Software Engineering I: Software Technology

WS 2008/09

System Design and Software Architecture

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen
Where are we?

• We have covered Ch 1 - 4
• We are moving to Chapter 5 and 6.
Munich Airport

- Sign up for the tour of the Munich Airport
- 3 possible slots, each of them between 10:30 and 12:30 o'clock.

  1. December 08
  11 December 08
  12 December 08
Why is Design so Difficult?

• **Analysis:** Focuses on the application domain
  • Relatively stable

• **Design:** Focuses on the solution domain
  1. The solution domain is changing very rapidly
     - Halftime knowledge in software engineering: About 3-5 years
  2. Cost of hardware rapidly sinking
  3. Design knowledge is a moving target
  4. Design must be done in a specific time

• **Design window:** Time in which design decisions have to be made.
The Scope of System Design

Bridge the gap
  • between a new system and an existing system in a manageable way

How?
  • Use Divide & Conquer:
    1) Identify design goals
    2) Model the new system as a set of subsystems
    3-8) Address the major design goals.
System Design: Eight Issues

1. Identify Design Goals
   - Identify Additional Nonfunctional Requirements
   - Discuss Trade-offs

2. Subsystem Decomposition
   - Layers vs Partitions
   - Coherence & Coupling

3. Identify Concurrency
   - Identification of Parallelism (Processes, Threads)

4. Hardware/Software Mapping
   - Identification of Nodes
   - Special Purpose Systems
   - Buy vs Build Decisions
   - Network Connectivity

5. Persistent Data Management
   - Storing Entity Objects
   - Filesystem vs Database

   - Access Control
   - ACL vs Capabilities
   - Security

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

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure.
Overview

System Design I (This Lecture)
  0. Overview of System Design
  1. Design Goals
  2. Subsystem Decomposition, Software Architecture

System Design II
  3. Concurrency: Identification of parallelism
  4. Hardware/Software Mapping:
     Mapping subsystems to processors
  5. Persistent Data Management: Storage for entity objects
  6. Global Resource Handling & Access Control:
     Who can access what?)
  7. Software Control: Who is in control?
  8. Boundary Conditions: Administrative use cases.
Analysis Sources: Requirements and System Model

Nonfunctional Requirements

Functional Model

Dynamic Model

Object Model

Functional Model

Dynamic Model

© 2008 Bernd Bruegge
Software Engineering WS 2008
How the Analysis Models influence System Design

• Nonfunctional Requirements
  => Definition of Design Goals

• Functional model
  => Subsystem Decomposition

• Object model
  => Hardware/Software Mapping, Persistent Data Management

• Dynamic model
  => Identification of Concurrency, Global Resource Handling, Software Control

• Finally: Hardware/Software Mapping
  => Boundary conditions
From Analysis to System Design

Nonfunctional Requirements
1. Design Goals
   Definition
   Trade-offs

Functional Model
2. System Decomposition
   Layers vs Partitions
   Coherence/Coupling

Dynamic Model
3. Concurrency
   Identification of Threads

Object Model
4. Hardware/Software Mapping
   Special Purpose Systems
   Buy vs Build
   Allocation of Resources
   Connectivity

5. Data Management
   Persistent Objects
   Filesystem vs Database

Functional Model
8. Boundary Conditions
   Initialization
   Termination
   Failure

Dynamic Model
7. Software Control
   Monolithic
   Event-Driven
   Conc. Processes

6. Global Resource Handling
   Access Control List
   vs Capabilities
   Security
Subsystem Decomposition

• **Subsystem**
  - Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
  - The objects and classes from the object model are the “seeds” for the subsystems
  - Subsystems are modeled in UML as components

• **Service**
  - A set of named operations that share a common purpose
  - The origin (“seed”) for services are the use cases from the functional model

• **Services are defined during system design.**
Subsystem Interfaces vs API

- **Subsystem interface:** Set of fully typed UML operations
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - Refinement of service, should be well-defined and small
  - *Subsystem interfaces are defined during object design*

- **Application programmer’s interface (API)**
  - The API is the specification of the subsystem interface in a specific programming language
  - APIs are defined during implementation

- The terms subsystem interface and API are often confused with each other
  - *The term API should not be used during system design and object design, but only during implementation.*
Subsystem Interface Object

- Good design: The subsystem interface object describes *all* the services of the subsystem interface

- **Subsystem Interface Object**
  - The set of public operations provided by a subsystem

  Subsystem Interface Objects should be realized with the Façade pattern (=> lecture on design patterns).
Properties of Subsystems: Layers and Partitions

- A layer is a subsystem that provides a service to another subsystem with the following restrictions:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called partitions
  - Partitions provide services to other partitions on the same layer
  - Partitions are also called “weakly coupled” subsystems.
Relationships between Subsystems

- Two major types of Layer relationships
  - Layer A “depends on” Layer B (compile time dependency)
    - Example: Build dependencies (make, ant, maven)
  - Layer A “calls” Layer B (runtime dependency)
    - Example: A web browser calls a web server
- Can the client and server layers run on the same machine?
  - Yes, they are layers, not processor nodes
  - Mapping of layers to processors is decided during the Software/hardware mapping!
- Partition relationship
  - The subsystems have mutual knowledge about each other
    - A calls services in B; B calls services in A (Peer-to-Peer).
Example of a Subsystem Decomposition

Layer 1

A:Subsystem

Layer 2

B:Subsystem

Partition relationship

C:Subsystem

D:Subsystem

Layer Relationship „depends on“

E:Subsystem

F:Subsystem

Layer 3

Layer Relationship „calls“

G:Subsystem
Virtual Machine

- A **virtual machine** is a subsystem connected to higher and lower level virtual machines by "provides services for" associations.
- A virtual machine is an abstraction that provides a set of attributes and operations.
- The terms layer and virtual machine can be used interchangeably.
  - Also sometimes called “level of abstraction”.
Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.
Building Systems as a Set of Virtual Machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.

Existing System

Operating System, Libraries
Closed Architecture (Opaque Layering)

- Each virtual machine can only call operations from the layer below

**Design goals:**
Maintainability, flexibility.
Open Architecture (Transparent Layering)

• Each virtual machine can call operations from any layer below

Design goal:
Runtime efficiency
Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design, because it reduces complexity
  - low coupling
- They have also reduced testing times
- Closed architectures are more portable
- Open architectures are more efficient
- Layered systems often have a chicken-and-egg problem

How do you open the symbol table when you are debugging the File System?
Coupling and Coherence of Subsystems

• Goal: Reduce system complexity while allowing change

• **Coherence** measures dependency among classes
  • **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via many associations
  • **Low coherence:** Lots of miscellaneous and auxiliary classes, almost no associations

• **Coupling** measures dependency among subsystems
  • **High coupling:** Changes to one subsystem will have high impact on the other subsystem
  • **Low coupling:** A change in one subsystem does not affect any other subsystem.
Coupling and Coherence of Subsystems

• **Goal:** Reduce system complexity while allowing change

• **Coherence** measures dependency among classes
  - **High coherence:** The classes in the subsystem perform similar tasks and are related to each other via associations
  - **Low coherence:** Lots of miscellaneous and auxiliary classes, no associations

• **Coupling** measures dependency among subsystems
  - **High coupling:** Changes to one subsystem will have high impact on the other subsystem
  - **Low coupling:** A change in one subsystem does not affect any other subsystem
Good Design

• Goal: Reduce system complexity while allowing change

• **Coherence** measures dependency among classes
  
  High coherence: The classes in the subsystem perform similar tasks and are related to each other via associations
  
  Low coherence: Lots of miscellaneous and auxiliary classes, no associations

• **Coupling** measures dependency among subsystems
  
  High coupling: Changes to one subsystem will have high impact on the other subsystem
  
  Low coupling: A change in one subsystem does not affect any other subsystem
How to achieve high Coherence

• **High coherence** can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries

• Questions to ask:
  • Does one subsystem always call another one for a specific service?
    • Yes: Consider moving them together into the same subsystem.
  • Which of the subsystems call each other for services?
    • Can this be avoided by restructuring the subsystems or changing the subsystem interface?
  • Can the subsystems even be hierarchically ordered (in layers)?
How to achieve Low Coupling

- **Low coupling** can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**, Parnas)

- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

David Parnas, *1941,
Developed the concept of modularity in design.
Architectural Style & Software Architecture

• **Subsystem decomposition**: Identification of subsystems, services, and their relationship to each other.

• **Architectural Style**: A pattern for subsystem decomposition

• **Software Architecture**: Instance of an architectural style
Examples of Architectural Styles

• Client/Server
• Peer-To-Peer
• Repository
• Model/View/Controller
• Three-tier, Four-tier Architecture
• Service-Oriented Architecture (SOA)
• Pipes and Filters
Client/Server Architectural Style

- One or many servers provide services to instances of subsystems, called clients.
- Each client calls on the server, which performs some service and returns the result.
  The clients know the interface of the server.
  The server does not need to know the interface of the client.
- The response in general is immediate.
- End users interact only with the client.

```
  requester  provider
  +service1() +service2() +serviceN()

Client   *   Server
```
Client/Server Architectures

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

• Functions performed by client:
  • Input from the user (Customized user interface)
  • Front-end processing of input data

• Functions performed by the database server:
  • Centralized data management
  • Data integrity and database consistency
  • Database security
Problems with Client/Server Architectures

- Client/Server systems do not provide peer-to-peer communication
- Peer-to-peer communication is often needed
- Example:
  - Database must process queries from application and should be able to send notifications to the application when data have changed

```
application1:DBUser
1. updateData

application2:DBUser
2. changeNotification

database:DBMS
```
Peer-to-Peer Architectural Style

Generalization of Client/Server Architecture

Clients can be servers and servers can be clients

=> “A peer can be a client as well as a server”.

![Diagram showing peer-to-peer architectural style](image-url)
Example: Peer-to-Peer Architectural Style

- ISO’s OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection
- Reference model which defines 7 layers and communication protocols between the layers
OSI Model Layers and Services

- The **Application layer** is the system you are building (unless you build a protocol stack)
  - The application layer is usually layered itself
- The **Presentation layer** performs data transformation services, such as byte swapping and encryption
- The **Session layer** is responsible for initializing a connection, including authentication
OSI Model Layers and their Services

- **The Transport layer** is responsible for reliably transmitting messages
  - Used by Unix programmers who transmit messages over TCP/IP sockets

- **The Network layer** ensures transmission and routing
  - Services: Transmit and route data within the network

- **The Datalink layer** models frames
  - Services: Transmit frames without error

- **The Physical layer** represents the hardware interface to the network
  - Services: `sendBit()` and `receiveBit()`
The Application Layer Provides the Abstractions of the “New System”
An Object-Oriented View of the OSI Model

- The OSI Model is a closed software architecture (i.e., it uses opaque layering)
- Each layer can be modeled as a UML package containing a set of classes available for the layer above
Bidirectional associations for each layer
Middleware Allows Focus On Higher Layers

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

- Object
- Socket
- TCP/IP
- Ethernet
Repository Architectural Style

- Subsystems access and modify data from a single data structure called the repository
- Also called blackboard architecture
- Subsystems are loosely coupled (interact only through the repository)
- Control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives
Repository Architecture Example: Incremental Development Environment (IDE)
Model-View-Controller

- **Problem:** Assume a system with high coupling. Then changes to the boundary objects (user interface) often force changes to the entity objects (data)
  - The user interface cannot be reimplemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- **Solution:** The model-view-controller architectural style, which decouples data access (entity objects) and data presentation (boundary objects)
  - The Data Presentation subsystem is called the *View*
  - The Data Access subsystem is called the *Model*
    - So far this is the observer pattern!
  - The *Controller* is a new subsystem that mediates between View (data presentation) and Model (data access)
- Often called *MVC*. 
Model-View-Controller Architectural Style

- Subsystems are classified into 3 different types
  
  **Model subsystem:** Responsible for application domain knowledge

  **View subsystem:** Responsible for displaying application domain objects to the user

  **Controller subsystem:** Responsible for sequence of interactions with the user and notifying views of changes in the model

Better understanding with a Collaboration Diagram
Example: Modeling the Sequence of Events in MVC

4.0 User types new filename

5.0 Request name change in model

7.0 Show updated views

© 2008 Bernd Bruegge
Software Engineering WS 2008
3-Layer-Architectural Style

3-Tier Architecture

**Definition: 3-Layer Architectural Style**
- An architectural style, where an application consists of 3 hierarchically ordered subsystems
  - A user interface, middleware and a database system
  - The middleware subsystem services data requests between the user interface and the database subsystem

**Definition: 3-Tier Architecture**
- A software architecture where the 3 layers are allocated on 3 separate hardware nodes
- Note: *Layer* is a type (e.g. class, subsystem) and *Tier* is an instance (e.g. object, hardware node)
- Layer and Tier are often used interchangeably.
Virtual Machines in 3-Layer Architectural Style

A 3-Layer Architectural Style is a hierarchy of 3 virtual machines usually called presentation, application and data layer.

Presentation Layer (Client Layer)

Application Layer (Middleware, Business Logic)

Data Layer

Existing System
Example of a 3-Layer Architectural Style

• Three-Layer architectural style are often used for the development of Websites:
  1. The Web Browser implements the user interface
  2. The Web Server serves requests from the web browser
  3. The Database manages and provides access to the persistent data.
Example of a 4-Layer Architectural Style

4-Layer-architectural styles (4-Tier Architectures) are usually used for the development of electronic commerce sites. The layers are

1. The **Web Browser**, providing the user interface
2. A **Web Server**, serving static HTML requests
3. An **Application Server**, providing session management (for example the contents of an electronic shopping cart) and processing of dynamic HTML requests
4. A back end **Database**, that manages and provides access to the persistent data
   - In current 4-tier architectures, this is usually a relational Database management system (RDBMS).
MVC vs. 3-Tier Architectural Style

- The MVC architectural style is nonhierarchical (triangular):
  - View subsystem sends updates to the Controller subsystem
  - Controller subsystem updates the Model subsystem
  - View subsystem is updated directly from the Model subsystem

- The 3-tier architectural style is hierarchical (linear):
  - The presentation layer never communicates directly with the data layer (opaque architecture)
  - All communication must pass through the middleware layer

- History:
  - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc
  - 3-Tier (1990s): Originated with the appearance of Web applications, where the client, middleware and data layers ran on physically separate platforms
Additional Readings

• E.W. Dijkstra (1968)
  • The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457

• D. Parnas (1972)
  • On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058

• L.D. Erman, F. Hayes-Roth (1980)

• J.D. Day and H. Zimmermann (1983)
  • The OSI Reference Model, Proc. IEEE, Vol.71, 1334-1340

• Jostein Gaarder (1991)
  • Sophie’s World: A Novel about the History of Philosophy.
Summary

• System Design
  • An activity that reduces the gap between the problem and an existing (virtual) machine

• Design Goals Definition
  • Describes the important system qualities
  • Defines the values against which options are evaluated

• Subsystem Decomposition
  • Decomposes the overall system into manageable parts by using the principles of cohesion and coherence

• Architectural Style
  • A pattern of a typical subsystem decomposition

• Software architecture
  • An instance of an architectural style
  • Client Server, Peer-to-Peer, Model-View-Controller.