Chapter 7, System Design - Software Architectures

Object-Oriented Software Construction

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

...and their models

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)
The Purpose of System Design

- Bridging the gap between desired and an executable system in a manageable way
- Input: Analysis Models
- Output: System Model (or Software Architecture)
- Use Divide and Conquer
  - We model the new system to be developed as a set of subsystems
Overview - System Design

1. System Decomposition
   - Software Architecture
     - Layers/Partitions
     - Cohesion /Coupling

2. Design Goals
   - Definition
   - Trade-Offs

3. Concurrency
   - Identification of Threads

4. Hardware/Software Mapping
   - Special Purpose
   - Buy or Build
   - Trade-Off
   - Allocation
   - Connectivity

5. Data Management
   - Persistent Objects
   - Files
   - Databases
   - Data Structures

6. Global Resource Handling
   - Access Control
   - Security

7. Software Control
   - Monolithic
   - Event-Driven
   - Threads
   - Conc. Processes

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure

System Design
Overview
Four Sessions on System Design

♦ System Design I – Software Architecture
  ♦ System Design Concepts
  ♦ Subsystem Decomposition

♦ System Design II – Architecture Organization
  ♦ Hardware/Software Mapping
  ♦ Persistent Data Management
  ♦ Global Resource Handling and Access Control
  ♦ Software Control
  ♦ Boundary Conditions

♦ System Design III - Addressing Design Goals
  ♦ Tactics for Performance, Security, Availability, Modifiability

♦ System Design IV – Selected Topics: SOA and MDA
How to use the results from the Requirements Analysis for System Design

♦ Functional model ➔
  ♦ Activity 1: System decomposition (Selection of subsystems based on functional requirements)

♦ Nonfunctional requirements ➔
  ♦ Activity 2: Design Goals Definition

♦ Object model ➔
  ♦ Activity 4: Hardware/software mapping
  ♦ Activity 5: Persistent data management

♦ Dynamic model ➔
  ♦ Activity 3: Concurrency
  ♦ Activity 6: Global resource handling
  ♦ Activity 7: Software control

♦ Subsystem Decomposition
  ♦ Activity 8: Boundary conditions
The Output: Software Architecture

♦ The software architecture of a program or computing system is the structure or structures of the system, which comprise
  ♦ components (here: subsystems)
  ♦ the externally visible properties of those components (here: service interface)
  ♦ and the relationships among them (Bass et al., 1998)
♦ The architecture describes the structure of a single system.
♦ Guidelines for building software architectures:
  ♦ Software architectural styles (later on)
Subsystems

♦ Subsystem (UML: Package)
  ♦ Collection of classes, associations, operations, events and constraints that are interrelated

♦ (Subsystem) Service:
  ♦ A set of related operations that share a common purpose
  ♦ Services are defined in System Design
Services and Subsystem Interfaces

- Service is specified by Subsystem interface
  - Specifies interaction and information flow from/to subsystem boundaries
  - No information about information flow within the subsystem
  - Should be well-defined and small
  - Subsystem Interfaces are refined in Object Design
  - Often called API: Application programmer’s interface, but this term should be used during implementation, not during System Design

- In Programming Languages subsystems are often bundled
  - Java: Packages
  - C++: Namespaces
  - Modula: Module
  - …
Definition: Subsystem Interface Object

- A Subsystem Interface Object provides a service
  - This is the set of public methods provided by the subsystem
  - The Subsystem interface describes all the methods of the subsystem interface object
- Use a Facade pattern for the subsystem interface object
From Use Cases to Subsystems
Some heuristics

- No straightforward method available
- Start with UML use case and class diagram
  - representation of functional behavior (requirements)
- Assign objects identified in one use case into the same subsystem
- For those objects that exist in several use case: create a dedicated subsystem (e.g. data sharing)
- All objects in the same subsystem should be functionally related

- Apply same techniques as available during analysis
  - Abbott’s heuristics
- Each subsystem should realized by an individual team
Choosing Subsystems

♦ Primary Question
  ♦ What kind of service is provided by the subsystems (subsystem interface)?
  ♦ How do we implement the subsystem (→ Object Design)

♦ Secondary Question
  ♦ Who or what interacts with that service and in what order?
Choosing Subsystems

- Criteria for subsystem selection
  - Self-contained
    - Most of the interaction should be within subsystems, rather than across subsystem boundaries
  - Reusable
    - Usage of the same subsystem in a different context
  - Maintainable
    - Understanding the purpose of a subsystem even in the future
    - Make modifications easily
  - Correct
    - No side effects

Goal: Reduction of Complexity

Approach: Adopt metrics to measure the style of subsystem(s)
Decomposition Aspects
Coupling and Cohesion

♦ Cohesion measures the dependence among classes (within a subsystem)
  ♦ High cohesion: The classes in the subsystem perform similar tasks and are related to each other (via associations, use relations)
  ♦ Low cohesion: Lots of miscellaneous and helping classes, no relationships identifiable

♦ Coupling measures dependencies between subsystems
  ♦ High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
  ♦ Low coupling: A change in one subsystem does not affect any other subsystem

♦ Subsystems should have as maximum cohesion and minimum coupling as possible:
  ♦ How can we achieve high cohesion?
  ♦ How can we achieve low coupling?
Decomposition Aspects A: Reducing Coupling

♦ Systems n are all dependent on System A

♦ If System A’s interface will change other subsystems have to be rearranged too

♦ Adapter system reduces coupling
**Decomposition Aspects:**

**Low Cohesion (Coincidental Cohesion)**

- A subsystem has coincidental cohesion if it performs multiple, completely unrelated actions.

- Example: Subsystem for “dealing with clients”
  - `print_next_line();`
  - `addClient();`
  - `rollbackTransaction();`
  - `openOnlineHelp();`

It’s BAD BECAUSE:

- No clear purpose
- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
  - Break the module into separate modules, each performing one task
Decomposition Aspects: High Cohesion (Informational Cohesion)

- A subsystem has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure.
- Example: Subsystem for creating a client record
  - createClient()
  - deleteClient()
  - addDateOfBirth()
  - addAddress()

It’s GOOD BECAUSE:
- Exact purpose
- It increases maintainability
- High reusability
Decomposition Aspects
Trade-Off

- Cohesion and Coupling Measures are dependent
- Increase cohesion by decomposing the system into smaller subsystems
  - Consequence: increase of coupling

Recommendations
- Re-Think the system architecture
- Right amount of interacting subsystems has to be found
  - context-dependent
  - creative process
Partitioning and layering are techniques to achieve low coupling.

Partitions vertically divide a system into several independent (or weakly-coupled) subsystems that provide services on the same level of abstraction:
- Subsystem depend loosely on each other, mostly operate in isolation.

A layer is a subsystem that provides subsystem services to a higher layers (level of abstraction):
- A layer can only depend on lower layers.
- A layer has no knowledge of higher layers.

→ A large system is usually decomposed into subsystems.
Subsystem Decomposition Heuristics:
- No more than 7+/−2 subsystems
  - More subsystems increase cohesion but also complexity (more services)
- No more than 4+/−2 layers, use 3 layers (good)
Decomposition Aspects
Relationships between Subsystems

♦ Layer relationship
  ♦ Layer A “Calls” Layer B (runtime)
  ♦ Layer A “Depends on” Layer B (“make” dependency, compile time)

♦ Partition relationship
  ♦ The subsystems have mutual but not deep knowledge about each other
  ♦ Partition A “Calls” partition B and partition B “Calls” partition A
Decomposition Aspects
Virtual Machine

♦ Virtual Machines: Dijkstra - T.H.E. operating system (1965):
  ♦ A system should be developed by an ordered set of virtual machines, each built in terms of the ones below it

♦ A virtual machine is an abstraction
  ♦ It provides a set of attributes and operations.

♦ A virtual machine is a (sub)system
  ♦ uses services provided by lower level virtual machines and provides a service to higher level machines

♦ Virtual machines can implement two types of software architecture
  ♦ Open and closed architectures
Decomposition Aspects
Closed Architecture (Opaque Layering)

- Any layer can only invoke operations from the immediate layer below
- Design goal: High maintainability, flexibility
Decomposition Aspects
Open Architecture (Transparent Layering)

- Any layer can invoke operations from any layers below
- Design goal: Runtime efficiency

Diagram:

```
  C1
  atr
  op

  C1
  atr
  op

  C1
  atr
  op

  C1
  atr
  op
```

```
  VM1
  C1
  atr
  op

  VM2
  C1
  atr
  op

  VM3
  C1
  atr
  op

  VM4
  C1
  atr
  op
```
ARENA Example

User Interface

Manages tournaments, applications, promotions.

Administers user accounts

Tournament

Stores user profiles (contact & subscriptions)

User Directory

User Management

Advertisement

Component Management

Session Management

Tournament Statistics

Stores user profiles

User Interface

Administers user accounts
ARENA Example
A 3-layered architecture

- User Interface
- Advertisement
- Tournament
- Component Management
- User Management
- Session Management
- Tournament Statistics
- User Directory
Example of a Closed Layering: ISO OSI 7-Layer Model

- ISO’s OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- Reference model defines 7 layers of network protocols and strict methods of communication between the layers.
- Closed software architecture
Example of a Open Layering: OSI model Packages and their Responsibility

Application layer:
- is the system you are designing (unless you build a protocol stack).
- The application layer is often layered itself

Presentation layer:
- performs data transformation services, such as byte swapping and encryption

Session layer:
- is responsible for initializing a connection, including authentication.

Transport layer:
- is responsible for reliably transmitting from end to end
  - TCP/IP sockets

Network layer:
- is responsible for that the data are reliably transmitted and routed within a network

Datalink layer:
- allows to send and receive frames without error using the services from the Physical layer

Physical layer:
- represents the hardware interface to the net-work.
- It allows to send() and receive bits over a channel.
Example of a Open Layering: Middleware allows Focus On The Application Layer

- Application
- Presentation
- Session
- Transport
- Network
- DataLink
- Physical

If using a middleware, this is where you operate:

- Corba / RMI
- TCP/IP
- Ethernet
- Wire
- Data in Socket
- Signal in Cable
- Object
Software Architectural Styles

♦ More sophisticated decomposition rules are required
♦ Need for recurring patterns → architectural styles
♦ Architectural Style:
  ✔ A description of element and relation types together with a set of constraints on how they are used (Shaw and Garlan, 1996).
  ✔ Describes a family (or set) of software architectures
  ✔ Concrete Architecture = instance of a style
♦ An architectural style also includes
  ✔ Control issues
  ✔ Data Issues
  ✔ Constraints
♦ Prominent software architectural styles:
  ✔ Client/Server, Peer-To-Peer, Repository, Model/View/Controller
  ✔ Three Tier, N-Tier, Service-oriented (SOA)
Software Architectural Styles: Client/Server Pattern 1

♦ One or many servers provides services to instances of subsystems, called clients.

♦ Client calls on the server, which performs some service and returns the result
  ♦ Client knows the interface of the server (its service)
  ♦ Server does not need to know the interface of the client

♦ Response in general immediately

♦ Users interact only with the client
Software Architectural Styles: Client/Server Pattern 2

- Often used in database systems
  - Front end: User application (client)
  - Back end: Database access and manipulation (server)

**Functions performed by client**
- Customized user interface
- Front end processing of data
- Initiation of server remote procedure calls
- Access to database server across the network

**Functions performed by the database server**
- Centralized data management
- Data integrity and database consistency
- Database security
- Concurrent operations (multiple user access)
- Centralized processing (for example archiving)

---

Armin B. Cremers, Sascha Alda & Tobias Rho (based on Bruegge & Dutoit)
Software Architectural Styles: 
Design Goals for Client/Server Systems

♦ Service Portability
   ♦ Server can be installed on a variety of machines and operating systems and functions in a variety of networking environments

♦ Transparency, Location-Transparency
   ♦ The server might itself be distributed, but should provide a single "logical" service to the user

♦ Performance
   ♦ Client should be customized for interactive display-intensive tasks
   ♦ Server should provide CPU-intensive operations

♦ Scalability
   ♦ Server should have spare capacity to handle larger number of clients

♦ Flexibility
   ♦ The system should be usable for a variety of user interfaces and end devices (e.g., WAP Handy, wearable computer, desktop)

♦ Reliability
   ♦ System should survive node or communication link problems
Software Architectural Styles: Peer-to-Peer Architectural Style

- Generalization of Client/Server Architecture
- Peer acts as a client and a server at the same time
- More difficult to implement due to possible constraints:
  - Temporary availability
  - End-user interaction
- Further problems:
  - Maintainability

Diagram:

```
Peer
  service1()
  service2()
  ...
  serviceN()
```

```
application1:DBUser
```

```
application2:DBUser
```

```
database:DBMS
```

1. updateData
2. changeNotification
Software Architectural Styles: Model/View/Controller

- Used to model interactive systems
- Subsystems are classified into 3 different types
  - Model subsystem
    - Responsible for maintaining a central data structure (knowledge)
  - View subsystem
    - Responsible for displaying data of the model to the user
  - Controller subsystem
    - Responsible for sequence of interactions with the user
- Model subsystem notifies views whenever changes have occurred
Model View Controller

- Multiple views can exist at the same time
- New views can be added without any changes to the model
Software Architectural Styles:
Sequence of Events (Communication)

1. Views subscribe to event
2. User types new data
3. Request name change in model
4. Notify subscribers
Software Architectural Style: Repository (Blackboard Architecture)

- Subsystems access and modify data from a single data structure
- Subsystems are loosely coupled (interact only through the repository)
- Control flow is dictated by central repository (triggers) or by the subsystems (locks, synchronization primitives)
- Difference to client-server system?

Diagram:

```
Subsystem  Repository
createData() setData() searchData()
getData()
```

Armin B. Cremers, Sascha Alda & Tobias Rho (based on Bruegge & Dutoit) Object-Oriented Software Construction 38
**Software Architectural Style: Pipe and Filter**

- Subsystems process data received from a set of inputs and send results to other, subsequent subsystems via a set of outputs
- Subsystems: filters
- Associations between filters: pipe
- Application: Transformations of data streams
- Example: Unix shell

![Diagram of Pipe and Filter](image)
Software Architectural Style: SOA

- Default Standard: Web Services (SOAP, UDDI, WSDL)
- Service Composition (BPEL, WSCI)
Software Architectural Style: Three Tier

- Typically used for Enterprise Applications (booking systems, portals)
- Essentially a three layered architecture (layer = tier)
- Very substantial
- J2EE 1.4 specification: approx. 1400 pages (!)
Summary

♦ System Design
  ◆ Reduces the gap between requirements and the (virtual) machine
  ◆ Decomposes the overall system into manageable parts

♦ Subsystem Decomposition
  ◆ Results into a set of loosely dependent parts which make up the system

♦ Next:
  ◆ Architecture Organization