Engineering Large Software Systems Notes

Created: 2024-09-04
Updated: 2024-09-14

References

- Engineering Large Software Systems course at the University of Toronto

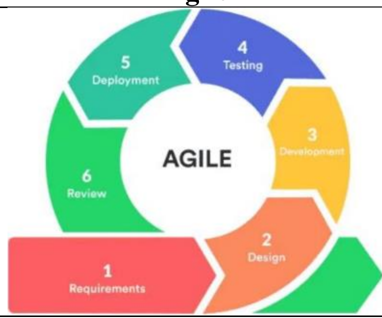
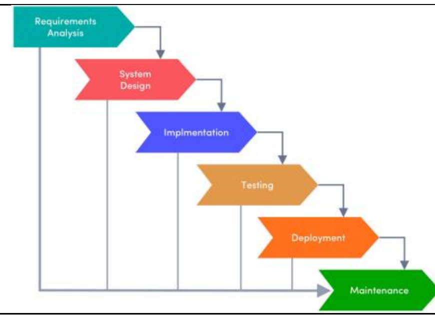
Prerequisites

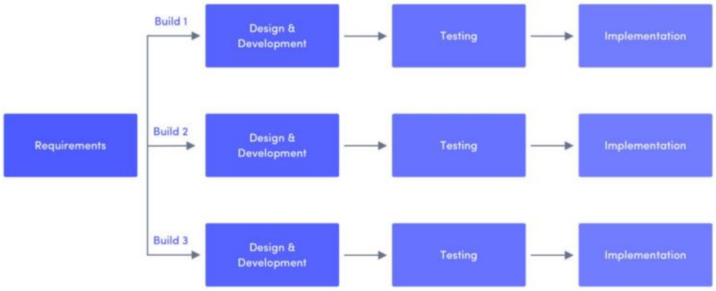
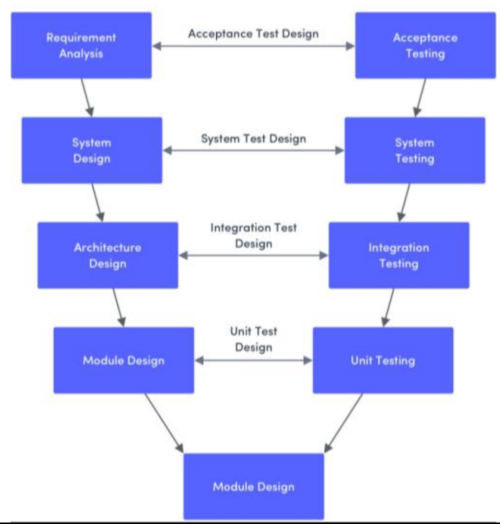
1. Design pattern theory (eg. factory, builder, observer, strategy, etc.)
   - Observer: whenever action occurs, observers will be notified, aka. listeners.
   - Factory: a class that can generate more classes.
   - Strategy: similar to factory but dealing with functions.
2. Testing code
   - Unit testing
   - 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.
<u>Software Engineering</u>: 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

<table>
<tr><th>Agile</th><th>Waterfall</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>Iterative development process.</td><td>Defined requirements at the beginning.</td></tr>
<tr><td>Client can change requirements.</td><td>Scope does not change.</td></tr>
</table>

<table>
<tr><th>Iterative & Incremental Model</th><th>V Model</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

1.1. 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: Requirements that enable a system to operate, maintain, and extend in quality.

- ***Operational NFRs***: performance of a system.
- ***Structural NFRs***: code quality of the system.

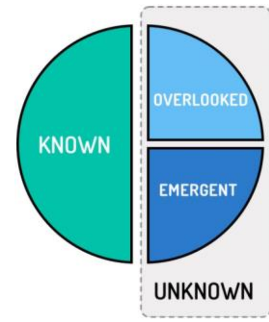


Figure: Relationship between types of system requirements.

1.2. System Design

System Design: Defining software architecture of a system; solving structural and not code problems.

- Creating software architecture documents
- Architecture risks
- How to calculate cost for infrastructure
- Increase in **NFRs** aims to increase quality and performance of system at cost of capital and time.
- ***Vertical Scaling***: improving existing infrastructure by switching to better hardware.
- ***Horizontal Scaling***: adding/expanding more infrastructure e.g., more servers. Requires additional component to manage/integrate new additions e.g., load balancer server, Kubernetes. Creates network overhead.
    - Horizontal scaling costs more than vertical scaling because increasing spec is generally proportional to cost while adding a unit incurs extra cost with integration with the rest of the infrastructure.
    - Not practical to use only vertical scaling because there's an upper limit to how good specs are.
    - Determine scaling method by first estimating load, expected requests per second (RPS).
- ***Scalability***: ability for application to respond well to heavy load.
- ***Elasticity***: ability for application to scale up with sudden demand.

Deciding on Tradeoffs

- ***Consistency***: every read returns most recent write or error.
- ***Availability***: every request to non-failing node must result in response even in server failure and update/maintenance (security updates, infrastructure version updates).
    - Metric: time-based availability. **Availability = uptime / (uptime + downtime)**.
        - Periodic request to **health check endpoint** to confirm server responsive to requests, API and DB connections/latency, cache warmed.
        - **Service-Level Agreement** (SLA), e.g., 99.9% availability allows only 43 min downtime/month. Reducing downtime involves automated recovery and alerting.
    - Metric: aggregated availability. **Availability = successful requests / total requests**.
- ***Partition tolerance***: system continues to operate even if some nodes fail. E.g., a system depending on a single database is not partition tolerant; solution: source-replica DB replication (lacks consistency in CAP assuming asynchronous writes to DB. ADIC principle. Synchronous writes loses availability but gains consistency.

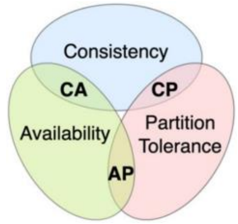


Figure: ***CAP Theorem***: can only satisfy 2/3 of these characteristics.

- Low ***latency***: response time to request.
- High **performance**: improve performance by improving data retrieval (e.g., caching) and compute time.
    - ***Caching***: method to increase data retrieval performance. Retrieve from RAM instead of disk. Perform computation in advance instead of on request. E.g., memcached, redis.
    - ***Read-through caching***: data loaded into cache on demand.
    - ***Write-through caching***: write operation writes to DB and cache.

Cache	Pro	Con
Read-through	Fast for frequently accessed data and reduces load on DB.	Slow initial read if not cached. Infrequent calls does not benefit from performance boost.
Write-through	Fast for accessing recently written data.	Slower write performance. Writing too much.
Read-through and write-through	Fast access for most recent data.	Performance overhead. Complexity in implementation and maintenance.

    - Increase in performance with caching leads to a:
        - decrease in code readability, freshness of data (data synchronization).
        - increase in maintenance cost (more code (bugs), infrastructure cost (reddis), memory usage.
- Size of input / output.
    - Decreasing size of output:
        - Reduces transport time.
        - Increases time to compress and decompress.
- Maintainability
- Cost

TODO: o'reilly site reliability engineering

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

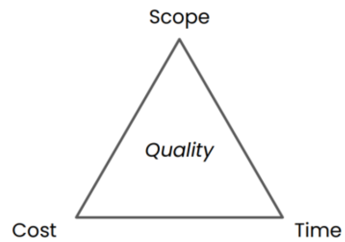


Figure: interdependence of scope, cost, and time on software quality.

Evaluating the Success of a Project

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

See Appendix A – The CHAOS Report (1995)

1.4. Testing

1.5. Deployment

Deployment: Making software available to users.

1.6. Maintenance

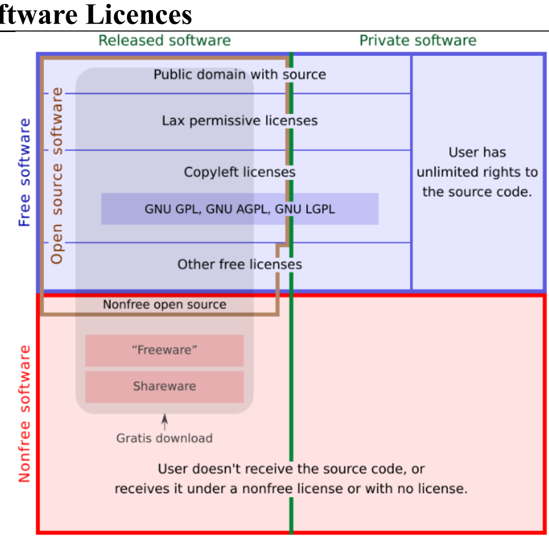
Maintenance: Ensuring software continuously satisfy users.

1.7. Contributing to Open Source

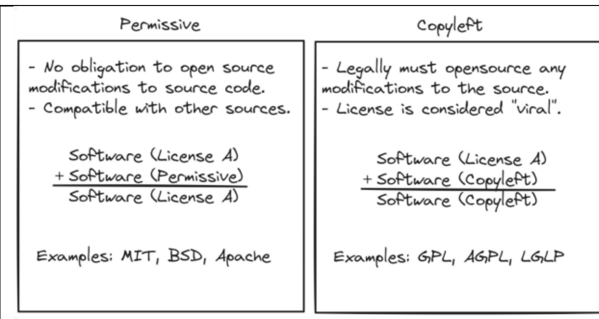
Open/Closed source software: whether source code of a software program is public.

- 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
- Fork repo. Why fork? So your changes can’t affect changes in parent repository. Then PR.

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

Appendix A – The CHAOS Report (1995)

https://www.csus.edu/indiv/r/rengstorffj/obe152-spring02/articles/standishchaos.pdf

- Investigation into the success of software development projects of large, medium, and small companies found that majority projects do not deliver on-time, on-budget, and on-scope.
- When a project fails, it's important to investigate, study, report, and share the cause.
- Delivering software in smaller time intervals early and often increases success rate.
- A successful project has the following characteristics.

	Criteria	Importance
1	User involvement	19
2	Executive management support	16
3	Clear statement of requirements	15
4	Proper planning	11
5	Realistic expectations	10
6	Smaller project milestones	9
7	Competent staff	8
8	Ownership	6
9	Clear vision & objectives	3
10	Hard-working, focused staff	3

Appendix B – Google Site Reliability Engineering Book

https://sre.google/sre-book/table-of-contents/

B.3. Embracing Risk

https://sre.google/sre-book/embracing-risk/

B.21. Handling Overload

https://sre.google/sre-book/handling-overload/
[CRITICAL TODO]

B.22. Addressing Cascading Features

https://sre.google/sre-book/addressing-cascading-failures/
[CRITICAL TODO]