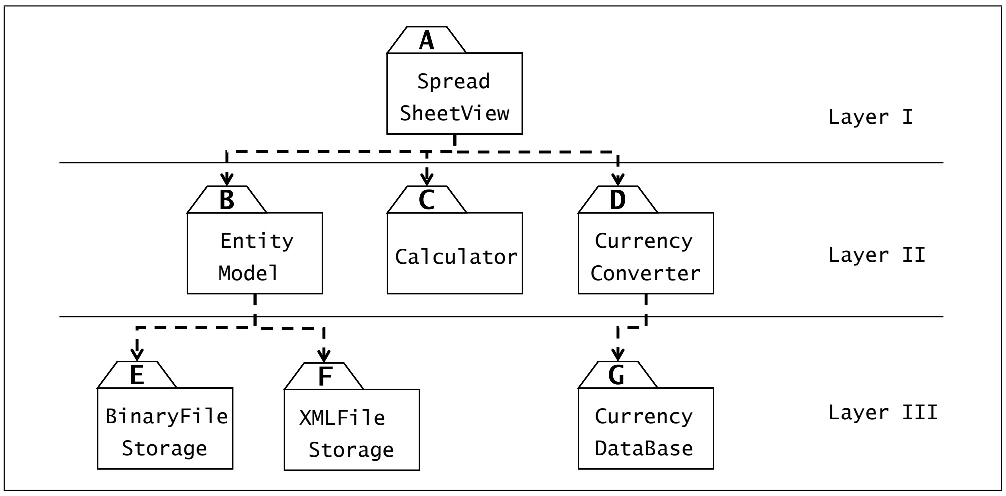


Figure 18: Packages (A-G) represent the individual components to be tested.

The following strategies differ from each other, in the order and combination, they test the individual system components.

The big bang testing strategy assumes that all components are first tested individually and then tested together as a single system.

- test **A**, **B**, **C**, **D**, **E**, **F**, **G** individually
- test (A-G) together

If a test uncovers a failure, it is difficult to pinpoint the specific component (or combination) responsible for that failure.

The bottom-up testing strategy first tests each component of the bottom layer individually and then integrates them with components of the next layer up (until all layers are combined and tested together).

- test **E**, **F**, **C**, **G**
- test (B,E,F)
- test (D,G)
- test (A-G)

Test drivers are used to simulate the components of *higher* layers that have not yet been integrated.

The top-down testing strategy unit tests the components of the top layer first and integrates the components of the next layer down (until all layers are combined and tested together).

- test layer 1: **A**
- test layer 1-2: (A,B,C,D)
- test layer 1-3: (A-G)

Test stubs are used to simulate the components of *lower* layers that have not yet been integrated. Since this strategy tests the top layer first, (user) interface faults can be found more easily.

The sandwich testing strategy combines both, top-down and bottom-up strategy, attempting to make use of the best of both. The tester must be able to map the subsystem decomposition into *three layers*, a target layer ("the meat"), a layer above and one below the target layer ("slices of bread").

- test **A**, **E**, **F**, **G**
- test (A,B,C,D)
- test (B,E,F)
- test (D,G)
- test (A-G)

Using the target layer as the focus of attention, top-down testing and bottom-up testing can now be done in parallel. As a result, test stubs and drivers need not to be written for the top and bottom layers (during a top-down/bottom-up testing), because the actual components from the target layer are used.

The **modified sandwich testing** strategy tests the three layers individually before combining them in incremental tests with one another. The *individual* layer tests consist of a *top layer* test with subs (for the target layer), a *target layer* test with drivers and stubs (replacing the top and bottom layers) and a *bottom layer* test (with drivers for the target layer). The combined layer tests consist of the top layer accessing the target layer and the bottom layer accessed by the target layer.

Vertical Integration

Vertical integration strategies (e.g. used by Extreme Programming (XP)) focus on early integration. For a given use case, the needed parts of each component are identified, developed in parallel and then integration tested.

6.4.3. System Testing

Once components have been integrated, system testing ensures that the complete system complies with the requirements. We discuss three different black-box testing methods.

Functional testing validates *functional requirements*, test cases are derived from the use case model. Since it is usually impossible to test all use cases for all valid and invalid input, the goal is to select those tests that are relevant to the user and have high probability of uncovering a failure.

Performance testing validates *non-functional requirements*. This includes among others, *stress testing* (simultaneous requests, any bottlenecks?), *volume testing* (large amounts of data), *security testing*, *timing testing* (e.g. response time) and *recovery tests* (ability to recover from erroneous states).

Acceptance testing validates the *client's expectations*. Test cases that represent typical conditions under which the system should operate are prepared by the client. An *alpha test* runs at the developer's environment (bugs might be fixed immediately), a *beta test* runs at the client's environment.

6.4.4. Regression Testing

Object-oriented development is an iterative process. When modifying a component, developers design new unit tests exercising the new feature under consideration. Once the modified component passes the unit tests, developers can be reasonably confident about the changes within the component. However, the should not assume that the rest of the system will work with the modified component. Integration tests that are rerun on the system to produce failures are called regression tests.

A. Analysis and Design Documents

Requirements Analysis Document (RAD)

The results of *requirements elicitation* and the *analysis* activities are documented in the RAD. This document completely describes the system in terms of functional and nonfunctional requirements. The sections *Object model* and *Dynamic models* are completed during analysis. The RAD, once published, is baselined and put under configuration management.

Requirements Analysis Document

1. Introduction
    - 1.1 Purpose of the system
    - 1.2 Scope of the system
    - 1.3 Objectives and success criteria of the project
    - 1.4 Definitions, acronyms and abbreviations
    - 1.5 References
    - 1.6 Overview
2. Current system
3. Proposed system
    - 3.1 Overview
    - 3.2 Functional requirements
    - 3.3 Nonfunctional requirements
        - 3.3.1 Usability
        - 3.3.2 Reliability
        - 3.3.3 Performance
        - 3.3.4 Supportability
        - 3.3.5 Implementation
        - 3.3.6 Interface
        - 3.3.7 Packaging
        - 3.3.8 Legal
    - 3.4 System models
        - 3.4.1 Scenarios
        - 3.4.2 Use case models
        - 3.4.3 *Object model*
            - 3.4.3.1 *Data dictionary*
            - 3.4.3.2 *Class diagrams*
        - 3.4.4 *Dynamic model*
        - 3.4.5 User interface
4. Glossary

System Design Document (SDD)

The SDD is used to define interfaces between teams of developers and serves as reference when architecture-level decisions need to be revised.

System Design Document

1. Introduction
    - 1.1 Purpose of the system
    - 1.2 Design goals
    - 1.3 Definitions, acronyms and abbreviations
    - 1.4 References
    - 1.5 Overviews
2. Current software architecture
3. Proposed software architecture
    - 3.1 Overview
    - 3.2 Subsystem decomposition
    - 3.3 Hardware/software mapping
    - 3.4 Persistent data management
    - 3.5 Access control and security
    - 3.6 Global software control
    - 3.7 Boundary conditions
4. Subsystem services
5. Glossary

B. Abbreviations

API	Application Programmer Interface
MVC	Model/View/Controller
OCL	Object Constraint Language
OOSE	Object-Oriented Software Engineering
RAD	Requirements Analysis Document
SDD	System Design Document
SUT	System under test
UML	Unified Modeling Language
XP	Extreme Programming