Object-Oriented Architecture & Design – Lecture 1 (of 2)
Problem Decomposition

Ivo Krka, Daniel Link

University of Southern California
Computer Science Department
Software Architecture Group
Story Arc

• Storyline in episodic form
• Goal for OOA&D: **Move from requirements to design, reified by a model**
• OOA&D will be split across 2 lectures
  – Lecture 1: Problem Decomposition (today)
  – Lecture 2: UML in Depth and Advanced Topics (10/5)
• OOA&D III & Workshop (10/24)
  – Interactive session to review your designs
Outline for Today

• The Role of Software Architecture
• Object Oriented Design Basics
• Design in Context
• Internet Bookstore Example
  – Domain Modeling
  – Use Case Modeling
Software Architecture

• Principal design decisions
• Architecture is manifested as a system’s:
  – Components
  – Connectors
  – Topology/Configuration
• Architecture != Design (global vs. local)
• Architecture is both the process of arriving at these principal decisions and the creation of artifacts reifying these decisions
Why deal with Architecture?

Why not get started without one?
Why deal with Architecture?

- General aspects
  - All software systems have an architecture
    - Big ball of mud counts
  - All software systems have an architect
- Risk reduction benefits
  - Higher level of reasoning necessary to deal with complex systems (such as 577 ones)
  - Artifacts of design decisions persist (and are useful) throughout the software lifecycle
    - You might not implement the software in this class (but rather someone else may later)
    - You will need to communicate design to stakeholders
    - Someone may still need to maintain software or add new features in 10 years
Conceptual Design Tools

• Abstraction and Terminology
  – What are the fundamental concepts in your system?

• Separation of Concerns
  – Isolate likely change
  – Identify commonalities

• Refined Experience
  – What have other architects found useful?
Examples of Past Experience

- Domain-Specific Software Architectures
- Architectural Patterns
- Styles
- (Program) Design Patterns

Scope

Application Structure

System Structure

Programming (language level)

Application Domain Knowledge

Shallow

Deep
Styles: Client-Server

- Description: Clients send requests to servers, which perform the required function and reply with requested information
- Clients initiate interaction
- Basic Example: Web Browser
Styles: Publish-Subscribe

• Description: Subscribers register/deregister for specific messages or content. Event servers maintain a list of subscribers. Content-based routing is possible.

• Basic Example: Multiplayer networked games, news, business/stock market transactions
Power of Architecture: REST

- The application is distributed (actually, decentralized) hypermedia
- Architecture of the Web is wholly separate from the code
- There is no single piece of code that implements the architecture.
- There are multiple pieces of code that implement the various components of the architecture.
  - E.g., different Web browsers
- Stylistic constraints of the Web’s architectural style are not apparent in the code
  - The effects of the constraints are evident in the Web
- One of the world’s most successful applications is only understood adequately from an architectural vantage point.

REST Principles

• [RP1] The key abstraction of information is a resource, named by a URL. Any information that can be named can be a resource.

• [RP2] The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components.

• [RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

REST Principles (cont’d)

- [RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically.

REST Principles (cont’d)

• [RP5] Idempotent operations and representation metadata are encouraged in support of caching and representation reuse.

• [RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.

When To Break The Rules

- Remember - patterns and styles are abstractions
  - Design guides from past experience
- OK to deviate, but you should do so for a reason
- Example: An existing layered architecture must add role-based security.
  - LDAP is chosen as both the authentication and authorization tool (single, gold standard).
  - All front-ends must support user authentication
  - Before any action is performed, the system must validate that the user is authorized to perform the action.
Object Oriented Design

- OO in the `60s & `70s – Informal notions
  - Simula, Smalltalk
- In `82, Grady Booch coined the term Object Oriented Design
  - The idea was to combine a design methodology with language constructs that implement design concepts
  - Objects in design space are classes in implementation space
OOP

• Major features include
  – Encapsulation
    • Access is restricted (separation of concerns, no spaghetti code)
  – Message passing
    • Well-defined interfaces to the outside
  – Inheritance
    • Reuse, subtypes
What Are Objects?

• Three key attributes:
  – State
    • The collection of information known to an object
    • State changes are a consequence of operations performed on an object
  – Operations
    • The actions that can be performed on an object
  – Identity
    • Two objects with identical state are still different from one another

Source: *Mastering Object-Oriented Design in C++* by Cay S. Horstmann
Design Goals for OOD

• Booch defined three goals
  – Identify the objects/classes
  – Identify the functionality of these objects/classes
  – Identify the relationship between these objects/classes

• This is an iterative process
  – Decisions in design space are complex
  – Identification and specification of one aspect of a class might force changes in other aspects

Source: Mastering Object-Oriented Design in C++ by Cay S. Horstmann
Simple Methodology

• Objects are **nouns** in a problem statement or requirement
• Operations are **verbs** in the problem statement or requirement
  – A **user** shall be able to **delete** a **message** from a **mailbox**
• Relationships are found by looking for use, aggregation, and inheritance
  – “is-a” & “has-a” statements
  – **Basic Use**: Does an operation of Object A modify, require, or produce an Object B?
  – **Aggregation**: Does Object A have a reference to Object B?
  – **Inheritance**: Does Object A contain all aspects of Object B? Are all operations on Object B therefore allowed for Object A?

Source: *Mastering Object-Oriented Design in C++* by Cay S. Horstmann
Beyond the Simple Methodology

• Good for very constrained problems, but not larger, more complex software systems.
• What about NFPs, levels of service, interaction with software frameworks and middleware?
• Largely ties back to reasoning about the Software Architecture.
• What about past experience?
Design Patterns: Factory Method

- The factory method allows the developer to specify an interface, but to defer the actual instantiation to subclasses
- Often used in Framework development
Design Patterns: Facade Method

- Simplify multiple disjoint interfaces into a single class
- More readable and understandable
- Classic use of abstraction to interface with multiple libraries, objects, etc.

```java
class Computer {
    private CPU cpu=null;
    private Memory memory=null;
    private HardDrive hardDrive=null;

    public Computer() {
        this.cpu=new CPU();
        this.memory=new Memory();
        this.hardDrive=new HardDrive();
    }

    public void startComputer() {
        cpu.freeze();
        memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
    }
}
```
Design in Context

- Design – Requirements Interplay
  - Design forms the vocabulary for requirements
  - Requirements constrain design

- Existing patterns, styles and frameworks can often be leveraged
  - Seldom do we as software engineers get to design from a clean slate (Greenfield)
  - Frameworks & Libraries are a codification of common design idioms in a particular domain
  - COTS often an option for implementing a subsystem/portion of functionality
  - Brownfield design is also common
    • Your system will need to be integrated into an existing environment
COTS Selection

- **COTS Characteristics**
  - Concurrency, Dynamism, Encapsulation, …

- **COTS Interfaces**
  - Inputs, Outputs, Protocols, …

- **COTS Dependencies**
  - Software(s) required to run the OTS product
COTS Integration Analysis Goals

• Identify and mitigate integration risks during design
• Reduce rework occurring from integration incompatibilities
• Select a combination of COTS to minimize integration effort and schedule
• Detect architecture mismatches occurring due to assumptions made by COTS components
• Rapidly filter high-risk COTS combinations
Integration Matrices

• An integration matrix is a simple and straightforward knowledge capture method
• Alternative or recurring design decisions across the columns
• Desired system properties along the rows
• Cells capture compatibilities, conflicts, or other relationships
<table>
<thead>
<tr>
<th>Integration styles vs. Properties</th>
<th>Topology</th>
<th>Linkage</th>
<th>Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>Point-to-Point</td>
<td>Hub and Spoke</td>
<td>Adapter</td>
</tr>
<tr>
<td>Local</td>
<td>Peer-to-Peer</td>
<td>Shared Bus</td>
<td>Translator</td>
</tr>
<tr>
<td>Secure</td>
<td>Peer-to-Peer</td>
<td>Shared Data</td>
<td>Arbitrator</td>
</tr>
<tr>
<td>Data intensive</td>
<td>Peer-to-Peer</td>
<td>Messaging</td>
<td>Distributor</td>
</tr>
<tr>
<td>Data formats incompatible</td>
<td>Peer-to-Peer</td>
<td>Explicit invocation</td>
<td></td>
</tr>
<tr>
<td>Data consistency</td>
<td>Peer-to-Peer</td>
<td>Data Streaming</td>
<td></td>
</tr>
<tr>
<td>Interaction protocols incompatible</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliable</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real time</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One-to-many</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many-to-one</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Always available</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Periodically scheduled</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loose coupling</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamically reconfigurable</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalable</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Caching</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed transactions</td>
<td>Peer-to-Peer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integrations styles vs. Properties</td>
<td>Topology</td>
<td>Linkage</td>
<td>Connector</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------</td>
<td>-------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td></td>
<td>Point-to-Point</td>
<td>Hub and Spoke</td>
<td>Shared Bus</td>
</tr>
<tr>
<td>Distributed</td>
<td>0</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Local</td>
<td>0</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Secure</td>
<td>+</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Data intensive</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Data formats incompatible</td>
<td>0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Data consistency</td>
<td>0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Interaction protocols incompatible</td>
<td>0</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Reliable</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Real time</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
</tr>
<tr>
<td>One-to-many</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Many-to-one</td>
<td>-</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Always available</td>
<td>+</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>Periodically scheduled</td>
<td>+</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Loose coupling</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Robustness</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Dynamically reconfigurable</td>
<td>-</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td>Scalable</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
Architectural Middleware: Prism-MW

- Architectural middleware for distributed, resource constrained, mobile, and embedded systems
- Supports architecture-based software development
  - Architecture-based software development is the implementation of a software system in terms of its architectural elements
- Efficient, scalable, flexible and extensible
  - Allows us to cope with heterogeneity
- Supports arbitrarily complex architectures
- Supports multiple architectural styles (event-based is the primary one)
class DemoArch {
    static public void main(String argv[]) {
        Architecture arch = new Architecture("DEMO");
        // create components
        ComponentA a = new ComponentA("A");
        ComponentB b = new ComponentB("B");
        ComponentD d = new ComponentD("D");
        // create connectors
        Connector conn = new Connector("C");
        // add components and connectors
        arch.addComponent(a);
        arch.addComponent(b);
        arch.addComponent(d);
        arch.addConnector(conn);
        // establish the interconnections
        arch.weld(a, conn);
        arch.weld(b, conn);
        arch.weld(conn, d)
    }
}
Component D sends an event

Event e = new Event("Event_D");
e.addParameter("param_1", p1);
send (e);

Component B handles the event and sends a response

public void handle(Event e)
{
    if (e.equals("Event_D")) {
        ...
        Event e1= new Event("Response_to_D");
e1.addParameter("response", resp);
send(e1);
    }
    ...
}
UML

- Unified Modeling Language
- Models the processes in project lifecycle
- Standardized Language developed by Grady Booch et al.

Diagrams Source: UML basics, Donald Bell
UML Introductions

• http://edn.embarcadero.com/article/31863
• http://agile.csc.ncsu.edu/SEMaterials/UMLOverview.pdf
UML Software

• Visual Paradigm for UML
• Linked from Tools page at http://greenbay.usc.edu/csci577/fall2013/tools
• Get license from Tools page
• Versions for all major operating systems
• Familiarize yourselves
In The Next Lectures…

• We will cover model representations (UML)
• We will discuss incorporation of COTS in design
• We will discuss how the choice of framework impacts architecture (and design)

• But with the rest of our time today…
  – An example
Example – Internet Bookstore

• Online retail outlet for purchasing books
• Provide basic user capabilities for:
  – Searching for books
  – Reviews & rating of books
  – Purchase of books
• Interfaces to shipping and inventory management
Internet Bookstore Requirements

(1/2)

• The bookstore shall accept orders over the internet
• The bookstore shall maintain a list of accounts for up to 1,000,000 customers
• The bookstore shall provide password protection for all accounts
• The bookstore shall provide the ability to search the master book catalog
• The bookstore shall provide a number of search methods on that catalog, including search by author, search by title, search by ISBN number, and search by keyword
• The bookstore shall provide a secure means of allowing customers to pay by credit card
Internet Bookstore Requirements

(2/2)

• The bookstore shall provide a secure means of allowing Customers to pay via purchase order
• The bookstore shall provide a special kind of account that is pre-authorized to pay via purchase order
• The bookstore shall provide electronic links between the Web and the book information database (BID) and the shipping fulfillment center
• The bookstore shall maintain reviews of books and should allow anyone to upload review comments
• The bookstore shall maintain ratings on books based on customer input
Domain Modeling

• Necessary to ground both requirements and design (and hopefully implementation) in terminology shared by all stakeholders

• A methodology for building terminology, as well as discovering and refining use-cases and requirements

• Many techniques exist
Domain Modeling Guidelines (1/2)

- Focus on real-world (problem domain) objects.
- Use generalization (is-a) and aggregation (has-a) relationships to show how the objects relate to each other.
- Limit your initial domain modeling efforts to a couple of hours (for simplicity).
- Organize your classes around key abstractions in the problem domain.
- Don’t mistake your domain model for your data model.

Source: Use case driven object modeling with UML: theory and practice by Doug Rosenberg and Matt Stephens
Domain Modeling Guidelines (2/2)

• Don’t confuse an object (a single instance) with a database table (a collection of things).
• Use the domain model as a project glossary.
• Do your initial domain model before you write your use cases, to avoid name ambiguity.
• Don’t expect your final class diagrams to precisely match your domain model, but there should be some resemblance between them.
• Don’t put screens and other GUI-specific classes on your domain model.

Source: Use case driven object modeling with UML: theory and practice by Doug Rosenberg and Matt Stephens
Sounds a lot like OO, right?
Example Technique: Color Modeling

• First used for Java software development in 1997
• Refined by Peter Coad, et. al., in Java Modeling in Color with UML
• Focuses on identifying different varieties of domain classes (or ‘archetypes’):
  – Party/Place/Thing
  – Role
  – Activity
  – Description (we aren’t going to cover this one)
• First step in the technique is to start with enumerating the activities in the problem domain that the software should support
  – Identify each Party, Place or Thing who participate in the activity
  – Identify what Roles the Party, Place or Things play while interacting with the activity
As an initial example let us focus on the activity of account registration.

Two requirements suggest that this is a domain activity:
- “The bookstore shall maintain a list of accounts for up to 1,000,000 customers”
- “The bookstore shall provide password protection for all accounts”

Since the requirements refer to accounts with password protection, you can assume that ‘People’ in the role of ‘Customers’ will have to register with the Internet Bookstore in order to create accounts for themselves.

Of course, like everything else during this early phase of the development process, this assumption must be verified with the relevant critical stakeholders.
In color modeling, each object has a stereotype (this is a UML term) in ‘<>’ and a color indicating its archetype:

- <activity> classes are colored pink
- <party>, <place>, and <thing> classes are colored green
- <role> classes are colored yellow

Why is there a privileged account vs. an ordinary account?

"The bookstore shall provide a special kind of account that is pre-authorized to pay via purchase order"
Use Case Modeling Basics

- The purpose of Use-Case Design is to discover the necessary:
  - *Interface Objects & their classes*
    - e.g. pages, dialog boxes, representations of entities
  - *Entity Objects & their classes*
    - Both temporary & persistent,
    - e.g., data structures, DB tables, etc
  - *Control Objects & their classes*
    - Activity or control flow classes
  - The *messages they exchange*
UML Use Case Symbols

• Interface Classes are labeled with: 🔄

• Entity Classes are labeled with: 🔄

• Control Classes are labeled with: 🔄

• Actors in the use-case are labeled with: 🔄

NOTE: Don’t get caught up on diagram specifics
Login Use Case

1. The Customer clicks the Login button on the Home Page.
2. The system displays the Login Page.
3. The Customer enters his or her user ID and password and then clicks the Login button.
4. The system validates the login information against the persistent Account data and then returns the Customer to the Home Page.
Use Case: Step 1

• Step 1: The Customer clicks the Login button on the Home Page

Q. Why did we not explicitly model the login button?

Short Answer: Judgment call
Longer Answer: “Click log In” is a simple behavior, the whole of which can be conveyed simply.
Use Case: Step 2

• The system displays the Login Page.

A good intuitive understanding of Control Objects is that they will ultimately become methods.
Use Case: Step 3

- The Customer enters his or her user ID and password and then clicks the Login button.
Use-Case: Step 4

- The system validates the login information against the persistent Account data and then returns the Customer to the Home Page.
Conclusions

• OO is a useful design construct
• OO has a place in Architecture, but Architecture is a lot larger than OO
• Design is an iterative process
  – Requirements are incomplete
  – Requirements conflict with other requirements
  – Requirements can be non-functional and imply levels-of-service
  – What parts need to be built from scratch?
  – What are the best choices for reuse?
• Domain Modeling and Use Case Modeling are useful activities to flesh out requirements and design concerns as well as standardize on language
## CRC Cards

<table>
<thead>
<tr>
<th>Class:</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10 minute in-class

• Sit in teams
• Model the system to be built using CRC cards
DEN Discussion Board
Why not email us?

- Duplication (actually multiplication) of our efforts
- Other students may be able to reply faster than staff
- Slowing us down makes us less able to help you
- Same quality of service for all
Exceptions

• Anything involving private information
• Grades of your graded homework
• Concerns about others (team members, customers, staff)
• Not exceptions: Submission and grading questions before submission
Misc

• Don’t send unsolicited private info or documents (account quota: 100 MB)
• Separate mail alias for graders will be created
• Thank you for following these rules!