ARCHITECTURE

Michael Weintraub
Spring, 2016
Unit Objective

• Understand what architecture is and why it is important
• Be introduced to architectural patterns

• Acknowledgment
  – This lesson was prepared with significant help by Magy Seif El-Nasr
Setting the Stage

Requirements describe the function of a system as seen by the stakeholders, especially the clients.

Given a set of requirements, the software development team must deliver a system that meets enough of those expectations to be useful.

Key design activities are inter-related

Designs will emerge during the requirements phase.
What is Design?

A Blueprint For The System

Designing is NOT coding

Coding should waiting until you understand the design, or you run a significant risk of rework
Why is Design Important?

Why can’t we skip this step?
Goal for the Design Process

Establish the design model

Should be complete
Should satisfy requirements
Should be documented, readable and clear
Design Quality

- **Functionality**
  - A design should support all the functions need given the requirements. It should be complete.

- **Modularity**
  - A design should be modular. The software should be partitioned into logical elements or subsystems.

- **Modifiability**
  - A design should enable change easily and effectively.

- **Extensibility**
  - A design should flexible so new features may be added easily.

- **Reliability/Maintainability**
  - A design should enable the quick and direct identification of problems and solutions.
Characteristics of Good Designs

- A design should exhibit an architecture
  - Created using recognizable architectural styles and patterns
  - Is composed of components that exhibit good design characteristics
  - Can be implemented in an evolutionary fashion

- A design should contain distinct representations of data, architecture, interfaces, and components

- A design should lead to components that exhibit functional characteristics
System and program design are creative activities.

User experience design is a creative activity.

The constant challenge is to find simple ways to implement complex requirements.

The highest praises for a design are:
1. It works
2. It is elegant
3. It is easy to implement, test, and maintain
Two Tools for Finding Simple Solutions

Decomposition

Breaking the system into discrete sub-systems

Sub-system

Grouping of pieces that form a part of a system.
Sub-systems may be decomposed into further sub-systems.

Abstraction

Building a model over at least two parts
Components – The Atomic Unit

A widget that provides a service.
(A system element)

1. Delivers one or more services
2. Provides services to other system elements via an interface
3. Replaceable by any other component that provides the same interfaces
   - Assumption is I/O relationship is preserved
So, How Do You Design For Good Modularity
Achieving Good Modular Design

Effective Modular Design Components

Functional Independence
Coupling
Cohesion
Module Functional Independence

Emphasizes *single-minded* function
Avoids excessive interaction with other modules

Each module addresses a specific sub-function of requirements and provides a simple interface to other modules

Functional Independence Benefits
- Easier development
- Easier to test and maintain
- Inhibits error propagation
- Reusable modules
Abstraction Barriers

- Separating the *users* of a component or layer from its *implementation*

- Enables loosely coupled systems

- Frees the client and implementer to work independently
  - So long as they stick to the contract
Describing Systems - Coupling

The degree to which two systems depend on each other

1. Decoupled
2. Loosely
3. Tightly

- Loosely coupled means the systems have little or no knowledge of the definitions of other separate components.
- Tightly coupled means there is an intimate relationship between two systems.

Loosely coupled systems are attractive because they allow for independent development and maintenance.
Describing Systems - Cohesion

The degree to which elements *within a system* belong together

In the limit, perfect cohesion would be a single atomic component doing a single function

High cohesion is preferred to low cohesion

Think clear sense of purpose or focus versus grab bag
Architecture

How the software system should be organized and the overall structure of the system

There is real overlap between requirements engineering and architecture

You will constantly move between the two in your mind’s eye.

However, the challenge will be control/temper the urge to jump too far into the design until you understand the problem well enough.
The overall design of the system

A system is a set of connected things or parts forming a complex whole, in particular. (per http://www.oxforddictionaries.com/us/)

- Computers and networks / SERVICES
  - Monolithic, distributed…

- Interfaces and protocols
  - HTTP, ODBC…

- Databases
  - Relational, distributed…

- Security
  - Authentication, authorization, privacy models…

- Operations
  - Backup, archiving, audit trails…

At this stage, you should also be selecting or at least researching:

- Software environments
- languages, database systems, class frameworks
- Testing frameworks

You should also be standardizing the modeling language

- You have UML in the toolkit
Architectural Patterns

• Nee Architectural Styles

• An abstract description of an architectural practice
  
  – It is a model. It is a principle for organizing the elements of a system.
  
  – There is no universally good architecture. They come in waves, much like trends in how building styles come and go.
Layered Architectures

- Organizes the system into layers
  - Related functionality is put into a layer
- A layer provides services (functionality) to layers above it
  - Provides isolation from underlying changes
Model-View-Controller

- **Dominant model for UI**
  - Implemented in many tools
    - Spring Framework, Ruby on Rails, Django

- **Controller**
  - Sends commands to the model to update its state.
  - Can send commands to its associated view to change the view's presentation of the model for actions that do not change the model's state.

- **Model**
  - Stores data coming from the controller
  - Displays in the view.

- **View**
  - Presents a perspective of the model.
A typical distributed architecture
- Early evolution from uni-system models
- Organized into tiers

Server
- Offers services to the preceding tier

Client
- Uses server-side services

Network
- Connects clients with servers
Distributed Data Architectures

- Aims to provide data in a highly available and efficient manner
- Key challenges are *concurrency* and *consistency* across the data sets

Replication Models
Complete data replication across two or more stores
- Either side allows updates and the sides synchronize

Cache Models
- Master has authoritative view of the data
  - Manages changes and pushes updates
- Slaves are replicas of master
- Examples are DNS and LDAP
Web Services

With the development of the WWW, the client-server tier model expanding to the idea that services could be anywhere

Emphasized loosely coupled designs

• Services are loosely coupled and independent
  – Provide consistent functionality defined by platform independent API
  – Components supply functionality and are hidden from the service consumer

• Designs shifted toward integrating services
  → Focus on standard transport (http) and data exchange (XML – eXtensible Markup Language) protocols
Web Services Implications

1. Service consumers and providers do not have to negotiate service designs

2. Late bindings
   - Can delay binding until service call actually happens
   - Broker models emerge for locating services or referrals
   - Load balancer based models emerge

3. Reduces individual compute needs as systems distribute over many hosts

4. Pay for use rather than provision models

5. Opportunistic service creation
   - Removal/deprecation is a challenge
Web Services Models Evolved to Service Oriented Architectures (SOA)
Web Services Evolved to Service Oriented Architectures (SOA)

Response identifies services and how to communicate with provider (Service Description)

Communications use SOAP (Simple Object Access Protocol) and XML
Service Oriented Architectures (SOA)

- Response identifies services and how to communicate with provider
- (Service Description)

Communications use SOAP (Simple Object Access Protocol) and XML
Service Oriented Architectures (SOA)

**Standardized service contract**
Services adhere to a communications agreement, as defined collectively by a service-description document.

**Service loose coupling**
Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.

**Service abstraction**
Beyond descriptions in the service contract, services hide logic from the outside world.

**Service reusability**
Logic is divided into services with the intention of promoting reuse.

**Service autonomy**
Services have control over the logic they encapsulate, from a Design-time and a Run-time perspective.

**Service statelessness**
Services minimize resource consumption by deferring the management of state information when necessary.

**Service discoverability**
Services are supplemented with communicative meta data by which they can be effectively discovered and used.

**Service composability**
Services are effective composition participants, regardless of the size and complexity of the composition.

Taken from Wikipedia (https://en.wikipedia.org/wiki/Service-oriented_architecture)
Standard Components

- **WS Support Services**
  - Reliable Messaging
    - Insures single delivery
  - Security
    - Policies and signatures
  - Addressing
    - Address formats within SOAP messages
  - Transactions
    - Coordinating transactions

- **Web Services Description Language**
  - Interface definitions

- **Universal Description, Discovery, and Integration**
  - Helper for service discovery

- **BPEL**
  - Defines workflows among services

- **SOAP**
  - Message transport for services

**Support Services**
- WS-Security
- WS-Addressing

**Service Definition**
- (UDDI, WSDL)

**Process**
- (WS-BPEL)

**SOAP**

**XML**
- (including XSD, XSLT, ...)

**Transport**
- (http, https, smtp, snmp, ...)

Northeastern University
WSDL

WSDL Service Definition

Abstract Section

Introduction

Interfaces

Concrete Section

Implementation

<definitions>

<!-- abstract definitions -->
<types> .

 data type definitions..
</types>

<message>

 Definition of data being communicated...
</message>

<portType>

 set of operations...
</portType>

<!-- concrete definitions -->
<binding>

 Protocol and data format specifications
</binding>

</definitions>
Hello World Example

<definitions name="HelloService"
    targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/" <!-- XML NameSpace -->

    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
    xmlns:xsd=http://www.w3.org/2001/XMLSchema <!-- XML Schema Definition -->
>

</Definition />
</Data types />

• The root element of all WSDL documents.
• Defines
  1. Name of the web service
  2. Declares multiple namespaces used throughout the remainder of the document,
  3. Contains all the service elements

Example taken from http://www.tutorialspoint.com/wSDL/wSDL_example.htm
Walking The Hello World Example

<message name="SayHelloRequest">
  <part name="firstName" type="xsd:string"/>
  <!-- XML Schema Definition -->
  <!-- A string (a name) comes in -->
</message>

<message name="SayHelloResponse">
  <part name="greeting" type="xsd:string"/>
  <!-- A string (a greeting) goes out -->
</message>

An abstract definition of the data used in the message

Example taken from http://www.tutorialspoint.com/wSDL/wSDL_example.htm
• **Port type**
  – An abstract set of operations mapped to one or more end-points.
  – *sayHello* consists of a request and a response service.

• **Operation**
  – An abstract definition of the message’s operation, such as naming the entity (method, message queue, or business process), that accepts and processes the message.

Example taken from http://www.tutorialspoint.com/wsdl/wsdl_example.htm
Walking The Hello World Example

```xml
(binding name="Hello_Binding" type="tns:Hello_PortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http/>

  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="urn:examples:helloservice" use="encoded"/>
    </input>

    <output>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                 namespace="urn:examples:helloservice" use="encoded"/>
    </output>
  </operation>
</binding>
```

- **Binding**
  - The concrete protocol and data formats for the operations and messages defined for a particular port type.
  - Directs the consumer to use SOAP / HTTP.

Example taken from http://www.tutorialspoint.com/wsdl/wsdl_example.htm
Walking The Hello World Example

```xml
<service name="Hello_Service">
  <documentation>
    WSDL File for HelloService
  </documentation>
  <port binding="tns:Hello_Binding" name="Hello_Port">
    <soap:address location="http://www.examples.com/SayHello/"/>
  </port>
</service>
```

- **Service**
  - The related end-points for the service definitions in the file; the services bind the port and include any extensibility definitions.
  - Hello_Service is available at http://www.examples.com/SayHello/

- **Port**
  - Provides the target address for the service message.
  - Identifies where the service runs: the URI http://www.examples.com/SayHello/.

Example taken from http://www.tutorialspoint.com/wsdI/wsdl_example.htm
Applying SOA in the Enterprise: Enterprise Service Bus (ESB)

- Origin in enterprise application integration
  - SOA construct

- Middleware that performs publish and subscribe functions between different applications
  1. Exchanges messages
  2. Executes transactions
  3. Orchestrates services
Orchestration

• The process of building application logic flows by stitching components of the application together through a workflow engine, message bus or service bus.

• BPEL
  – Business Process Execution Language

• ESB’s provide this via message queuing and managing
  – Component lifecycles
  – Multiple component instances
  – Converts interfaces and/or protocol that allow components to be dynamically linked without having compatible native interfaces.

• Flexibility versus Performance?
  – Obviously flexible and allows loosely coupled (organizations) systems
  – Messaging creates routing and possibly interface conversion overhead
    • Many strategies exist to speed up performance
BPEL (ws-bpel)

Primitives

<invoke>
   Invoking other Web services

<receive>
   Receives a request

<reply>
   Generates a response for synchronous operations

<assign>
   For setting data variables

<throw>
   Indicates faults and exceptions

<wait>
   Waits for some time

<terminate>
   Ends the entire process

Complex Terms

<sequence>
   – Defines a set of activities that will be invoked in an ordered sequence

<flow>
   – Defines a set of activities that will be invoked in parallel

<switch>
   – Branching

<while>
   – Loops

<pick>
   – Select one of several alternative paths

<variable>
   – Declaring variables
Approach to Writing BPEL and an Example

1. Get familiar with the involved Web services
2. Define the WSDL for the BPEL process
3. Define partner link types
4. Develop the BPEL process:
   1. Define partner links
   2. Declare variables
   3. Write the process logic definition

Taken from http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html
N-tiers Are Cool, But…

1. Is it the right decomposition (abstraction) and grouping of components if you are making adjustments or changes incrementally (small steps)?

2. Does it drive high degrees of resource utilization?

3. Does it scale into the uber-sphere?

4. Does it enable rapid development and testing as the tier grows?
What’s Also Happening

1. Networks are getting faster

2. Virtualization is getting better, slimmer, and faster

3. Notions of scale are changing

4. The expectation on time between dev and prod is changing
Micro-Services

• An approach to building complex applications using small, highly decoupled components
  – Emphasis on interfaces
• Components focus on doing a small task
  – Emphasis on highly modular development
• Makes composition a bigger deal

• Things to Consider
  – Communications overhead
  – Having more entities to manage
    • Versioning
    • Deployments
    • Managing a busy prod environment
  – fault tolerance

• How to avoid spaghetti or excessive overhead?

How do you understand the whole system?
Representational State Transfer (REST)

- Lightweight architectural model
- REST-ful
  - Anything that satisfies the REST architectural constraints
    - A uniform interface provides an abstraction barrier
      - The individual resources are identified in the request
      - If a client maintains a representation of a resource, it has enough information to MACD the resource
      - A Request contains enough information to describe how to process it
  - Session state is held client-side
    - The client passes to the server whatever state information is needed
- Interactions are defined as cacheable or not
REST Architectures

- Everything is a resource
  - Unique identifier (URI)
- Every resource has four operations

**Generic Resource and Actions**
- Create
- Read
- Update
- Delete

**Web Resource and Actions**
- POST
- GET
- PUT
REST Implementations

- **Services may not be XML-based**
  - Alternates like JSON are acceptable
    - SOAP requires XML
    - JSON: Javascript Object Notation
  - Lighter-weight than SOAP

- **Self defined QoS monitoring and management**
  - No reliance on WS-Reliability or WS-Transactions

**Message**

```
GET https://www.googleapis.com/language/translate/v2/detect?key=AIzaSyA-X23EG3eTTGXsCeIZhY438obFCzsYkLo
&source=en&q=Managing%20Software%20Development&target=fi
```

**Returns JSON**

```
200 OK
{
  "data": {
    "translations": [
      {
        "translatedText": "toimitusjohtaja ohjelmistojen suunnittelu"
      }
    ]
  }
}
```

Example of a RESTful call to Google's translate service
Final thoughts on Good Design (from Anders Toxboe)

- **Good design is innovative**
  - Innovative design can both be a *break-through* product or service, and a redesign of an existing product or service.

- **Good design is functional**
  - Useful design fills its intended function – and likely both a primary and secondary function. A useful design solves problems and through its design it optimizes a given functionality.

- **Good design makes a product useful**
  - “It has to satisfy certain criteria, not only functional, but also psychological and aesthetic. Good design emphasises the usefulness of a product whilst disregarding anything that could possibly detract from it.

- **Good design is aesthetic**
  - An aesthetic product has an inherent power of being able to fascinate and immediately appeals to its users senses.

- **A good design is intuitive**
  - Intuitive design explains itself and makes a user manual unnecessary. A design makes how to use, perceive, and understand a product obvious. A good design explains its function.
  - “It clarifies the product’s structure. Better still, it can make the product clearly express its function by making use of the user’s intuition. At best, it is self-explanatory.

- **A good design is long-lasting**
  - “[…]It avoids being fashionable and therefore never appears antiquated. Unlike fashionable design, it lasts many years …”

- **A good design is user-oriented**

- **A good design is unobtrusive**

- **A good design is thorough – down to the last detail**
  - “Nothing must be arbitrary or left to chance. Care and accuracy in the design process show respect towards the consumer.”

- **A good design is always the simplest possible working solution.**
  - **A good design is as little design as possible**

- **A good design is focused**

- **A good design is effective and efficient in fulfilling its purpose.**
  - It relies on as few external factors and inputs as possible, and these are easy to measure and manipulate to achieve an expected other output. A good design is always the simplest possible working solution.

http://ui-patterns.com/blog/What-is-good-design