Software Engineering

Software Architecture:
Architectural design, System decomposition, and Distribution architecture

1102SE05
MBA, IM, NTPU (M5010) (Spring 2022)
Wed 2, 3, 4 (9:10-12:00) (B8F40)

Min-Yuh Day, Ph.D,
Associate Professor

Institute of Information Management, National Taipei University
https://web.ntpu.edu.tw/~myday

2022-03-30
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Subject/Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2022/02/23</td>
<td>Introduction to Software Engineering</td>
</tr>
<tr>
<td>2</td>
<td>2022/03/02</td>
<td>Software Products and Project Management:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software product management and prototyping</td>
</tr>
<tr>
<td>3</td>
<td>2022/03/09</td>
<td>Agile Software Engineering:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Agile methods, Scrum, and Extreme Programming</td>
</tr>
<tr>
<td>4</td>
<td>2022/03/16</td>
<td>Features, Scenarios, and Stories</td>
</tr>
<tr>
<td>5</td>
<td>2022/03/23</td>
<td>Case Study on Software Engineering I</td>
</tr>
<tr>
<td>6</td>
<td>2022/03/30</td>
<td>Software Architecture: Architectural design,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>System decomposition, and Distribution architecture</td>
</tr>
<tr>
<td>Week</td>
<td>Date</td>
<td>Subject/Topics</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>2022/04/06</td>
<td>Make-up holiday (No Classes)</td>
</tr>
<tr>
<td>8</td>
<td>2022/04/13</td>
<td>Midterm Project Report</td>
</tr>
<tr>
<td>9</td>
<td>2022/04/20</td>
<td>Cloud-Based Software: Virtualization and containers, Everything as a service, Software as a service</td>
</tr>
<tr>
<td>10</td>
<td>2022/04/27</td>
<td>Cloud Computing and Cloud Software Architecture</td>
</tr>
<tr>
<td>11</td>
<td>2022/05/04</td>
<td>Microservices Architecture, RESTful services, Service deployment</td>
</tr>
<tr>
<td>12</td>
<td>2022/05/11</td>
<td>Industry Practices of Software Engineering</td>
</tr>
</tbody>
</table>
# Syllabus

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Subject/Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>2022/05/18</td>
<td>Case Study on Software Engineering II</td>
</tr>
<tr>
<td>14</td>
<td>2022/05/25</td>
<td>Security and Privacy; Reliable Programming; Testing: Test-driven development, and Code reviews; DevOps and Code Management: DevOps automation</td>
</tr>
<tr>
<td>15</td>
<td>2022/06/01</td>
<td>Final Project Report I</td>
</tr>
<tr>
<td>16</td>
<td>2022/06/08</td>
<td>Final Project Report II</td>
</tr>
<tr>
<td>17</td>
<td>2022/06/15</td>
<td>Self-learning</td>
</tr>
<tr>
<td>18</td>
<td>2022/06/22</td>
<td>Self-learning</td>
</tr>
</tbody>
</table>
Software Architecture:
Architectural design, System decomposition, and Distribution architecture
Software Engineering
and
Project Management

Analyse
- Requirements definition

Design
- System and Software design

Build
- Implementation and unit testing

Test
- Integration and system testing

Deliver
- Operation and maintenance

Project Management
Information Management (MIS) Information Systems

Fundamental MIS Concepts

Project-based software engineering

CUSTOMER
generates
implemented-by
DEVELOPER
helps-with

1 Requirements

CUSTOMER and DEVELOPER

Software execution models

**Stand-alone execution**
- User’s computer
  - User interface
  - Product functionality
  - User data
- Vendor’s servers
  - Product updates

**Hybrid execution**
- User’s computer
  - User interface
  - Partial functionality
  - User data
- Additional functionality
  - User data backups
  - Product updates
- Vendor’s servers

**Software as a service**
- User’s computer
  - User interface (browser or app)
- Product functionality
  - User data
- Vendor’s servers

- User data backups
- Product updates

Product management concerns

Business needs

Product manager

Technology constraints

Customer experience

Technical interactions of product managers

- Product vision management
- Product backlog management
- Acceptance testing
- User stories and scenarios
- User interface design
- Customer testing

Software Development Life Cycle (SDLC)
The waterfall model

Plan-based and Agile development

Plan-based development

- Requirements engineering
- Requirements specification
- Design and implementation

Requirements change requests

Agile development

- Requirements engineering
- Design and implementation

The Continuum of Life Cycles

Predictive Life Cycle

Analyze → Design → Build → Test → Deliver

Iterative Life Cycle

A Life Cycle of Varying-Sized Increments

Iteration-Based and Flow-Based Agile Life Cycles

**Iteration-Based Agile**

- Requirements Analysis
- Design
- Build
- Test

- Repeat as needed

**Flow-Based Agile**

- Requirements Analysis
- Design
- Build
- Test
  - the number of features in the WIP limit

- Repeat as needed

From personas to features

1. Personas
   A way of representing users

2. Scenarios
   Natural language descriptions of a user interacting with a software product

3. Stories
   Natural language descriptions of something that is needed or wanted by users

4. Features
   Fragments of product functionality

Multi-tier client-server architecture

Service-oriented Architecture

Everything as a service

- **Infrastructure as a service (IaaS)**
  - Cloud data center

- **Platform as a service (PaaS)**
  - Photo editing

- **Software as a service (SaaS)**
  - Logistics management
  - Cloud management
  - Monitoring
  - Storage
  - Network

Software as a service

Microservices architecture – key design questions

- What are the microservices that make up the system?
- How should microservices communicate with each other?
- How should data be distributed and shared?
- How should the microservices in the system be coordinated?
- How should service failure be detected, reported and managed?

Types of security threat

Availability threats
An attacker attempts to deny access to the system for legitimate users

Data theft

Integrity threats
An attacker attempts to damage the system or its data

Distributed denial of service (DDoS) attack

Virus

Confidentiality threats
An attacker tries to gain access to private information held by the system

Ransomware
Software product quality attributes

1. Reliability
2. Availability
3. Resilience
4. Maintainability
5. Responsiveness
6. Usability
7. Security

A refactoring process

1. Identify code ‘smell’
2. Identify refactoring strategy
3. Make small improvement until strategy completed
4. Run automated code tests

Functional testing

Start

1. Unit Testing
2. Feature Testing
3. System Testing
4. Release Testing

Test-driven development (TDD)

1. Identify new functionality
2. Identify partial implementation of functionality
3. Write code stub that will fail test
4. Run all automated test
5. Implement code that should cause failing test to pass
6. Run all automated test
7. Refactor code if required

Functionality complete
Functionality incomplete
All tests pass

DevOps

Development

Deployment

Support

Multi-skilled DevOps team

Code management and DevOps

DevOps automation

- Continuous integration
- Continuous deployment
- Continuous delivery
- Infrastructure as code

Code management system

- Branching and merging
  - Recover version information
  - Code repository
  - Save and retrieve versions
- Transfer code to/from developer’s filestore

DevOps measurement

- Data collection
- Data analysis
- Report generation

Software Architecture:
Architectural design, System decomposition, and Distribution architecture
Software architecture

• To create a reliable, secure and efficient product, you need to pay attention to architectural design which includes:

  • its overall organization,
  • how the software is decomposed into components,
  • the server organization
  • the technologies that you use to build the software. The architecture of a software product affects its performance, usability, security, reliability and maintainability.

Software architecture

• There are many different interpretations of the term ‘software architecture’.
  • Some focus on ‘architecture’ as a noun
    - the structure of a system
  and others consider ‘architecture’ to be a verb
  - the process of defining these structures.

The IEEE definition of software architecture

• **Architecture** is the fundamental organization of a software system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

Software architecture and components

• A **component** is an element that implements a coherent set of functionality or features.

• **Software component** can be considered as a collection of one or more services that may be used by other components.

• When **designing software architecture**, you don’t have to decide how an architectural element or component is to be implemented.

• Rather, you **design the component interface** and leave the implementation of that interface to a later stage of the development process.

Access to services provided by software components

Services accessed directly by other components

Component 1

Services accessed through the component API

Component 2

Why is architecture important?

• **Architecture** is important because the architecture of a system has a fundamental influence on the non-functional system properties.

• **Architectural design** involves understanding the issues that affect the architecture of your product and creating an architectural description that shows the critical components and their relationships.

• **Minimizing complexity** should be an important goal for architectural designers.

Non-functional system quality attributes

• **Responsiveness**
  Does the system return results to users in a reasonable time?

• **Reliability**
  Do the system features behave as expected by both developers and users?

• **Availability**
  Can the system deliver its services when requested by users?

• **Security**
  Does the system protect itself and users’ data from unauthorized attacks and intrusions?

Non-functional system quality attributes

• **Usability**
  Can system users access the features that they need and use them quickly and without errors?

• **Maintainability**
  Can the system be readily updated and new features added without undue costs?

• **Resilience**
  Can the system continue to deliver user services in the event of partial failure or external attack?

Centralized security architectures

• The benefits of a centralized security architecture are that it is easier to design and build protection and that the protected information can be accessed more efficiently.

• However, if your security is breached, you lose everything.

• If you distribute information, it takes longer to access all of the information and costs more to protect it.

• If security is breached in one location, you only lose the information that you have stored there.

Shared database architecture

Multiple database architecture

User interface

C1

C1 database

C2

C2 database

C3

Database reconciliation

Maintainability and performance

• **Shared database architecture:**
  - system with two components (C1 and C2) that share a common database.

• **Multiple database architecture:**
  - each component has its own copy of the parts of the database that it needs.
  - If one component needs to change the database organization, this does not affect the other component.

• A multi-database architecture may run more slowly and may cost more to implement and change.
  - A multi-database architecture needs a mechanism (component C3) to ensure that the data shared by C1 and C2 is kept consistent when it is changed.

Issues that influence architectural decisions

1. Nonfunctional product characteristics
2. Product lifetime
3. Software reuse
4. Number of users
5. Software compatibility

The importance of architectural design issues

1. Nonfunctional product characteristics

Nonfunctional product characteristics such as security and performance affect all users. If you get these wrong, your product will is unlikely to be a commercial success. Unfortunately, some characteristics are opposing, so you can only optimize the most important.

The importance of architectural design issues

2. Product lifetime

If you anticipate a long product lifetime, you will need to create regular product revisions. You therefore need an architecture that is evolvable, so that it can be adapted to accommodate new features and technology.

The importance of architectural design issues

• Software reuse

You can save a lot of time and effort, if you can reuse large components from other products or open-source software. However, this constrains your architectural choices because you must fit your design around the software that is being reused.

The importance of architectural design issues

4. **Number of users**

   If you are developing consumer software delivered over the Internet, the number of users can change very quickly. This can lead to serious performance degradation unless you design your architecture so that your system can be quickly scaled up and down.

The importance of architectural design issues

• **Software compatibility**

  For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.

Trade off: Maintainability vs performance

- System **maintainability** is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers.
  - You improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required.
- In architectural terms, this means that the system should be **decomposed into fine-grain components**, each of which does one thing and one thing only.
  - However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature, the software will be slower.

Trade off: Security vs usability

- You can achieve security by designing the system protection as a series of layers.

- An attacker has to penetrate all of those layers before the system is compromised.

- Layers might include system authentication layers, a separate critical feature authentication layer, an encryption layer and so on.

- Architecturally, you can implement each of these layers as separate components so that if one of these components is compromised by an attacker, then the other layers remain intact.

Authentication layers

Usability issues

• A layered approach to security affects the usability of the software.
  • Users have to remember information, like passwords, that is needed to penetrate a security layer. Their interaction with the system is inevitably slowed down by its security features.
  • Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate to access system features or data.

• To avoid this, you need an architecture:
  • that doesn’t have too many security layers
  • that doesn’t enforce unnecessary security
  • that provides helper components that reduce the load on users

# An architectural model of a document retrieval system

<table>
<thead>
<tr>
<th>Web browser</th>
<th>User interaction</th>
<th>Local input validation</th>
<th>Local printing</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interface</td>
<td>Authentication and authorization</td>
<td>Form and query manager</td>
<td>Web page generation</td>
</tr>
<tr>
<td>management</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information</td>
<td>Search</td>
<td>Document retrieval</td>
<td>Rights management</td>
</tr>
<tr>
<td>retrieval</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Document index</td>
<td>Index management</td>
<td>Index querying</td>
<td>Index creation</td>
</tr>
<tr>
<td>Basic services</td>
<td>Database Query</td>
<td>Query validation</td>
<td>Logging</td>
</tr>
<tr>
<td>Databases</td>
<td>DB1</td>
<td>DB2</td>
<td>DB3</td>
</tr>
</tbody>
</table>

Examples of component relationships

C1 is part of C2

C1 uses C2

C1 is-located-with C2

C1 shared-data-with C2

Architectural design guidelines

**Separation of concerns**
Organize your architecture into components that focus on a single concern

**Stable interfaces**
Design component interfaces that are coherent and that changes slowly

**Implement once**
Avoid duplicating functionality at different places in your architecture

Cross-cutting concerns

Security  Performance  Reliability

User interface
Application
Infrastructure
Operating System
Hardware

A generic layered architecture for a web-based application

- Browser-based or mobile user interface
- Authentication and user interaction management
- Application-specific functionality
- Basic shared services
- Transaction and database management

## A layered architectural model of the iLearn system

<table>
<thead>
<tr>
<th>User interface</th>
<th>Web browser</th>
<th>iLearn app</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interface management</td>
<td>Interface creation</td>
<td>Forms management</td>
</tr>
<tr>
<td>Configuration services</td>
<td>Group configuration</td>
<td>Application configuration</td>
</tr>
<tr>
<td>Application services</td>
<td>Archive access</td>
<td>Word processor</td>
</tr>
<tr>
<td>Integrated services</td>
<td>Resource discovery</td>
<td>User analytics</td>
</tr>
<tr>
<td>Shared infrastructure services</td>
<td>Authentication</td>
<td>Logging and monitoring</td>
</tr>
</tbody>
</table>

Distribution architecture

• The distribution architecture of a software system defines the servers in the system and the allocation of components to these servers.

• Client-server architectures are a type of distribution architecture that is suited to applications where clients access a shared database and business logic operations on that data.

• In this architecture, the user interface is implemented on the user’s own computer or mobile device.
  
  • Functionality is distributed between the client and one or more server computers.

Client-server architecture

The Model-View-Controller (MVC) pattern

Mobile Web App

HTML

Templates

CSS

JavaScript

Mobile frameworks and Libraries

Phone Data

External Data

Source: Scott Preston, Learn HTML5 and JavaScript for iOS: Web Standards-based Apps for iPhone, iPad, and iPod touch, Apress, 2012
MVC Framework of Mobile Apps (HTML5, CSS3, JavaScript)

Source: http://sc5.io/blog/2012/02/anatomy-of-a-html5-app/
Multi-tier client-server architecture

Service-oriented Architecture

• Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another.

• Many servers may be involved in providing services.

• A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

Service-oriented Architecture

Issues in architectural choice

• Data type and data updates
• Change frequency
• The system execution platform

Issues in architectural choice

• Data type and data updates
  • If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data is distributed across services, you need a way to keep it consistent and this adds overhead to your system.

Issues in architectural choice

• Change frequency

  • If you anticipate that system components will be regularly changed or replaced, then isolating these components as separate services simplifies those changes.
Issues in architectural choice

• The system execution platform
  • If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler.
  • If your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

Technology choices

• **Database**
  Should you use a relational SQL database or an unstructured NOSQL database?

• **Platform**
  Should you deliver your product on a mobile app and/or a web platform?

• **Server**
  Should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?

• **Open source**
  Are there suitable open-source components that you could incorporate into your products?

• **Development tools**
  Do your development tools embed architectural assumptions about the software being developed that limit your architectural choices.

Summary

• **Software architecture** is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

• **The architecture of a software system** has a significant influence on **non-functional system properties** such as reliability, efficiency and security.

• **Architectural design** involves understanding the issues that are critical for your product and creating system descriptions that shows components and their relationships.

Summary

• The principal role of architectural descriptions is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.

• System decomposition involves analyzing architectural components and representing them as a set of finer-grain components.

Summary

• To minimize complexity, you should separate concerns, avoid functional duplication and focus on component interfaces.

• Web-based systems often have a common layered structure including user interface layers, application-specific layers and a database layer.

• The distribution architecture in a system defines the organization of the servers in that system and the allocation of components to these servers.

Summary

• **Multi-tier client-server** and **service-oriented architectures** are the most commonly used architectures for web-based systems.

• **Making decisions on technologies such as database and cloud technologies** are an important part of the architectural design process.

References


• Titus Winters, Tom Manshreck, and Hyrum Wright (2020), Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media.

