

Engineering Large Software Systems Notes

Created: 2024-09-04

Updated: 2024-10-03

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


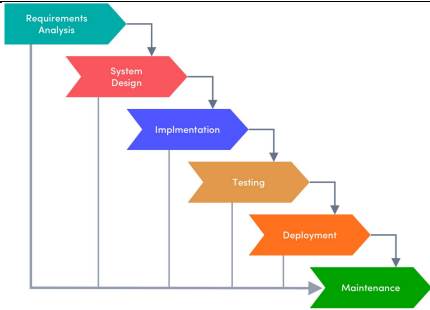
	pro	Con
rebase	Full history	Need to deal with merge conflict on each commit since commits are replayed over the new base.
merge	Merge commit once	History not as clean as rebase.

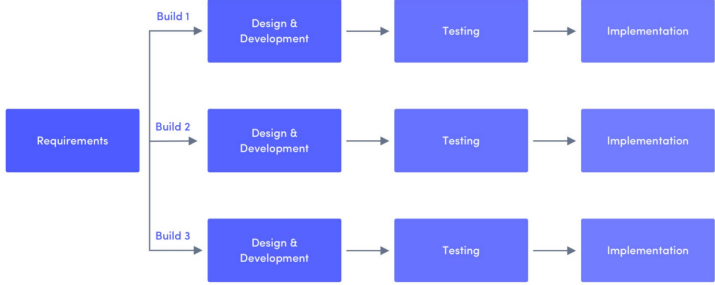
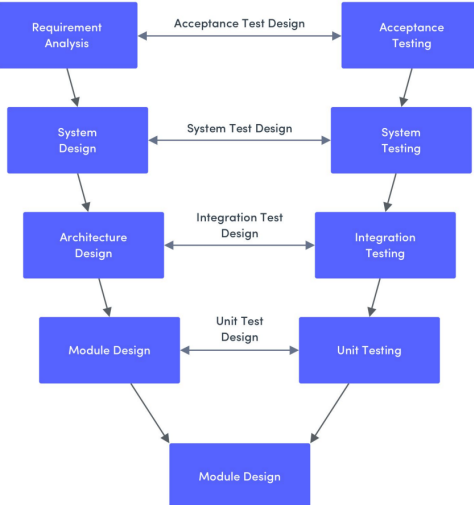
1. Large Software Systems

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
	

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.

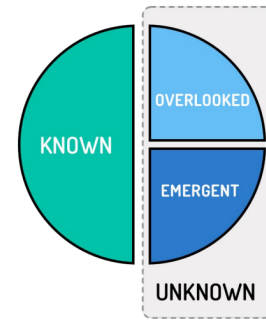


Figure: Relationship between types of system requirements.

Structural NFRs

- **Upgradability:** ease at which a system can be upgraded.
 - Problems: keeping track of versions, dependents of upgraded code need to change, backward compatibility (supporting older versions in a version upgrade).
 - Obstacles: API specification changes, DB schema changes (data model refactoring), data migrations (add/delete rows/columns).
 - Semantic versioning: Major.minor.patch where major is backwards incompatible, minor and patch are backwards compatible, minor is substantial new functionality, patch is bug fix.
- Deployability: deployment strategies, given that multiple replicas exist and a load balancer server controls traffic to the connected servers.
 - Recreate: 1. Package new version 2. Shutdown old version (show maintenance page). While offline perform data migration, DB schema updates, etc. 3. Spin up new version.
 - Ramped: 1. Package new version 2. Shutdown one old replica. 3. Spin up one new replica. 4. Repeat.
 - Canary: manually perform step 3 in ramped. Then observe before rolling out more releases. See Appendix C.16.
 - Blue-green (red-black): 1. Package new version. 2. Spin up all new replicas. 3. Load balancer redirect traffic to new replicas. 4. Shut down all old replicas.

	Pro	Con
Recreate	Simple, low cost Don't need to sync data since no data can be written when the service is unavailable. Fast to fix urgent bug fixes, security vulnerability.	Website down time.
Ramped	Automated, risk controlled. No downtime.	Slow to spin up shut down one by one Possible race conditions if load balancer not perfect; requests sent to a service that's shutting down Need to sync data written to new and old replicas. Will not be able to handle as many requests (solution: n+1 replicas).

Canary	Quick addressing of errors if they arise (find bugs quickly).	Even slower than ramped. Need to sync data written to new and old replicas.
Blue-green	Efficient. No downtime. Don't need to worry about different behaviour with different versions Easy to roll back to old version if there are bugs, advantage over ramped. Fast to fix urgent bug fixes, security vulnerability.	Costly to duplicate. Need to sync data written to new and old replicas.

- Compatibility
- **Customizability**: ease at which software behaviour changes with modification of application code by developers, devOps, support. Modifying code needs to go through deployment process.
 - Fast customizations can lead to decrease in maintainability of code.
 - Customizability is important for keeping up with high frequency of changes.
 - See Appendix D.
- **Configurability**: ease at which software behaviour changes without modifying application code. Anything an end-user can control is configuration.
 - Highly configurable application can skip lengthy deployment processes.
 - Use JSON to model the transition between pages (akin to Finite State Machines) where transitions are triggered by user actions (ex. button clicks). However, this is insufficient to achieve full configurability as only developers know the “language” and are still required to change app behaviour.
 - **TODO**: <https://blue.verto.health/changing-change-management/>
 - **Tradeoff Between Customizability and Configurability**
 - Increased configurability leads to decreased customizability since 1. More effort required to add features, 2. More regression testing on all possible configurations. 3. Converting customizations into configurations.
 - Increased customizability leads to decreased configurability. 1. Each path will need to map to a configuration. 2. Need heavy refactoring to remove hardcoded attributes without functionality change. (??)
- Security
 - **TODO**:
 - <https://owasp.org/www-project-top-ten/>
 - https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
 - Should be an implicit top priority and considered explicitly when driven by another NFR or evaluating risk of architecture (ex: code execution environment).
- **Legality**: regulations that govern what you can(not) do.
 - Ex. PHIPA requires patient authorization of information sharing.
- Time to market
 - Considerations: agility, deployability, testability.
- **Maintainability**: degree of ease for a developer to change behaviour of application.
 - Maintainability is inversely proportional to lines of code.
 - React is built to be un-opinionated which allows various ways of styling CSS.
 - Opinionated frameworks improve trainability, agility, and maintainability.
 - Improved maintainability by enforcing SOLID, and clear naming e.g., Factory, Strategy, Builder in code that implements these design patterns.

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.

1.2.1. DECIDING ON TRADE-OFFS / OPERATIONAL NFRs

- **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** = $\text{uptime} / (\text{uptime} + \text{downtime})$.
 - 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** = $\text{successful requests} / \text{total requests}$.
- **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).

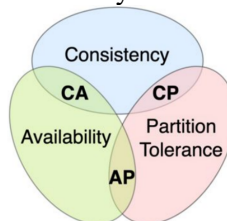


Figure: **CAP Theorem:** can only satisfy 2/3 of these characteristics.

- 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 (redis), memory usage).
- Size of input / output.
 - Decreasing size of output:
 - Reduces transport time.
 - Increases time to compress and decompress.
- Cost

1.2.2. MONOLITHIC ARCHITECTURE STYLES

- Server depends on DB.
- NFR considerations: cost (cheap to run, develop), availability (more expensive to ensure availability), testability, maintenance, performance, scalability/ elasticity/ deployability.
- Cons:
 - long startup time (a lot of code to initialize e.g., ORM, cache warming), upgrade requires restarting entire monolith (more downtime), hard to scale up specific functionality
 - Server code is ungoverned, and modules are more likely to be tightly coupled.
 - Less maintainable with more lines of code and availability/scalability increases only with increase in cost.
- **Design payoff line**: Trade-off between time-to-market and design quality.
 - No design: more functionality in less time with small codebase
 - Design: more functionality in less time as codebase becomes larger

Layered Architecture Style

- Enforces placing the right code in the right layer. Layers are independent and dependencies are downwards only.
- E.g., Presentation Layer → Business Layer / Persistence (ORM) Layer → DB
- Code structure:
 - Enhanced MVC: view → controller → service → model → DB
 - Frontend: components, pages. Backend: models, services, controllers.

Pipeline Architecture Style

- Used in data processing.
- Producer (e.g., Request) → filter → ... → filter → tester → consumer (e.g., DB, display to UI).
E.g., Apache Beam, LangChain.
- Extract data from DB, Transform data, Load into target DB usually analytical (ETL)
- More modular than layered since filters are reusable and isolated features make it easier to test.

Microkernel Architecture Style

[TODO]: [4. Microkernel Architecture - Software Architecture Patterns, 2nd Edition \[Book\] \(oreilly.com\)](#)

1.2.3. DISTRIBUTED ARCHITECTURE STYLES

- Multiple deployment units (services) connected through remote access protocols.
- Pros: better scalability, elasticity, employability, fault tolerance over monolithic.
- Cons:
 - much greater latency communicating cross-service compared to between components.
 - Possible no response after request, service cannot be reached.
 - Each service needs to be secured with its own auth policy.
 - Doesn't necessarily loosen coupling.
- Synchronous: request → logic → response.
- Asynchronous: request → acknowledgement. Completed.
- Message queues: async service-to-service comms, used for batching and buffering.
- Repot Procedure Calls: enables seamless remote communication between services.

Service-Based Architecture Style

- Multiple mini-monoliths potentially sharing a DB. UI → API Gateway → many services each connects to DB.
- Characteristics: No interservice communication, domain separated services.
- API Gateway: addresses NFR considerations: security, authentication, auditability, metrics.
- Each individual service can be scaled up.
- No interservice communication brings loose coupling but requires shared library for sharing logic. This means upgrade shared lib then upgrade every monolith.
- 4/5 modularity
- 2/5 elasticity: replicating mini-monolith to add services increasing costs

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

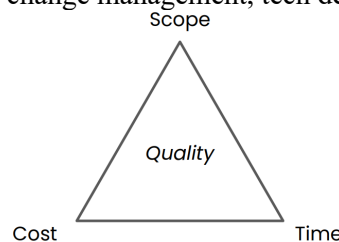


Figure: interdependence of scope, cost, and time on software quality.

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

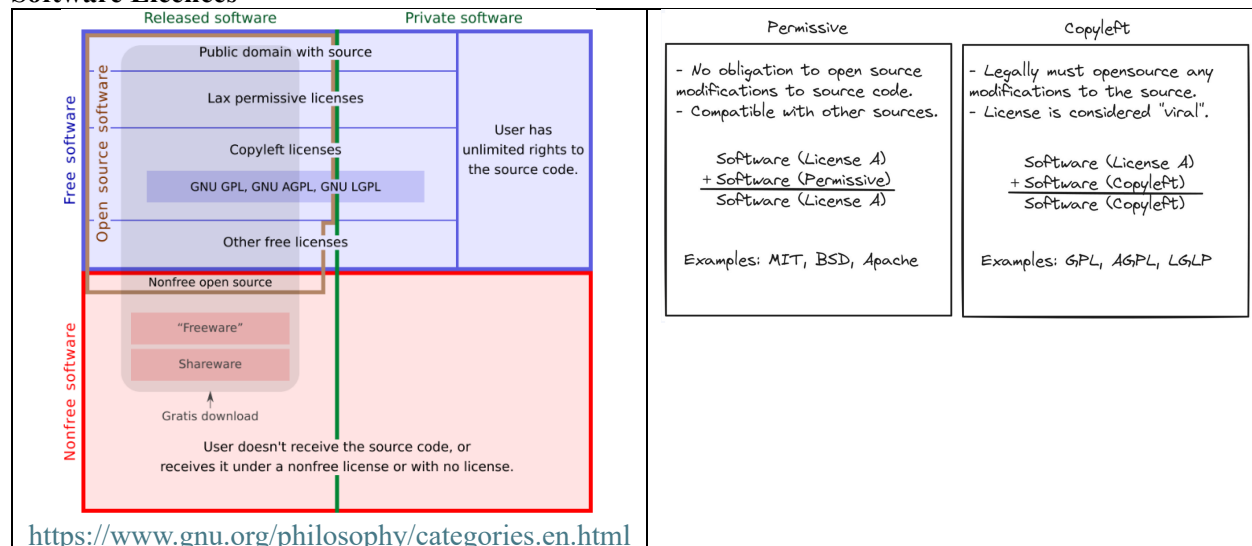
Maintenance: Ensuring software continuously satisfy users.

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



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/>

[TODO]

B.21. Handling Overload

<https://sre.google/sre-book/handling-overload/>

- The responsibilities of load balancer policies are to distribute work to prevent system overload.
- Overload is inevitable if usage grows. In handling these situations, “redirect when possible, serve degraded results when necessary, and handle resource errors transparently when all else fails.”
- The **queries per second** metric cannot accurately capture resource requirement when measuring capacity. Instead, measure capacity in terms of available resources, e.g., available CPUs, memory. Consequently, cost of a request can be measured in normalized CPU time.
- **Global overload** occurs when all global resources are full and when it does it's important to deliver errors to only affected customers (the app making the request). This is done by allocating certain CPU consumption times to each app: gmail, calendar, android etc.
- Rejecting requests takes up resources and in certain cases more than performing the request. **Adaptive throttling** works on the client side to restrict the requests sent if within the last two minutes the number of requests exceeds K multiples of accepted requests, $acc < \frac{1}{K} req$.
- Requests to backend are associated with 4 levels of **criticality**: CRITICAL_PLUS (serious user-visible impact), CRITICAL (user-visible impact), SHEDDABLE_PLUS (partial unavailability expected), SHEDDABLE (frequent partial unavailability expected).

- In **task-level overload protection**, the executor load average is calculated by finding number of active (running/ready) threads in the process. Spikes in requests get smoothed out while sustained increase starts getting rejected. Can use memory pressure instead of CPU usage/CPU reserved.

[CRITICAL] TODO: Handling Overload Errors]

B.22. Addressing Cascading Features

<https://sre.google/sre-book/addressing-cascading-failures/>

[CRITICAL TODO]

Appendix C – Google Site Reliability Engineering Workbook

<https://sre.google/workbook/table-of-contents/>

C.16. Canarying Releases

<https://sre.google/sre-book/handling-overload/>

[CRITICAL TODO]

Appendix D – Enabling Contained Customization with FSM & Server-Driven UI

<https://blue.verto.health/enabling-contained-customization-with-fsm-server-driven-ui/>

- Verto Health needed a maintainable way to scale their code for growing number of clients (health care institutions) to support vaccination appointment scheduling during the covid pandemic.
- Problem: each client needed a different logic flow for appointment scheduling. Creating various if-statements for each client becomes inefficient.
 - Solution: using XState to define states and transitions between pages / functions.
- Problem: question page between different clients is highly similar but different.
 - Solution: server-driven UI that allows questionnaires to be defined with configuration files.
- By combining FSM and Server-Driven UI, developers no longer needs to be involved in the process of creating and maintaining the flow for appointment scheduling for various clients. The responsibility is transferred to IT Support.