Software Engineering – Fall 2015
(CSC 4350/6350)
TR. 5:30 pm – 7:15 pm

Rao Casturi
10/27/2015
http://cs.gsu.edu/~ncasturi1
System Design Activities: Addressing Design Goals
Addressing Design Goals

1. Mapping Subsystems to Processors and Components
2. Identifying and Storing Persistent Data
3. Providing Access Control
4. Designing the Global Control Flow
5. Identifying Services
6. Identifying Boundary Conditions
7. Reviewing the System Design Model
Addressing Design Goals

Define design goals

Define subsystems

Implement subsystems

Map subsystems to hardware/software platform

Manage persistent data

Define access control policies

Select a global control flow

Describe boundary conditions

Source: OOSE-Bernd Bruegge & Allen H. Dutoit
1. Mapping Subsystem to Processes and Components

• **UML Deployment Diagrams**

• DDs are used to show the relationship among run-time **components** and **nodes**.

• **Components**: Self-contained entities that provide services to other components. (Eg: Web browsers, Web Servers etc)

• **Node**: Is a physical device or an execution environment in which components are executed.

• **System**: Composed of interacting run-time components that can be distributed among several nodes. A node can contain another node.
Example: Components and nodes
1. Mapping Subsystem to Processes and Components

- Many Systems run on more than one computer.
- High-Performance needs
- Multiple uses
- Selection of virtual machine
- Hardware mapping activity has significant impact on the performance and complexity of the system.
- Suggested approach is to perform it early in system design.
2. Persistent Data Store

- **Flat Files**: Storage abstractions provided by operating system. Data stored as a sequence of bytes. This is relatively low level and could enhance speed. However, data can be corrupted or/and lost easily. Should be used for very small applications that may not grow in size.

- **Relational Database**: Data stored in tables – each column is an attribute and each row represents a data item as a tuple of attribute values.

- **OO database**: Provide similar services to relational databases, but stores data as objects and associations. It also provide inheritance and abstract datatypes. However, it is slower than relational database.
When to use flat files, relational databases, and object-oriented databases

When should you choose flat files?
• Voluminous data (e.g., images)
• Temporary data (e.g., core file)
• Low information density (e.g., archival files, history logs)

When should you choose a relational or an object-oriented database?
• Concurrent accesses
• Access at finer levels of detail
• Multiple platforms or applications for the same data

When should you choose a relational database?
• Complex queries over attributes
• Large data set

When should you choose an object-oriented database?
• Extensive use of associations to retrieve data
• Medium-sized data set
• Irregular associations among objects
3. Designing the Global Control Flow

Control flow is the sequencing of actions in a system.
• Control flow is a design problem.

There are 3 possible control mechanisms:

• **Procedure-driven control**: Operations wait for input whenever data is needed. Used mainly in systems written using procedural languages.

• **Event-driven control**: A main loop, also waits for an external event. A main loop used. Useful for testing subsystems. Also the most commonly used.

• **Threads**: The system creates an arbitrary number of threads and run them concurrently. Each thread is run on the basis of procedure-driven.
1. **Procedure-driven control:**

Operations wait for input whenever data is needed. Used mainly in systems written using procedural-languages.

```java
Stream in, out;
String userid, passwd;
/* Initialization omitted */
out.println("Login:");
in.readLine(userid);
out.println("Password:");
in.readLine(passwd);
if (!security.check(userid, passwd)) {
    out.println("Login failed.");
    system.exit(-1);
}
/* ... */
```

Source: OOSE-Bernd Bruegge & Allen H. Dutoit
2. Event-driven control:

A main loop, also waits for an external event. A main loop used. Useful for testing subsystems. Also the most commonly used.

```java
Iterator subscribers, eventStream;
Subscriber subscriber;
Event event;
EventStream eventStream;
/*@ ... */
while (eventStream.hasNext()) {
    event = eventStream.next();
    subscribers = dispatchInfo.getSubscribers(event);
    while (subscribers.hasNext()) {
        subscriber = subscribers.next();
        subscriber.process(event);
    }
}/*@ ... */
```

Source: OOSE-Bernd Bruegge & Allen H. Dutoit
3. **Threads-driven control**: Concurrent variation of the procedure driven control flow

```java
Thread thread;
Event event;
EventHandler eventHandler;
boolean done;
/**< ...*/
while (!done) {
    event = eventStream.getNextEvent();
    eventHandler = new EventHandler(event)
    thread = new Thread(eventHandler);
    thread.start();
}
/**< ...*/
```

Source: OOSE-Bernd Bruegge & Allen H. Dutoit
Identifying Services & Boundary Conditions

• Configuration
  – Creation of the object and destroying of the object

• Start-up and Shutdown
  – For each component add 3 uses cases – Start, Shutdown and Configuration of the component

• Exception handling
  – Notify the user when there is any error
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 Overview

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

Glossary
System Design and Decomposition summary

- **Subsystem decomposition** describes the decomposition into subsystems and the responsibilities of each. This is the main product of system design.
- **Hardware/software mapping** describes how subsystems are assigned to hardware and off-the-shelf components. It also lists the issues introduced by multiple nodes and software reuse.
- **Persistent data management** describes the persistent data stored by the system and the data management infrastructure required for it. This section typically includes the description of data schemes, the selection of a database, and the description of the encapsulation of the database.
- **Access control and security** describes the user model of the system in terms of an access matrix. This section also describes security issues, such as the selection of an authentication mechanism, the use of encryption, and the management of keys.
- **Global software control** describes how the global software control is implemented. In particular, this section should describe how requests are initiated and how subsystems synchronize. This section should list and address synchronization and concurrency issues.
- **Boundary conditions** describes the start-up, shutdown, and error behavior of the system. (If new use cases are discovered for system administration, these should be included in the requirements analysis document, not in this section.)
Review System Design

• System Design is an Iterative process
• System Design model equal to Analysis Model

✓ Correctness
✓ Completeness
✓ Consistent
✓ Realistic
✓ Readable
Object Design
• Close the gap between the application objects and the off-the-shelf components.
• Identify additional solution objects.
• Refine the existing objects.
Object Design includes
(4 Groups of Activities)

1. Reuse (Component Selection)
2. Service Specification
3. Object Model Restructuring
4. Object Model Optimization
Reuse or off-the-shelf components

• Off-the-Shelf components are identified during system design.
• Class Libraries and additional components are selected for basic data structures and services.
• Design patterns are selected for solving common problems.
• Classes are protected from future changes.
• Buy-versus-Build trade off
The subsystem services identified during the system design are specified in terms of:

[Class Interfaces, Operations, arguments, type signatures and exceptions].

Additional operations and objects needed for data transfer among subsystems are identified.

The subsystem service specification is also called API (Application Programmer Interface).
Object Model Restructuring

• In this phase the focus is more on the reuse of the code.
• Revisit the N-ary Association of classes and see if we can get it to Binary association
• Look for Merging 2 similar classes, 2 similar subsystems.
• Attribute merging.
• Complex class decomposition for simple inheritance.
• Maintainability, Readability, Understandability
Object Model Optimization

• This phase addresses the performance requirements of the system model.
• Efficiency of algorithms, Memory management is a focus.
• Techniques like simplicity vs multiplicity
• Rearrange Execution order for efficiency
• Adding redundancy / derived attributes for speedup.
Christopher Alexander says, “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it same way ever twice”.

*Objects and interfaces instead of Walls and doors*
Reuse Concepts:
Solution Objects, Inheritance, and Design Patterns

• Application Objects and Solution Objects
• Specification Inheritance and Implementation Inheritance
• Delegation
• The Liskov Substitution Principle
• Delegation and Inheritance in Design Patterns
Objects

• During analysis, we identify entity objects and their relationships, attributes, and operations. Most entity objects are application objects that are independent of any specific system.

• During analysis, we also identify solution objects that are visible to the user, such as boundary and control objects representing forms and transactions defined by the system.

• During system design, we identify more solution objects in terms of software and hardware platforms.

• During object design, we refine and detail both application and solution objects and identify additional solution objects needed to bridge the object design gap.
Inheritance

• Superclass, Subclass
• The focus of inheritance during object design is to reduce redundancy and enhance extensibility.
• By factoring all redundant behavior into a single superclass, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we have to make changes only once for all subclasses
• By Providing abstract classes interfaces that are used by the application, we can write new specialized behavior by writing new subclasses that comply with the abstract interfaces.
Delegation

- **Delegation** is the alternative to implementation inheritance that should be used when reuse is desired.
- A class is said to delegate to another class if it implements an operation by resending a message to another class.
- Delegation makes explicit the dependencies between the reused class and the new class.
The Liskov Substitution Principle

• If a client code uses the methods provided by a superclass, then developers should be able to add new subclasses without having to change the client code.
The Liskov Substitution Principle

- **Liskov Substitution Principle**
- If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

**Interpretation**

In object design, the Liskov Substitution Principle means that if all classes are subtypes of their superclasses, all inheritance relationships are specification inheritance relationships. In other words, a method written in terms of a superclass T must be able to use instances of any subclass of T without knowing whether the instances are of a subclass. Consequently, new subclasses of T can be added without modifying the methods of T, hence leading to an extensible system. An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.
Design pattern – 4 Essential Elements

1. A **name** that uniquely identifies the pattern from other patterns.
2. A **problem description** that describes the situations in which the pattern can be used. Problems addressed by design patterns are usually the realization of modifiability and extensibility design goals and nonfunctional requirements.
3. A **solution** stated as a set of collaborating classes and interfaces.
4. A set of **consequences** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.
<table>
<thead>
<tr>
<th>Scope</th>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter</td>
<td>Adapter (Class)</td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Class)</td>
<td></td>
<td>Template Method</td>
</tr>
<tr>
<td>Object</td>
<td>Object</td>
<td>Abstract</td>
<td>Adapter</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>Factory</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Builder</td>
<td></td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td>Prototype</td>
<td>Composition</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Façade</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td></td>
<td>Flyweight</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td>Class</td>
<td></td>
<td>Proxy</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>
A.1 Abstract Factory: Encapsulating Platforms

**Name**
Abstract Factory Design Pattern

**Problem description**
Shield the client from different platforms that provide different implementations for the same set of concepts.

**Solution**
A platform (e.g., a windowing system) is represented as a set of AbstractProducts, each representing a concept (e.g., a button) that is supported by all platforms. An AbstractFactory class declares the operations for creating each individual product. A specific platform is then realized by a ConcreteFactory and a set of ConcreteProducts (one for each AbstractProduct). A ConcreteFactory depends only on its related ConcreteProducts. The Client depends only on the AbstractProducts and the AbstractFactory classes, making it easy to substitute platforms.

**Consequences**
- Client is shielded from concrete product classes.
- Substituting families at runtime is possible.
- Adding new products is difficult since new realizations for each factory must be created.

**Examples**
- Statically encapsulating incompatible infrastructures for intelligent houses (Section 8.4.4)
- Game independence in ARMA (Section 8.6.1)

**Related concept**
Specification inheritance and implementation inheritance (Section 8.5.2).

*Figure A.1* The Abstract Factory design pattern (adapted from [Gamma et al., 1995]).
Questions?
References

• Use Cases Combined with BOOCH, OMT UML Process and Products
  - Putnam P Texel, Charles B Williams

• Object-Oriented Software Engineering Using UML Patterns, and JAVA
  - Bernd Bruegge & Allen H. Dutoit

• Software Engineering 9th Edition
  - Ian Sommerville