Architecture and Design Patterns

CSCI577A Fall2017
Kan Qi, Bo Wang
Outline

• In this lecture, we will cover:
  – Terminology: Architecture, Styles, Patterns, etc.
  – Examples: typical design patterns and architectural Styles.
  – Considerations & Example: Requirements & Design Interplay
  – When to Break the Rules
  – Summary
  – Workshop
Architecture is Pervasive

• Architecture is NOT a single phase
  – Design != Architecture

• Architecture is a process and artifacts
  – Requirements are essentially a set of design problems, while architecture refers to the process of solving, as well as the solution to, those problems

• All software systems have an architecture

• All software systems have an architect
Conceptual Design Methodologies

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

• Separation of Concerns
  – Isolate likely change
  – e.g., components & connectors

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

- Design Patterns
- Architectural Patterns
- Domain-Specific Software Architectures
- Styles

Application Domain Knowledge

Deep

Shallow

Programming (language level)

Application Structure

Scope

System Structure
Design Patterns

• Repeatable solutions to commonly occurring problems

• In 1994, four authors (Gang of Four) Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software

• Creational
  – E.g. Factory Method, Singleton

Structural
  – E.g. Decorator, Facade

Behavioral
  – E.g. Observer
Design Patterns: Factory Method

• The factory method allows the developer to specify an interface but to defer that actual instantiation to subclasses
• Customizable
• Often used in Framework development
Design Patterns: Singleton

- In the singleton method, only one object is instantiated for the whole program
- Introduces global state
- Easily abused!
  - Ownership
  - Lazy initialization
  - Global access
- Be attention to apply design patterns reasonably!
Design Patterns: Decorator

• Used to extend the default behavior of an object/component at runtime

```cpp
Widget* aWidget = new BorderDecorator(
    new HorizontalScrollBarDecorator(
        new VerticalScrollBarDecorator(
            new Window( 80, 24 ))));
aWidget->draw();
```
Design Patterns: Facade

- Simplify multiple disjoint interfaces into a single class
- Classic use of abstraction to interface to 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 Patterns: Observer

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Architectural Styles vs. Patterns

• Both are a known set of design decisions
  – Remember that every system has an architecture
• Patterns are more restrictive to a particular development context
  – Three-tier.
• Styles are general and can be parameterized for a context
  – Client-Server
Styles: Client-Server

- **Description**: Clients send requests to servers, which perform the required function and reply the clients with requested information. Clients initiate interactions.
- **Components**: clients, server
- **Connectors**: RPC, Network protocols
- **Examples**: web applications, mobile applications, operating systems, etc.
Styles: Object-Oriented

- **Components**: objects, including data and associated operations
- **Connectors**: method invocations
- **Style invariants**: Objects are responsible for their internal representation integrity. Internal representation is hidden from other objects.
- **Advantages**: “Infinite malleability” of object internals. System decomposition into sets of interacting agents.
- **Disadvantages**: Objects must know identities of servers. Unknown side effects in object method invocations.
**Styles: Pipe and Filter**

- **Description:** Separate programs - filters, are executed, potentially concurrently with data streams connecting the output of one filter to the input of the next.

- **Components:** Filters

- **Connectors:** Pipes

- **Examples:** The Unix Pipeline Operator
Styles: Event-based

- **Description**: Independent components asynchronously emit and receive events communicated over event buses.
- **Components**: Independent, concurrent event generators and/or consumers.
- **Connectors**: Event buses (at least one), through which data elements (events) are sent as the first-class entities.
- **Topology**: Components communicate with the event buses, not directly with each other.
- **Variants**: Communication between the components and the event bus may either be push or pull based.
- **Advantages**: Highly scalable, easy to evolve, effective for highly distributed applications.
Styles: Blackboard

- **Description**: Independent components communicate exclusively through a shared global data repository, which is called blackboard.
- **Components**: Functional units & Blackboard
- **Connectors**: Data Access
- **Examples**: Heuristic Problem Solving in A.I.
Styles: Publish-Subscribe

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

- **Components:** subscribers, publishers

- **Connectors:** Network protocols, event bus

- **Examples:** Multiplayer networked games, RSS feeds, etc.
Patterns: Three-tier

• Three tier systems are very common in business applications:
  – Front tier is traditionally focused on user interaction
  – Middle tier is business logic
  – Backend tier addresses data access and persistence
Domain-Specific Soft. Arch.

• A DSSA includes:
  – A reference architecture known to work well in a particular domain
  – Reference components, which contain reusable chunks of domain expertise
  – Methodologies for applying the architecture to particular problems

• Example: Lunar Lander DSSA
A DSSA for Lunar Lander

- Structural view of Lunar Lander DSSA
- Invariant with explicit points of variation
  - Satellite relay
  - Sensors
Design in Context

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

• One example of applying Three-tier Pattern:
  – A metadata repository called “CAS-File Manager”
CAS-File Manager Example

• Project Goals:
  – Develop a web-based application that allows a user to manage files and associated metadata, including:
    • extracting metadata from existing files
    • persisting metadata into a database

• Design Solution: Application of Three-tier Pattern
  – Web-based GUI
  – File Manager provides REST APIs
  – An underlying database to persist data and metadata.
CAS-File Manager
CAS-File Manager Example

- Web Server
  - Tomcat Web Container
  - UI Javascript/Jquery
- Infrastructure Server
  - File Manager
  - Oracle
- Client
  - Web Browser

Arrows indicate communication:
- REST
- SQL
- HTTP

Sections:
- Presentation
- Business Logic
- State
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: the existing layered architecture must add role-based security.
  – LDAP (Lightweight Directory Access Protocol) is chosen as both the authentication and authorization tool (single, gold standard).
  – The front-end must support user authentication
  – Before any action is performed, the system must validate that the user is authorized to perform the action.
CAS-Curator With Roles

Consider the up-call in the layered architecture, but allowable in favor of a single source of security information.
Summary

• Design patterns are helpful at the developer level
• Styles are good sources of inspiration, but one size does not fit all
  – Complex software systems often exhibit multiple styles
  – Breaking style rules can be OK, but make sure it’s for a good reason.
• Requirements drive design and vice versa.
Reference

[1] David Woollard, Architecture, Design Patterns and Faithful Implementation
System and Software Architecture Description (SSAD)
Outline

• The Role of Software Architecture
• SSAD
  – Intro & purpose of the SSAD
  – System analysis
  – Technology-Independent Model
  – Technology-Specific System Design
• Comments from ARB sessions
Why deal with Architecture?

• General aspects
  – All software systems have an architecture
  – All software systems have an architect

• Risk reduction benefits
  – Higher level of reasoning necessary to deal with complex systems
  – Artifacts of design decisions persist (and are useful) throughout the software lifecycle
    • You might not implement the software in this class (but rather later)
    • You will need to communicate design to stakeholders
    • Someone may still need to maintain software or add new features in 10 years
SSAD: Purpose

• SSAD describes how your system is built / going to be built
• Transferable knowledge about the system design
• SSAD captures design decisions
• SSAD captures rationale behind your decisions
SSAD: System analysis

• Rationale analysis: why?
  – Template:
    “because we identified this ..., we decided to do this ...
    ”
  e.g. “because we found that generating a schedule under given constraints is an NP-hard problem, we decided to use the following heuristics: ...
    ”
  – If not all aspects of the system designed yet, indicate that.
  – Explain how you want to address parts of the system that are not designed yet. Add plan, if necessary.
SSAD: System analysis

• System overview:
  – System Context
  – Artifacts and Information
  – Behavior (use cases)
SSAD: Technology-independent model

• Ask yourself: what would be the same in your architecture if tomorrow instead of platform/framework/language X you would use something different?

• Technology-Independent System Design is the first section of the SSAD that need not be understood by non-technical stakeholders.
SSAD: Technology-independent model

- Conceptual Domain Model
- Software / Hardware Component Diagram
- Deployment Diagram
- Interface Class Diagram
  - Indicate which parts will be developed and what views were prototyped
- Process Realization
  - Robustness diagram for use case analysis
  - Sequence diagrams for complex behavior
SSAD: Technology-specific model

- Software / Hardware Component Diagram
- Deployment / Support Software Infrastructure
- Software Class Diagram
  - Indicate which parts was / will be developed with more detail
- Process Realization
  - Robustness diagram for use case analysis
  - Sequence diagrams for complex behavior
SSAD requirements and comments

• Make sure final deliverables covers all diagrams.
• Make sure the consistency between SSAD and your implementation.
  – Once a new decision is made during the implementation, please update the SSAD
• Level of detail
  – Understand the purpose of each diagram
  – Add value to the eliminate the uncertainty and inaccuracy of the system
• Keep the terms consistent
• Use Visual Paradigm !!
Happy Halloween
Workshop

• Choose one from below
  – Pick one Design pattern which can be applied to your team project
  – Pick one Architectural style or pattern which can be applied to your team project

• Explain why you make the decision