Chapter 8, System Design (2/2)

Object-Oriented Software Construction

Armin B. Cremers, Sascha Alda & Tobias Rho
(based on Bruegge & Dutoit)

b-it
Overview

♦ Where are we right now? (last Lecture):
  ♦ Overview of System Design
  ♦ Design Goals
  ♦ Subsystem Decomposition
    ♦ Architectural Styles (Client-Server, Peer-to-Peer, … )

♦ Overview of this Lecture:
  ♦ System Design continued:
    ♦ Refinement of Decomposition
    ♦ Involvement of more technical issues
Software Lifecycle Activities

Requirements Elicitation
Analysis
System Design
Object Design
Implementation
Testing

Use Case Model
Application Domain Objects
Sub-systems
Solution Domain Objects
Source Code
Test Cases

Expressed in Terms of
Structured by
Realized by
Implemented by
Verified by

Armin B. Cremers, Sascha Alda & Tobias Rho (based on Bruegge & Dutoit)
Object Oriented Software Construction
The Purpose of System Design

- Input: Analysis Model
- Output: System Model
- Use Divide and Conquer
  - We model the new system to be developed as a set of subsystems
Activities necessary for System Design

♦ Identify Design Goals
  ◆ Describes and prioritizes the qualities that are important for the system and that developers should optimize
  ◆ Derived from non-functional requirements
  ◆ Goals must be trade off against each other

♦ Design initial Subsystem Decomposition
  ◆ Results into a set of loosely dependent parts which make up the system
  ◆ Usage of architectural styles
  ◆ Can be realized by individual teams

♦ Refine Decomposition to address design goals
  ◆ Iterate and refine Design until all designs goals are met
  ◆ Incorporate more technical issues
Aspects for refining Subsystem Decomposition

♦ Mapping of Subsystem to Hardware
  ◆ Determine the hardware configuration

♦ Selection of Off-The-Shelf Components
  ◆ Purchase existing components to realize subsystems more economically

♦ Design of Persistent Data Management
  ◆ Realize Storage System for objects that outlive a single execution of the system

♦ Define Global Control flow
  ◆ Determine the sequence of operations within the subsystems

♦ Define Access Control
  ◆ Protect Shared objects for concurrent access

♦ Define Boundary Conditions
  ◆ Determine conditions for system initialization and shut-down
Hardware Mapping

♦ This activity addresses two questions:
  ♦ What is the most appropriate hardware environment?
  ♦ How are objects and subsystems mapped on the chosen hardware?
    ♦ Mapping Objects onto Reality: Processor, Memory, Input/Output
    ♦ Mapping Associations onto Reality: Connectivity

♦ Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints.
Hardware Mapping:
Mapping the Objects

♦ Processor issues:
  ♦ Is the computation rate too demanding for a single processor?
  ♦ Can we get a speedup by distributing tasks across several processors?
  ♦ How many processors are required to maintain steady state load?

♦ Memory issues:
  ♦ Is there enough memory to buffer bursts of requests?

♦ I/O issues:
  ♦ Does the response time exceed the available communication bandwidth between subsystems?
Hardware Mapping: Mapping the Subsystems Associations

- Describe the *physical connectivity* of the hardware:
  - Which associations in the object model are mapped to physical connections?
  - Which of the client-supplier relationships in the analysis/design model correspond to physical connections?

- Describe the *logical connectivity* (subsystem associations):
  - Identify associations that do not directly map into physical connection. Distinction:
    - Logical Connection within a local system
    - Logical Connection within a distributed system
    - How should these associations be implemented?
Hardware Mapping: Logical vs. Physical Connectivity

Bidirectional associations for each layer
Hardware Mapping: Connectivity Mapping Questions

♦ What is the connectivity among physical units?

♦ What is the appropriate communication protocol between the subsystems, especially Distributed Systems?
  ♦ Function of required bandwidth, latency and desired reliability, desired quality of service (QoS)

♦ Is certain functionality already available in hardware?
Hardware Mapping:

♦ New problems for allocating subsystems to *distributed hardware nodes* have to be faced (on a higher level)
  ♦ Transferring, replicating, or synchronizing data
  ♦ Exception Handling
  ♦ → New Subsystems have to be involved
Software Mapping: Selecting additional Software

♦ Selection of existing software subsystems (components, services, frameworks) that realize technical aspects
  ♦ Communication subsystems
  ♦ Data Management subsystems

♦ Selecting a Virtual Machine:
  ♦ Environment including operating system and any additional software components that are needed (e.g. DBMS)
  ♦ Reduces the distance between the system and hardware
  ♦ Reduction of development work (concentration on business code)
System design must model static and dynamic structures (dependencies among subsystems):

- **Component Diagrams for static structures**
  - show the structure at *design time* or *compilation time*

- **Deployment Diagrams for dynamic structures**
  - show the structure of the *run-time* system
Component Diagram

- Component Diagram
  - A graph of components connected by dependency relationships.
  - Shows static dependencies among software components.
- Dependencies are shown as dashed arrows from the client component to the supplier component.
  - The kinds of dependencies are implementation language specific.

Diagram:
- FireFox
- WebServer
- Database
Component Diagram

- Components can be refined to include information about the interfaces they provide and the classes they contain:

```
Diagram:
- WebServer
  - URI
  - HttpRequest
  - DBQuery

Edges:
- GET
- POST

External Component:
- FireFox
```
Component Diagram Example

Scheduler -> reservations
Planner -> update
GUI
Deployment Diagram

- Deployment diagrams are useful for showing a system design after the following decisions are made:
  - Subsystem decomposition
  - Selection of additional Software Components
  - Hardware Mapping

- A deployment diagram is a graph of nodes connected by dynamic communication associations:
  - Nodes are shown as 3-D boxes.
  - Nodes may contain component instances.
  - Components may contain objects (indicating that the object is part of the component)
Deployment Diagram Example

Compile Time Dependency

Runtime Dependency

Armin B. Cremers, Sascha Alda & Tobias Rho (based on Bruegge & Dutoit)  
Object-Oriented Software Construction
Connecting subsystems yields additional problems:

- Subsystems are implemented in different programming languages
- Subsystems does not fulfill its intended task, must be exchanged
- The intention (semantics) cannot be derived exactly from the interface (Example: difference between POST and GET?!)
- Subsystems are built on different communication layer

How to achieve interoperable subsystems?
Software Mapping: Demand: Interoperability

♦ Interoperability: The ability of two or more systems or components to exchange information and to use the information that has been exchanged

IEEE Glossary, 1990

♦ Interoperability denote systems that work together

Microsoft Webpage

♦ (Selected) Requirements for interoperable Systems:
  ♦ Standards for Communication and Interfaces
  ♦ Semantic Descriptions for Interfaces
  ♦ Mediator Components
  ♦ Dynamic Architectural Styles
Software Mapping: Standards

- Standards for integrating *distributed* Subsystems:
  - Middleware Frameworks (CORBA, RMI, DCOM, SOAP)

![Diagram]

- Interface Definition Languages (IDL) are used to formulate Interfaces of Subsystems in an language neutral way
- Compiler transforms IDL description to concrete components (Java, C++) and generates Stubs and Skeletons
Software Mapping: Semantic Descriptions

♦ Semantic Descriptions (Annotations) are instrumental to clarify the intention of a subsystem

Provider

- requestData()
- requestSession()
- requestNewSession()
- closeSession()
- query()
- getSemantics()

Questions:
- Order?
- Any Iterations()?
- (Exact) Purpose?
- Dependencies?

♦ Hardly implemented in standard technologies, only in academic fields or in terms of proprietary solutions

♦ New Approach: **Ontology**: Notation for describing process aspects in a common way

- To-Date: Promising, but also hardly implemented
Software Mapping: Mediators

- Problem: Incompatible Interfaces and/or different data formats (syntacs):

```plaintext
Generate RandomNumber

Produces „one“,
„two“, „three“...

getNextNumber()

German Number Printer

Expects „eins“,
„zwei“, „drei“...
```
Software Mapping: Mediators

- Problem: Incompatible Interfaces and/or different data formats (syntax):
- Solution: Mediator (implements both interfaces and transforms data in a format understandable for the client)

```java
private GenerateRandomNumber myRandom;

public String getNextNumber() {
    String input = myRandom.getNextNumber();
    if (input == "one") return "eins";
    if (input == "two") return "zwei";
}
```
Software Mapping: Dynamic Architectural Styles

- Problem: New Subsystems must be integrated during run-time
- Solution: Service-Oriented Architectures (SOA) allow for looking up for (new) subsystems (services) in a directory
Data Management

- Some objects in the models need to be persistent
  - In particular Entity Objects
  - Boundary Objects as well (e.g. user preferences in forms)
- A persistent object can be realized with one of the following
  - Files
    - Cheap, simple, permanent storage
    - Low level (Read, Write)
    - Applications must add code to provide suitable level of abstraction
  - Database
    - Powerful, scalable, portable
    - Supports multiple writers and readers
**File or Database?**

- **When should you choose a file?**
  - For extensive data (images)
  - For lots of raw data (event trace)
  - Temporary Data that is kept only for a short time

- **When should you choose a database?**
  - Data that require access at fine levels of details by multiple users and/or applications (concurrent access)
  - Data that must be ported across multiple platforms (heterogeneous systems)
  - Data management requires a lot of infrastructure?
Database Management System

- A collection of routines that enables you to store, modify, and extract information from a database
- Main functions (realization depends on Vendor)
  - Concurrent (write) access to the stored data
  - Transaction Management
  - Rollback Mechanisms
  - Crash Recovery
  - Optimizer for complex Queries
  - Authorization
Relational Databases

- Data are stored in tables that comply with a predefined type called a (relational) schema
  - Column represents an attribute of a schema
  - Row represents data item as a tuple of attribute values
  - Several tuples in different tables represent the attributes of an object
- Mapping rules to convert objects to relation scheme (not trivial)
- Ideal for querying large data sets with complex queries
- SQL is the standard language defining and manipulating tables (loose coupling)
- Leading commercial databases:
  - DB2 (IBM)
  - Oracle
  - MySQL
Object-Oriented Databases

- Support of all fundamental object modeling concepts
  - Stores data as objects and associations
  - Classes, Attributes, Methods, Associations, Inheritance
- Reduce the time for the initial development of storage subsystem:
  - Easy Mapping of Object model to Database schema
- Disadvantage:
  - Slower than relational databases even for simple queries
- Commercial Vendor
  - ObjectStore
  - Realization as “add-ons” for Relational DB (DB2) → Sense?
In multi-user systems different actors have access to different functionality and data.
  
  Example:
  - System Admin has unlimited access to any data
  - Client only has read access to data

Definition of access right during analysis:
  - Associate different use cases with different actors.

During system:
  - Determining which objects are shared among actors.
  - Define access rights for each objects (→ access matrix)

Subtopics here:
  - Authentication
  - Encryption
Access Matrix

♦ We model access on classes with an access matrix.
  ♦ The rows of the matrix represent the actors of the system
  ♦ The column represent classes whose access we want to control.

♦ **Access Right:** An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.
## Access Matrix: Example

<table>
<thead>
<tr>
<th>Objects</th>
<th>PressRelease</th>
<th>Cashflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>read()</td>
<td></td>
</tr>
<tr>
<td>Manager</td>
<td>read()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>write()</td>
<td>compute()</td>
</tr>
<tr>
<td></td>
<td>submit()</td>
<td>annoyAbout()</td>
</tr>
<tr>
<td></td>
<td>Reject()</td>
<td></td>
</tr>
<tr>
<td>Employee</td>
<td>read()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>adjust()</td>
<td></td>
</tr>
</tbody>
</table>
Access Matrix Implementations

♦ Global access table: Represents explicitly every cell in the matrix as a (actor, class, operation) 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.
  ♦ Requires a lot of space

♦ Access control list associates a list of (actor, operation) pairs with each class to be accessed.
  ♦ Every time an object is accessed, its private access list is checked for the corresponding actor and operation.
  ♦ Example: guest list for a party.

♦ A capability associates a (class, operation) pair with an actor.
  ♦ A capability provides an actor to gain control access to an object of the class described in the capability.
  ♦ Example: An invitation card for a party
Authentication and Encryption

♦ Authentication
  ♦ Process of verifying the association between the identity of a user (or calling subsystem) and the system
    ♦ User name / password
    ♦ Smart Cards
    ♦ Biometric Sensors (analyzing patterns of blood vessels in eyes / fingers

♦ Encryption
  ♦ Translating a message (plaintext) into an encrypted message (ciphertext).
  ♦ A key is used to encrypt (sender) and decrypt (receiver) a message

♦ Both approaches are fundamentally difficult problems
  ♦ → Select Off-the-Shelf packages !
Decide on Software Control

♦ Control Flow is the sequencing of actions in a system
  ◆ Object-oriented Systems: Deciding which operations should be executed in which order

♦ Control flow is realized by control objects. Activities:
  ◆ Record external events
  ◆ Store temporary states about events
  ◆ Issue the right sequence of method calls on both boundary and entity objects

♦ Examples for Control Models:
  ◆ Event-Driven Models (Model-View-Controller)
  ◆ Thread-based Models
Decide on Software Control

- Event-driven control
  - Control object runs a loop that for an external event
    - Example (Data) Model has changed
  - Whenever an event becomes available, it is dispatched to appropriate receiver components
  - Very flexible, good for the design of graphical user interfaces, easy to extend

![Diagram showing event-driven control with objects: Control, Program, Listener, Listener. Events: Something has changed, Update.]}
Decide on Software Control

- Thread-based (or decentralized) control
  - Control resides in several independent objects:
    - Each external event is assigned a single Thread
    - Each new Thread handles an event
  - Possible speedup by mapping the objects on different processors, increased communication overhead.
7. Boundary Conditions (1-2)

♦ Most of the system design effort is concerned with steady-state behavior.

♦ However, the system design phase must also address the initiation and finalization of the system. This is addressed by a set of new uses cases called *administration use cases*

- **Initialization**
  - Describes how the system is brought from an non initialized state to steady-state ("startup use cases").
    - How does the system start up?
    - What data need to be accessed at startup time?
    - What services have to registered?
    - What does the user interface do at start up time?
    - How does it present itself to the user?
7. Boundary Conditions (2-2)

- **Termination**
  - Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").
    - Are single subsystems allowed to terminate?
    - Are other subsystems notified if a single subsystem terminates?
    - How are local updates communicated to the database?

- **Failure**
  - Many possible causes: Bugs, errors, external problems (power supply).
  - Good system design foresees fatal failures ("failure use cases").
    - How does the system behave when a node or communication link fails? Are there backup communication links?
    - How does the system recover from failure? Is this different from initialization?
Defining Design Goals – Iterative Process

Define design goals

Define subsystems

Map subsystem to hardware/software platform

Manage persistent data

Define access control policies

Select a global control flow

Describe boundary conditions

Implement subsystems
Review

- Correct
- Complete
- Consistent
- Realistic

With respect to Use Cases and Analysis Model

- Readable
  - Can other developers understand system design
  - Meaningful names
  - Good description of functionality
Summary

♦ Hardware/Software mapping
♦ Persistent data management
♦ Global resource handling
♦ Software control selection
♦ Boundary conditions

♦ Each of these activities revises the subsystem decomposition to address a specific issue. Once these activities are completed, the interface of the subsystems can be defined.