System Design_cont

The system design goals are to identify design goals, to decompose the system into subsystems, and to refine the subsystem decomposition until all design goals are addressed. Now, we focus on design activities that address the design goals:

- *Selection of off-the-shelf and legacy components*,
- *Mapping of subsystem to hardware,*
- *Design of a persistent data management infrastructure,*
- *Specification of an access control policy,*
- *Design of the global control flow,*
- *Handling of boundary conditions,*

An Overview of System Design Activities

Design goals guide the decisions to be made by developers especially when trade-offs are needed. Developers divide the system into manageable pieces to deal with complexity. Each subsystem is assigned to a team and realized independently. For this to be possible, developers need to address system-wide issues when decomposing the system. They need to address the following issues:

- *Hardware/software mapping:*
    - What is the hardware configuration of the system?
    - Which node is responsible for which functionality?
    - How is communication between nodes realized?
    - Which services are realized using existing software components?
    - How are these components encapsulated?

- *Data management:*
    - Which data should be persistent?
    - Where should persistent data be stored?
    - How are they accessed?

  These issues must be addressed consistently at the system level. Often, this leads to the selection of a database management system and of an *additional subsystem* dedicated to the management of persistent data.

- *Access control:*
    - Who can access which data?
    - Can access control change dynamically?
    - How is access control specified and realized?

- *Control flow:*
    - How does the system sequence operations?
    - Is the system event driven?
    - Can it handle more than one user interaction at a time?

- *Boundary* conditions
  - How is the system initialized and shut down?
  - How are exceptional cases handled?

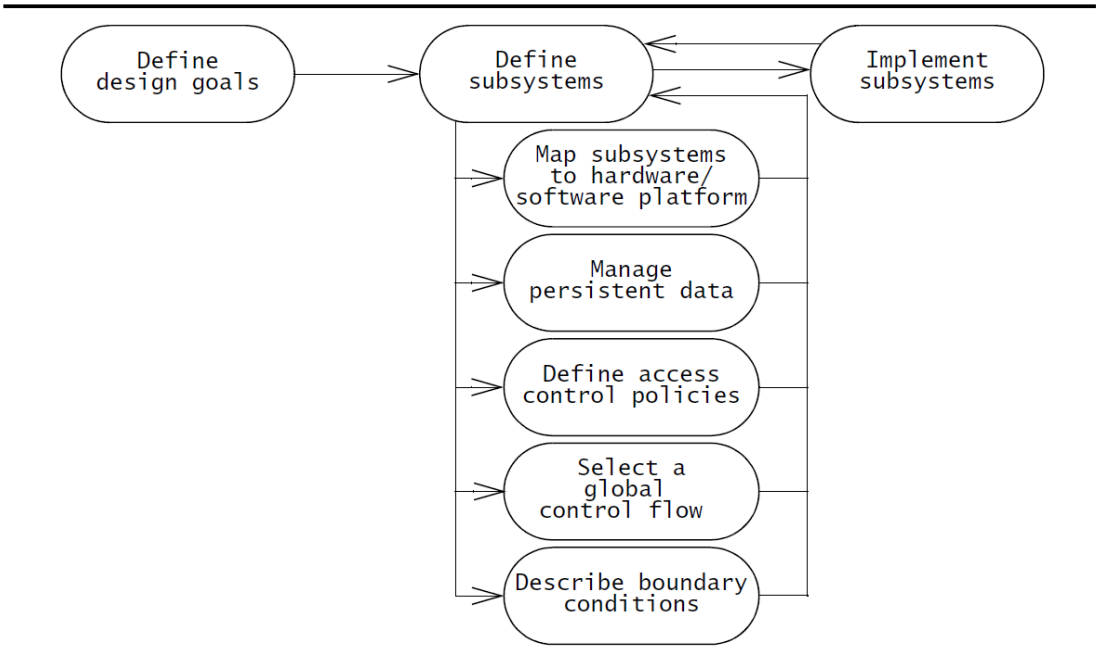


Figure 7-1 The activities of system design (UML activity diagram)

Mapping Subsystems to Processors and Components

Selecting a hardware configuration and a platform

Many systems run on more than one computer and depend on access to an intranet or to the Internet. The use of multiple computers can address high-performance needs and interconnect multiple distributed users. Consequently, we need to examine carefully the allocation of subsystems to computers and the design of the infrastructure for supporting communication between subsystems. These computers are modeled as nodes in UML deployment diagrams. The hardware mapping activity has significant impact on the performance and complexity of the system; therefore, we perform it early in system design.

Selecting a hardware configuration also includes selecting a virtual machine onto which the system should be built. The virtual machine includes the operating system and any software components that are needed, such as a database management system or a communication package.

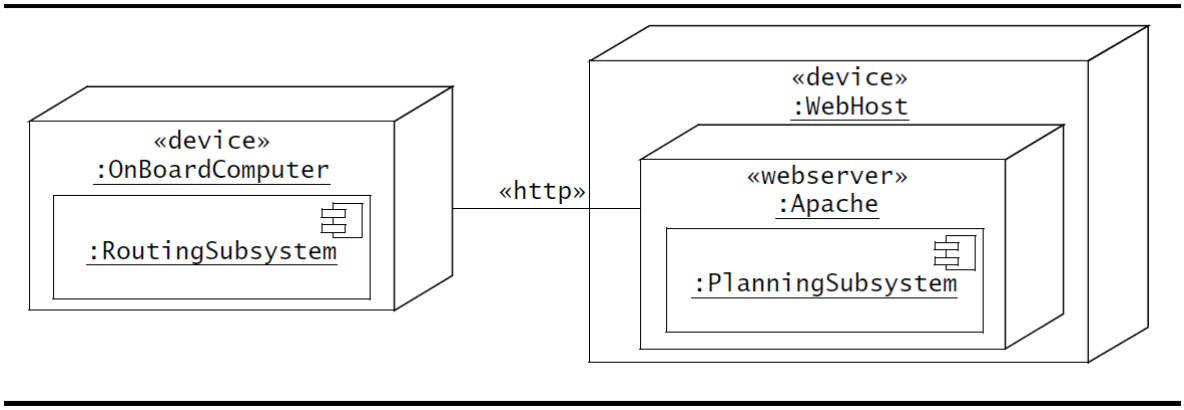
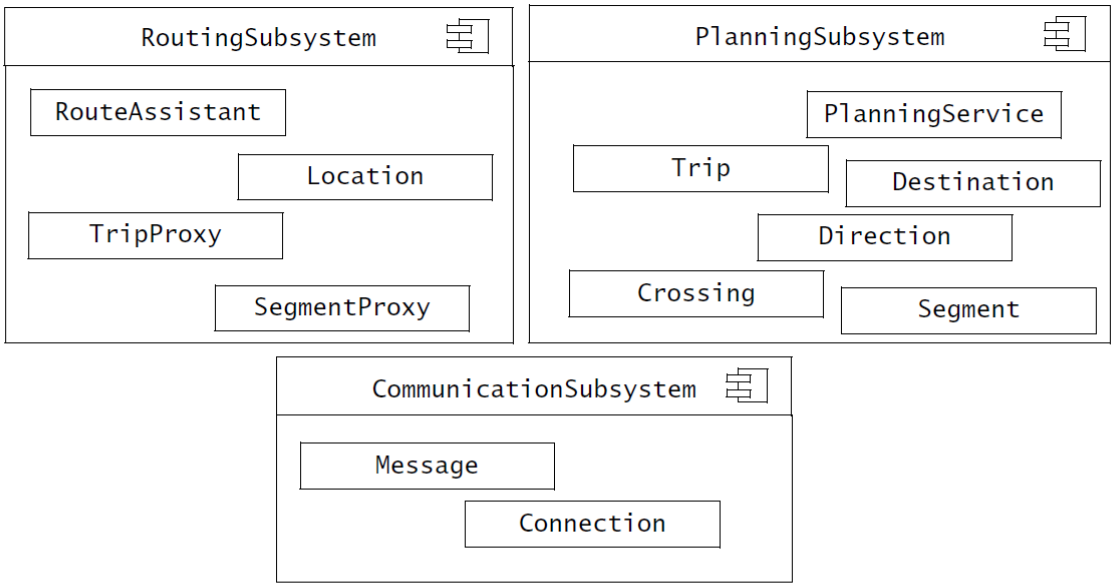


Figure 7-4 Allocation of MyTrip subsystems to devices and execution environments (UML deployment diagram). RoutingSubsystem runs on the OnBoardComputer; PlanningSubsystem runs on an Apache server.

Allocating objects and subsystems to nodes

Once the hardware configuration has been defined and the virtual machines selected, objects and subsystems are assigned to nodes. This often triggers the identification of new objects and subsystems for transporting data among the nodes.

In general, allocating subsystems to hardware nodes enables us to distribute functionality and processing power where it is most needed. Unfortunately, it also introduces issues related to storing, transferring, replicating, and synchronizing data among subsystems. For this reason, developers also select the components they will use for developing the system.



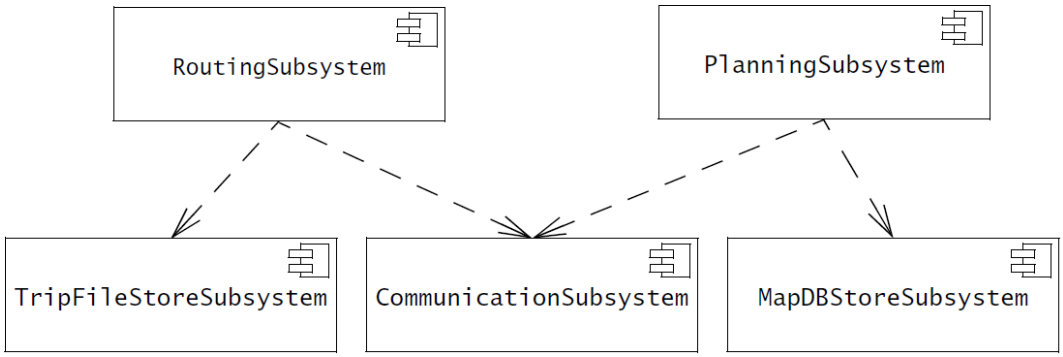
CommunicationSubsystem	The CommunicationSubsystem is responsible for transporting objects from the PlanningSubsystem to the RoutingSubsystem.
Connection	A Connection represents an active link between the Planning-Subsystem and the RoutingSubsystem. A Connection object handles exceptional cases associated with loss of network services.
Message	A Message represents a Trip and its related Destinations, Segments, Crossings, and Directions, encoded for transport.

Figure 7-5 Revised design model for MyTrip (UML component diagram).

Identifying and Storing Persistent Data

Persistent data outlive a single execution of the system. For example, at the end of the day, an author saves his work into a file on a word processor. The file can then be reopened later. The word processor need not run for the file to exist. Similarly, information related to employees, their employment status, and their paychecks live in a database management system. This allows all the programs that operate on employee data to do so consistently. Moreover, storing data in a database enables the system to perform complex queries on a large data set (e.g., the records of several thousand employees).

Where and how data is stored in the system affects system decomposition. In some cases, for example, in a repository architectural style, a subsystem can be completely dedicated to the storage of data. The selection of a specific database management system can also have implications on the overall control strategy and concurrency management.



TripFileStoreSubsystem	The TripFileStoreSubsystem is responsible for storing trips in files on the onboard computer. Because this functionality is only used for storing trips when the car shuts down, this subsystem only supports the fast storage and loading of whole trips.
MapDBStoreSubsystem	The MapDBStoreSubsystem is responsible for storing maps and trips in a database for the PlanningSubsystem. This subsystem supports multiple concurrent Drivers and planning agents.

Figure 7-6 Subsystem decomposition of MyTrip after deciding on the issue of data stores (UML component diagram).

Identifying persistent objects

First, we identify which data must be persistent. The entity objects identified during analysis are obvious candidates for persistency. In MyTrip, Trips and their related classes (Crossing, Destination, PlanningService, and Segment) must be stored. Note that not all entity objects must be persistent. For example, Location and Direction are constantly recomputed as the car moves. Persistent objects are not limited to entity objects, however. In a multi-user system, information related to users (e.g., Drivers) is persistent, as well as some attributes of the boundary objects (e.g., window positions, user interface preferences, state of long-running control objects). In general, we can identify persistent objects by examining all the classes that must survive system shutdown, either in case of a controlled shutdown or an unexpected crash. The system will then restore these long-lived objects by retrieving their attributes from storage during system initialization or on demand as the persistent objects are needed.

Selecting a storage management strategy

Once all persistent objects are identified, we need to decide how these objects should be stored. The decision for storage management is more complex and is usually dictated by nonfunctional requirements:

- Should the objects be retrieved quickly?
- Must the system perform complex queries to retrieve these objects?
- Do objects require a lot of memory or disk space?

In general, there are currently three options for storage management:

- *Flat files*.
  Files are the storage abstractions provided by operating systems. The application stores its data as a sequence of bytes and defines how and when data should be retrieved.

- *Relational database*.
  A relational database provides data abstraction at a higher level than flat files. Data are stored in tables that comply with a predefined type called a **schema**.

- *Object-oriented database*.
  An object-oriented database provides services like a relational database. Unlike a relational database, it stores data as objects and associations. In addition to providing a higher level of abstraction (and thus reducing the need to translate between objects and storage entities), object-oriented databases provide developers with inheritance and abstract data types. Object-oriented databases significantly reduce the time for the initial development of the storage subsystem. However, they are slower than relational databases for typical queries and are more difficult to tune.

Providing Access Control

In multi-user systems, different actors have access to different functionality and data. For example, an everyday actor may only access the data it creates, whereas a system administrator actor may have unlimited access to system data and to other users' data. During analysis, we modeled these distinctions by associating different use cases to different actors. During system design, we model access by determining which objects are shared among actors, and by defining how actors can control access. Depending on the security requirements of the system, we also define how actors are authenticated to the system (i.e., how actors prove to the system who they are) and how selected data in the system should be encrypted.

Defining access control for a multi-user system is usually more complex than in the MyTrip example presented in Bruegge's book. In general, we need to define for each actor which operations they can access on each shared object. For example, a bank teller may post credits and debits up to a predefined amount. If the transaction exceeds the predefined amount, a manager must approve the transaction. Managers can examine the branch statistics; but cannot access the statistics of other branches. Analysts can access information across all branches of the corporation but cannot post transactions on individual accounts. We model access on classes with an access matrix. The rows of the matrix represent the actors of the system. The columns represent classes whose access we control.

We can represent the access matrix using one of three different approaches: global access table, access control list, and capabilities.

- A **global access table** represents explicitly every cell in the matrix as a tuple. Determining if an actor has access to a specific object requires looking up the corresponding tuple. If no such tuple is found, access is denied.

- An **access control list** associates a list of pairs with each class to be accessed. Empty cells are discarded. Every time an object is accessed, its access list is checked for the corresponding actor and operation. An example of an access control list is the guest list for a party. A butler checks the arriving guests by comparing their names against names on the guest list. If there is a match, the guests can enter; otherwise, they are turned away.
- A **capability** associates a pair (class, operation) with an actor. A capability allows an actor access to an object of the class described in the capability. Denying a capability is equivalent to denying access. An example of a capability is an invitation card for a party. In this case, the butler checks if the arriving guests hold an invitation for the party. If the invitation is valid, the guests are admitted; otherwise, they are turned away. No other checks are necessary.

Designing the Global Control Flow

Control flow is the sequencing of actions in a system. In object-oriented systems, sequencing actions includes deciding which operations should be executed and in which order. These decisions are based on external events generated by an actor or on the passage of time. Control flow is a design problem. During analysis control flow is not an issue, because we assume that all objects are running simultaneously executing operations any time they need to. During system design, we need to consider that not every object has the luxury of running on its own processor. There are three possible control flow mechanisms:

- **Procedure-driven control.** Operations wait for input whenever they need data from an actor. This kind of control flow is mostly used in legacy systems and systems written in procedural languages. It introduces difficulties when used with object-oriented languages. As the sequencing of operations is distributed among a large set of objects, it becomes increasingly difficult to determine the order of inputs by looking at the code.

- **Event-driven control**. A main loop waits for an external event. Whenever an event becomes available, it is dispatched to the appropriate object, based on information associated with the event. This kind of control flow has the advantage of leading to a simpler structure and to centralizing all input in the main loop. However, it makes the implementation of multi-step sequences more difficult to implement.

- **Threads.** Threads are the concurrent variation of procedure-driven control: The system can create an arbitrary number of threads, each responding to a different event. If a thread needs additional data, it waits for input from a specific actor. This kind of control flow is the most intuitive of the three mechanisms. However, debugging threaded software requires good tools: preemptive thread schedulers introduce nondeterminism and, thus, make testing harder.

Identifying Boundary Conditions

In previous sections, we dealt with designing and refining the system decomposition. We now have a better idea of how to decompose the system, how to distribute use cases among subsystems, where to store data, and how to achieve access control and ensure security. We still need to examine the **boundary conditions** of the system—that is, to decide how the system is started, initialized, and shut down—and we need to define how we deal with major failures such as data corruption and network outages, whether they are caused by a software error or a power outage. Uses cases dealing with these conditions are called **boundary use cases**. It is common that boundary use cases are not specified

during analysis or that they are treated separately from the common use cases. For example, many system administration functions can be inferred from the everyday user requirements (registering and deleting users, managing access control), whereas, many other functions are consequences of design decisions (cache sizes, location of database server, location of backup server) and not of requirement decisions. In general, we identify boundary use cases by examining each subsystem and each persistent object:

- **Configuration**. For each persistent object, we examine in which use cases it is created or destroyed (or archived). For objects that are not created or destroyed in any of the common use cases.

- **Start-up and shutdown**. For each component, we add three use cases to start, shutdown, and configure the component. Note that a single use case can manage several tightly coupled components.

- **Exception handling**. For each type of component failure (e.g., network outage), we decide how the system should react (e.g., inform users of the failure). We document each of these decisions with an exceptional use case that extends the relevant common uses cases identified during requirements elicitation. Note that, when tolerating the effects of a failure, the handling of an exceptional condition can lead to changing the system design instead of adding an exceptional use case.

An **exception** is an event or error that occurs during the execution of the system. Exceptions are caused by three different sources:

- *A hardware failure.* Hardware ages and fails. A hard disk crash can lead to the permanent loss of data. The failure of a network link, for example, can momentarily disconnect two nodes of the system.
- *Changes in the operating environment*. The environment also affects the way a system works. A wireless mobile system can fail connectivity if it is out of range of a transmitter. A power outage can bring down the system, unless it is fitted with back-up batteries.
- *A software fault.* An error can occur because the system or one of its components contains a design error. Although writing bug-free software is difficult, individual subsystems can anticipate errors from other subsystems and protect against them.

Exception handling is the mechanism by which a system treats an exception. In the case of a user error, the system should display a meaningful error message to the user so that she can correct her input. In the case of a network link failure, the system should save its temporary state so that it can recover when the network comes back online.

Reviewing System Design

Like analysis, system design is an evolutionary and iterative activity. Unlike analysis, there is no external agent, such as the client, to review the successive iterations and ensure better quality. This quality improvement activity is still necessary, and project managers and developers need to organize a review process to substitute for it. Several alternatives exist, such as using the developers who were not involved in system design to act as independent reviewers, or to use developers from another project to act as a peer review. These review processes work only if the reviewers have an incentive to discover

and report problems. In addition to meeting the design goals that were identified during system design, we need to ensure that the system design model is correct, complete, consistent, realistic, and readable. The system design model is **correct** if the analysis model can be mapped to the system design model. You should ask the following questions to determine if the system design is correct:

- Can every subsystem be traced back to a use case or a nonfunctional requirement?
- Can every use case be mapped to a set of subsystems?
- Can every design goal be traced back to a nonfunctional requirement?
- Is every nonfunctional requirement addressed in the system design model?
- Does each actor have an access policy?
- Is every access policy consistent with the nonfunctional security requirement?

The model is **complete** if every requirement and every system design issue has been addressed. You should ask the following questions to determine if the system design is complete:

- Have the boundary conditions been handled?
- Was there a walkthrough of the use cases to identify missing functionality in the system design?
- Have all use cases been examined and assigned a control object?
- Have all aspects of system design been addressed?
- Do all subsystems have definitions?

The model is **consistent** if it does not contain any contradictions. You should ask the following questions to determine if a system design is consistent:

- Are conflicting design goals prioritized?
- Does any design goal violate a nonfunctional requirement?
- Are there multiple subsystems or classes with the same name?
- Are collections of objects exchanged among subsystems in a consistent manner?

The model is **realistic** if the corresponding system can be implemented. You ask the following questions to determine if a system design is realistic:

- Are any new technologies or components included in the system?
- Was the appropriateness or robustness of these technologies or components evaluated? How?
- Have performance and reliability requirements been reviewed in the context of subsystem decomposition?
- Have concurrency issues (e.g., contention, deadlocks) been addressed?

The model is **readable** if developers not involved in the system design can understand the model. You should ask the following questions to ensure that the system design is readable:

- Are subsystem names understandable?
- Do entities with similar names denote similar concepts?
- Are all entities described at the same level of detail?

Documenting System Design

System design is documented in the System Design Document (SDD). It describes design goals set by the project, subsystem decomposition (with UML class diagrams), hardware/software mapping (with UML

deployment diagrams), data management, access control, control flow mechanisms, and boundary conditions. The SDD is used to define interfaces between teams of developers and serve as a reference when architecture-level decisions need to be revisited. The audience for the SDD includes the project management, the system architects (i.e., the developers who participate in the system design), and the developers who design and implement each subsystem.

The first section of the SDD is an *Introduction*. Its purpose is to provide a brief overview of the software architecture and the design goals. It also provides references to other documents and traceability information (e.g., related requirements analysis document, references to existing systems, constraints impacting the software architecture).

The second section, *Current software architecture*, describes the architecture of the system being replaced. If there is no previous system, this section can be replaced by a survey of current architectures for similar systems. The purpose of this section is to make explicit the background information that system architects used, their assumptions, and common issues the new system will address.

The third section, *Proposed system architecture*, documents the system design model of the new system. It is divided into seven subsections:

1. Overview
2. Subsystem decomposition
3. Hardware/software mapping
4. Persistent data management
5. Access control and security
6. Global software control
7. Boundary conditions