**Engineering Large Software Systems Notes**

**References**

* Engineering Large Software Systems course at the University of Toronto

**Prerequisites**

1. Design pattern theory (eg. factory, builder, observer, strategy, etc.)

* Observer: whenever action occurs, observers will be notified, aka. listeners.
* Factory: a class that can generate more classes.
* Strategy: similar to factory but dealing with functions.

2. Testing code

* Unit testing
* Integration testing

3. Code smells

* Anti patterns when writing code

4. Code design principles (eg. SOLID)

* S: single responsibility principle, every module should focus on one task.
* O: open-close principle, open for extension closed for modification.

5. Git usage

* Git merge vs rebase

## 1. Large Software Systems

***Large***: Numerous contributors, impacts many stakeholders.

***Software Engineering***: six step process of creating software through SDLC, e.g., agile, waterfall. 1. requirements engineering. 2. system design. 3. implementation. 4. testing. 5. deployment. 6. maintenance.

### 1.0. SDLC Models

|  |  |
| --- | --- |
| **Agile** | **Waterfall** |
|  |  |
| Iterative development process. | Defined requirements at the beginning. |
| Client can change requirements. | Scope does not change. |

|  |  |
| --- | --- |
| **Iterative & Incremental Model** | **V Model** |

### Requirements Engineering

|  |  |
| --- | --- |
| ***Requirements Engineering***: Discovering and documenting requirements necessary for project success through talking with client, interviews, surveys.  ***Known Requirements***: What users told us.  ***Overlooked Requirements***: What users didn’t tell us yet.  ***Emergent Requirements***: What will surface while building product.  ***Functional Requirements***: What a system should do; system features.  ***Non-Functional Requirements***: How a system should perform (performance, reliability, usability, security, scalability). | Figure: Relationship between types of system requirements. |

|  |
| --- |
| TODO: chaos report 1995 standish group  <https://www.csus.edu/indiv/r/rengstorffj/obe152-spring02/articles/standishchaos.pdf> |

### 1.2. System Design

|  |  |
| --- | --- |
| ***System Design***: Defining software architecture of a system rather than solving code problems, solving structural problems.   * Non-functional requirements and how to build it * Creating software architecture documents * Architecture risks * How to calculate cost for infrastructure   **Deciding on Tradeoffs**   * Consistency, availability, partition tolerance * Performance * Maintainability | Figure: ***CAP Theorem***: can only satisfy 2/3 of these characteristics. |

### 1.3. Implementation

***Implementation***: Writing the code based on identified requirements and adherence to system design.

**Costs of Implementing Software**

1. Labor: developers, architects, management
2. Infrastructure: production and test environments
3. Maintenance: documentation, change management, tech debt

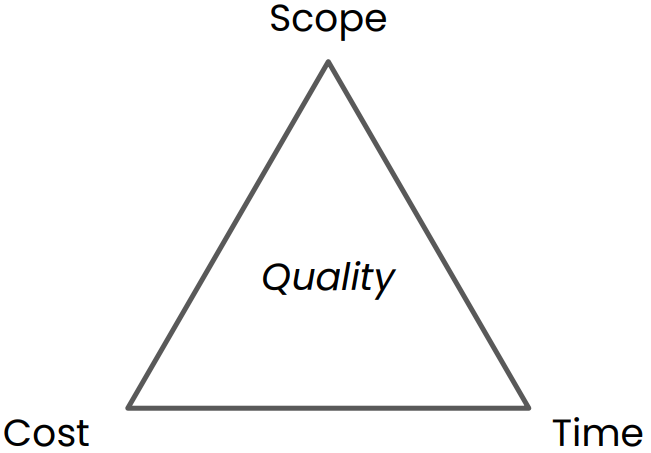


Figure: interdependence of scope, cost, and time on software quality.

**Evaluating the Success of a Project**

1. delivering on time
2. on scope
3. at cost.

### 1.4. Testing

### 1.5. Deployment

***Deployment***: Making software available to users.

### 1.6. Maintenance

***Maintenance***: Ensuring software continuously satisfy users.

### 1.7. Contributing to Open Source

***Open/Closed source software***: whether source code of a software program is public.

* Advantage of closed source: security by obscurity.
* Open source doesn’t mean free. Depends on licence.
* Free and open source software (FOSS) is both free and open source.
* Value of open source software: makes a developer’s life easier, e.g., git, TensorFlow, React.
* Why contribute? Get hired faster, access to industry talent, work on your craft.

**Software Licences**

|  |  |
| --- | --- |
| <https://www.gnu.org/philosophy/categories.en.html> |  |

**Contribute to Which Project?**

* GitHub stars,
* “used by” count
* Low barrier to contribute
  + Large number of contributors
  + Large rate of contributions over time (commits over time)
  + Good documentation (e.g., readme)
  + Simple development setup (Time to hello world)
  + Streamlined (easy to understand) design
  + Healthy project

**How to Contribute?**

* Look for contributor documentation

**Issue Hunting**

* Discover gaps in project by being a user
* GitHub issues: filter by tags such as “help wanted”, “good first issue”.
* Messaging forums: discord, reddit, GitHub discussions