Introduction I

Up to now, your programs have:
- Small
- Text/file
- Run locally
- Single user
- Some thick GUI (swing, gtk, “visual” MS products)
- Little to no interaction with other programs

Introduction II

Actual software is far more complex

Today, most software exists, not to solve a problem, but to interface with other software.

—I.O. Angell

Large, well-designed applications make use of architectural design.

- Software design on a larger-scale
- Make extensive use of databases
- Distributed, multi-user
- Organize responsibilities and interaction into tiers

Introduction III

Beware:
- Architecture Design can be loaded with buzz-wordiness
- Weasel-words; “Enterprisey-ness”
- Plagued with fads, “Cloud Computing”
- Seminar salesmen

1-Tier architecture I

Applications with a single-tier:
- Installed programs (thick clients)
- Access data locally (files, proprietary formats)
- Examples: MS Excel, winamp, etc.
- Single user (or resource allocation problems)
- Terminals with a networked file server

1-Tier architecture II

Figure: Single Tier Structure

2-Tier architecture I

- Client-Server Architecture
- Client runs the application: business logic & presentation
- Server handles persistence of data
- Client-server communicate back and forth
- More secure: authentication and role-based data protection
- Distributed, supports concurrency

2-Tier architecture II

![2-Tier Structure](image)

3-Tier architecture I

**Definition**

A 3-tier architecture is one which has a client tier, a middle tier, and a database tier.

- Database tier manages the data/database
- Middle tier contains most of the “business logic” and communicates between client and data tier
- Client tier is the interface between the user and the system, responsible for presentation (presentation layer)

Note: presentation layer may not necessarily be for human users!

3-Tier architecture II

![3-Tier Structure](image)

3-Tier architecture III

- Separation of tiers allows for modularity on a large scale
- Each tier can be completely different technologies
- Each tier can contain differing technologies (not a great situation)
- Interoperability

3-Tier architecture IV

**Examples**

- Database: MSSQL, Postgres, Oracle, MySQL, etc.
- Middle Tier: C#, .NET, ASP, JSP, Java, PHP, C++, Perl
- Frameworks: Rails, etc.
- Thick clients: Java apps (Swing, AWT–Abstract Window Toolkit), C++ (Microsoft Foundation Classes, GTK, Qt)
- Thin clients (Browser based, cross-browser compatibility issues): JavaScript and various emergent frameworks (Node, AngularJS, Backbone.js, Ember.js, etc. etc.)
- In-betweeners: MS Silverlight, Flash, Java Applets
- RESTful Services
N-Tier architecture I

More complex systems demand greater abstraction and interaction

▶ Easy to generalize concept into $N$-tiers for as many layers as you want to design
▶ Scalability demands distributed hardware, more abstraction needed to delegate communication
▶ At each level, software may interact (sideways) with other components (web services, wrapper interactions)
▶ Software as a Service (SaaS, PaaS, NaaS, IaaS)
▶ May have additional, orthogonal tiers that provide ancillary services

N-Tier architecture II

Example:

▶ Database Layer
▶ Data Access Layer (JPA, ADO)
▶ First level business tier (core level)
▶ Second level business tier (client-level to provide customizations for certain clients or users)
▶ Validation Layer
▶ Presentation Layer
▶ Templating layer

Design Decisions I

Tiered Architecture Design

▶ How many abstraction layers?
▶ What should be responsible for what?
▶ Thick or thin client (or both?)
▶ Webapp?
▶ How to manage data?
▶ Languages, platforms, technology, software, etc?

Design Decisions II

Factors

▶ User base (size, location, requirements)
▶ Budget (Correctness, Quickness, Cheapness: choose two)
▶ Available resources & knowledge
▶ Hardware

Design Decisions III

Systems evolve to meet new requirements, scale, etc.

▶ Design may need significant changes
▶ Additional layers introduced
▶ Refactoring
▶ Migrations
▶ Introduction of new techniques or technologies
▶ May be necessitated by market supply (programmers)

Design Decisions IV

Good design:

▶ True abstraction, loose coupling
▶ Good modularity means and entire layer can be replaced wholesale with no consequence
▶ Interoperable standards, protocols and technologies (WSDL, ECMA, XML, etc)
▶ Software as a Service, Service Oriented Architecture, multiple access methods
▶ Leaves a system easily extensible, updatable
Design Decisions V

Bad design:
- Demarcation of responsibility becomes blurred
- Business logic in the database (stored procedures)
- Presentation logic in the middletier (Servlets generating HTML)
- DBAs should not have to be application developers & vice versa
- Bugs harder to track down
- "Technical debt"\(^2\)

Design Decisions VI

Successful design requires the perfect storm of:
- Good design
- Appropriate resources
- Competent developers
- Motivated developers
- Sustainability
- Good management

\(^{2}\) Also known as the Iron Triangle of Project Management

MVC I

Model-View-Controller (MVC) architecture
- Popular realization of 3-tier architecture
- Introduced in Smalltalk (1970)
- Supported by many frameworks
- As usual: design principle, can (and often is) broken

MVC II

Model-View-Controller (MVC)

![Diagram of Model-View-Controller](Figure: Model-View-Controller)

MVC III

Model:
- Set of objects that represent & support the underlying problem
- Ideally: should be invariant (does not change as long as problem does not change)
- Exists & defined independent of data model, presentation

MVC IV

View:
- Application may have more than one interface: thick GUI, webapp, command line
- May expose application as webservice: no human interface at all
- Example: Lotus Notes (supports both thick client and web GUI)
- Interface may change
- May differ for users with different roles/rights/responsibilities
- Allows complex systems to be customized to different customers
- Modularization and composition of features (pay per play, a la carte)
**MVC V**

**Controller:**
- “handles the input”
- View and controller may essentially be the same; integrated
- Delegates interaction between model and view

**JSP Models I**

- JavaServer Pages (JSP) introduced in 1999
- Similar technologies: PHP, Active Server Pages (MS)
- Runs within an application server (platform independent)
- Examples: JBoss, Tomcat, Glassfish

**JSP Model 1 I**

- Initial, old pattern
- Client interacts with a JSP page
- One “page” responsible for:
  - Processing request
  - Validating data
  - Business logic
  - Generating response (or redirecting)
- May utilize JavaBean
- Good for small applications, quick development
- Less separation of business logic/presentation
- Lots of code duplication

**JSP Model 1 II**

- Use of Servlets (controller) to delegate processing requests
- Promotes better separation of BL/presentation
- More in-line with MVC
- Dynamically configurable

**JSP Model 2 I**

- Additional frameworks and technologies introduced
  - Struts2
  - EJBs
  - JAX-WS

**JSP Model 2 II**

- Figure: JSP Model 2
Software Engineering I

Software Engineering: much more

- Software Development Methodologies
  - Waterfall
  - Agile Family: Scrum, Extreme, etc.
  - Kanban
- Business Requirements
- Design Specifications
- Use Cases

Software Engineering II

- Testing & Quality Assurance
  - Unit Testing
  - Integration Testing
  - Automated Testing
  - Regression Testing
  - Performance & Load Testing
- Version Control
  - Content Versioning System (CVS)
  - SVN
  - Git
  - https://git.wiki.kernel.org/index.php/GitSvnComparison

Software Engineering III

- Build & Deployment measures
  - make
  - Ant
  - Apache Maven
- Legal & Ethical issues
  - IP: Copyright, patents, licenses
  - Privacy
  - Accountability & compliance (Sarbanes-Oxley, NIST)
- Many others

Web Application

Figure: 3 Tier Web Application

Being a Software Engineer I

A software engineer has to sometimes be a

- Lawyer
- Philosopher
- Diplomat
- Designer
- Manager
- Writer
- Programmer