Klausurvorbereitung Advanced Software Engineering

Christoph Siller

16. März 2022

Inhaltsverzeichnis

1 Build for ten years or more

1.1 What is advanced Software Engineering?

Dependable software (that always works) which has an **extended Lifecycle** (runs decades, not years) and is **large, complex and integrated**.

1.2 Attributes, means and threads of Dependability?

Dependability is an integrative concept with the following **attributes**:

- **Maintainability** (can be repaired and modified)
- **Availability** (always ready)
- **Reliability** (always correct)
- **Confidentiality** (no unauthorized disclosure of information)
- **Integrity** (no invalid system states)
- **Safety** (no catastrophic consequences for the user)

and the following **means:**

- **Fault forecasting:** Qualitative (identify, classify, rank) and Quantitative (probability model)
- **Fault prevention:** by quality control and software design (structured programming, information hiding, modularization)
- **Fault tolerance:** by error detection and recovery, error handling (roll-back /roll-forward), redundancy and fault isolation
- **Fault removal:** by static/dynamic verification, diagnosis and correction. Fault injection to test error handling and corrective/preventive maintenance

against the following **threads:**

- **Errors:** Discrepancy between a computed/observed/measured value and the correct value (e.g. null value when not valid)
- **Faults:** Abnormal condition that can cause failure (e.g. NPE)
- **Failures:** Termination of the ability of an element to perform a function as required (e.g. crash of a service)

1.3 What are possible flaws in Software Safety?

- **Management**
  - Diffusion of Responsibility and Authority
  - Limited Communication Channels and Poor Information Flow
- **Technical**
  - Inadequate System and Software Engineering (Poor Specification, Unnecessary Complexity or Functionality, Software Reuse or changes, Violation of Basic Safety Practices)
  - Poor Review
  - Poor System Safety Engineering
  - Poor Tests and Simulation
  - Bad Human Factors Design

1.4 Reasons, implications and prevention of Software Aging

Reasons for Software Aging:

- Lack of movement: Failure to modify the product to meet changing needs
- Ignorant surgery: Result of the changes that are made

Problems during lifecycle:

- Inability to keep up (growth)
- Reduced performance (poor design)
- Decreasing reliability (error injection)

Preventive measures

- Design and plan for change
- Documentation and Reviews
- Restructuring including partial replacement (amputation)
- Plan for retirement and replacement

1.5 Component VS. Service (after Fowler)

A **component** is a glob of software that is intended to be used, without change, by an application that is out of the control of the writer(s) of the component. 'Without change' means that the using application does not change the source code of the component, although they may alter the component's behavior.

A **Service** is used by foreign application similar to a component.
The main difference is that a component is used *locally* (e.g. a JAR file, assembly file) and a service is used *remotely* through some interface, either synchronous or asynchronous.

1.6 Forms of modularization

- **Build Time** (e.g. Maven)
  - Multiple JARs possible
  - Single Bundle (Container)
  - Update requires complete redeploy
  - Easy operation
  - Easy and fast intermodule communication

- **Runtime (single or multi VM)** (e.g. OSGI)
  - Multiple JARs required
  - Multiple Bundles (Container)
  - Update requires partial redeploy
  - Highly complex operation
  - *Single VM:* more complex but faster communication
    Multi VM: possibliy slow communication

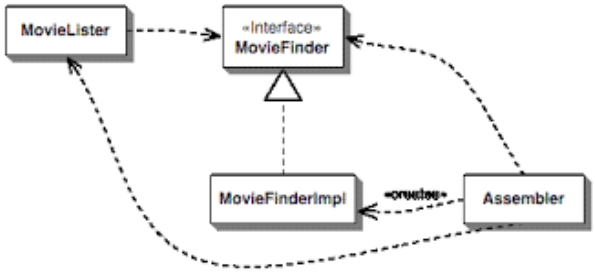
1.7 Dependency Injection

Dependency Injection is a pattern, where the gluing of objects is separated from the implementation. Therefore all implementation is against the API. There is a central definition and container that creates and binds objects together.

DI supports code reuse and independently testing classes. DI supports different bindings for different environments. DI supports lazy creation of objects.

A DI framework (often based on AOP) provides the runtime services for DI (e.g. Spring Framework).

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out. Dependencies can be injected into objects by many means (such as constructor injection or setter injection). One can even use specialized dependency injection frameworks (e.g. Spring) to do that, but they certainly aren't required. You don't need those frameworks to have dependency injection. Instantiating and passing objects (dependencies) explicitly is just as good an injection as injection by framework.



1.8 Aspect Oriented Programming (AOP)

The key unit of modularity in AOP is the aspect (as compared to the class in OOP).

Cross-cutting concerns are functions that span multiple points of an application and are conceptually separated from the applications business logic. (e.g. logging, declarative, transactions, security, and caching)

AOP helps you decouple cross-cutting concerns from the objects that they affect.

Aspects are „woven in“ at compile time or runtime.

2 Release Your stuff 3 Times a Day

2.1 Dependency Management - how to

- Declare which libraries you are using
- Declare which version of a library you are using
- Declare in which context you are using the library (test vs. production)
- Declare where these libraries are coming from
- Have these libraries declare, which libraries they are using
- Automatically retrieve all required libraries from a/your repository

Tools:

- **Maven:** manage dependencies, build project, run tests, release, execute plugins (old but common)
- **Gradle:** DSL instead of XML, official Android build tool, more flexible scripting
- **Apache Ivy:** Pure dependency management

2.2 Problems with keeping dependencies in a SCM (Source Code Manager)

- Difficult to (manually) find new libraries and updated versions
- Loose trace to source (e.g. download page)
- Loose version information (unless included in filename or package)
- No information about transitive dependencies
- SCM not built for versioning binaries (no diff, bad handling of binary files, high resource usage)
- DSCM (e.g. Git) especially bad (by design) at working with large binaries

2.3 What is a Repository Management Tool? examples?

Manage all used (third party) dependencies and repositories, even the ones that are not readily available in public repositories.
Proxy and cache remote repositories to achieve faster build times, easy traceability, fault tolerance and enhanced security.
Also manage all artifacts created by your project (binaries, source, documentation, configuration) and archive past releases.

Tools:

- jFrog Artifactory
- Sonatype Nexus
- Apache archiva

2.4 Semantic Versioning

Is a set of best practice rules for versioning of software with a public API.
Pattern: X.Y.Z

- **X - Major Version**: must be incremented if any backwards incompatiblechanges are introduced to the public API
- **Y - Minor Version**: must be incremented if new, backwards compatiblefunctionality is introduced to the public API
- **Z - Patch:** must be incremented if only backwards compatible bug fixesare introduced

2.5 Tasks of Build Management and Automation

1. Retrieve Dependencies
2. Prepare Resources
3. Compile source code to binary format
4. Package binaries
5. Execute automated test cases
6. Execute static code analysis and reporting
7. Generate documentation
8. Run your application locally

9. Deploy your application
10. Release and publish your artifacts

Build Tools:

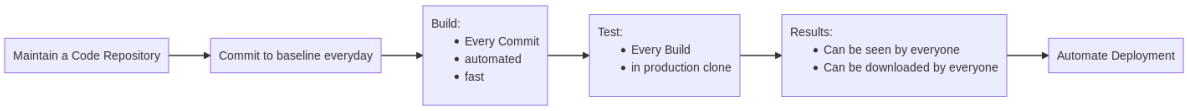
- make
- Apache Ant/NAnt
- Apache Maven
- Gradle
- MSBuild
- Rake

2.6 Continuous Integration (CI) principles by Fowler (+tools)

Constantly merge development work with mainline. Build and test automatically.
CI Tools: *Atlassian Bamboo*, Hudson/Jenkins, Apache Continuum, CruiseControl, GitlabCI,...

CI Principles by M.Fowler:

1. Maintain a code repository
2. Automate the build (maven, make, gradle)
3. Make the build self-testing (execute automated tests while building)
4. Everyone commits to the baseline every day (work in small batches)
5. Every commit (to baseline) should be built
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy to get the latest deliverables
9. Everyone can see the results of the latest build
10. Automate deployment



Continuous Delivery VS Continuous Deployment: only *deliver* your code to staging environment or also *deploy* it on production.

2.7 What are build pipelines?

Logically structures a CI build into a series of steps. Information about CI configuration is stored alongside the code (e.g. .gitlab-ci.yml)

3 Challenges to solve for every project

3.1 Types of Transaction Managers and how to choose one

Types of Transaction Management depend on who is responsible for the Transaction:

- **Local Transaction Model:** The underlying database handles the transaction (e.g. via auto commit)
- **Programmatic Transaction Model:** The developer handles the transaction through code
- **Declarative Transaction Model aka Container Managed Transactions (CMT):** Developers only specify the behavior, the application container handles the transaction

How to choose a type:

- If running inside a suitable container, declarative transactions are usually the safest bet
- Make sure you understand managed persistence context
- Completely managing transactions manually results in a lot of boiler-plate code and is error prone
- CMT makes automating tests hard(er) – an embedded container is necessary to provide a suitable environment

3.2 Difference between Logging and Auditing

Logging is ephemeral and usually used for debugging or informational purposes, e.g. to understand how the system works or in what state the system is. The output is not suitable for automated processing.

Auditing is persistent and audit logs are kept also for compliance reasons. An audit log enables to see what actions users performed.

Logging (e.g. Log4J, Logback, meta: slf4j)

- Technical, text based output primarily used for detecting and debugging problems
- Level of detail can usually be configured at runtime
- Output is not (easily) understandable for regular users of the system
- Output is not suitable for automated processing
- Short term retention (months)

Auditing (very domain specific, no frameworks)

- Domain specific, fine grained and structured output for tracing user activity
- Requirements are specified by legal and/or company specific policies
- Used by (limited) end user group (e.g. internal revision)
- Frequently coupled with long term retention requirements (e.g. 10+ years for regular businesses, 30+ for medical)

3.3 Good Log output

- Limit the amount of log levels used (usually DEBUG, INFO, WARN and ERROR are sufficient)
- Establish and enforce clear rules when to use which log category
- Automatically adjusting the log level according to the current situations can help to make problems visible earlier
- Contextual information helps you to trace the flow of execution
- Always provide a reference ("user created" vs. "user (id=1234) created")
- Make sure to tailor the log output after gathering experience in practice

3.4 How to correctly configure logging (and auditing)?

- Select a logging library with a concise API and good performance
- Configure the logging library to give you just enough information
- Define clear and simple rules for all developers to follow
- Review the quality of your log output on a regular basis
- Ensure the logs are easily and quickly accessible
- Auditing, if necessary, is a regular functional requirement
- Make sure auditing does not affect your performance

3.5 Authentication VS Authorization (+ examples)

Authentication is the process of verifying who a user is, while authorization is the process of verifying what they have access to.

Authentication examples

- **Username/Password** - easy to implement and use but insecure
- **Certificate based (client authentication)** - complex to roll out and manage but high security
- **Smart card, biometric** - needs client side support
- **Token based / Single Sign On (SSO)** - delegate authentication, use central identity (OAUTH, SAML)

Authorization examples

- **Role based access control (RBAC)** - for resource based systems. Simple and easy to grasp.
- **Permission based** - simple action based, complex expressions. Fine grained and more flexible.
- **Access control lists (ACL)** - fine grained, hard to manage, significant performance implications
- **Rule based** - for complicated and frequently changing business requirements

3.6 How to choose an appropriate access control mechanism?

- Reuse existing infrastructure or frameworks over building your own
- Decouple authentication, authorization and identity management
- Keep your business code clean of provider specific dependencies
- Make sure to adhere to organizational requirements
- Consider performance implications when using more complex authorization methods

3.7 Exception Handling Anti-Patterns

- Log an exception and rethrow it: Do either one of them.
- Catch "Exception" (i.e. all exceptions, not one in particular): It's like a fisher's net - you do not know what you will catch.
- Destructive wrapping: Always pass the causing exception.
- Catch and ignore: This one will come back to bite YOU.
- Throw from within finally: Will swallow any other exception.

3.8 Exception Translation Pattern

Do not expose "lower level" exceptions to upper layers of code to avoid "API bleeding". If exception cannot be handled at the current stage, wrap it in a module specific exception.

```
try {
    doSomething();
} catch (LowerLevelException e) {
    throw new MyBusinessException("message", e);
}
```

3.9 How to consistently manage user and program errors in your system?

- Do not use exceptions to direct regular program flow
- A good exception (handling) strategy will make your code usable and maintainable
- Consistency is key for maintainability and readability
- Do not overpower your end user with incomprehensible information

3.10 Localization (L10n) VS Internationalization (I18n)

Internationalization is done once: Preparation of a product to be used in the global market.

Localization is done for every locale: Specific adoptions to launch a product in a specific locale.

Examples: Language, Charset, Sorting, Names and Titles, measurement units, Currency, Date-Time, Numberformat,...

3.11 How to prepare you product for a global audience?

- Consider Internationalization right from the beginning, especially character encoding and Locale/Timezonesettings
- Know your target market to avoid unnecessary overhead
- I18n is not only translatable text
- Even if initially only one language is the target, investing in Internationalization can pre-empt changing requirements
- Make use of tools and frameworks

4 Microservices

4.1 Concept/Advantages/Disadvantages/Challenges of Microservices

A microservice architecture is a method of developing software applications as a suite of small, independently deployable, modular services in which each service runs a unique process and communicates through a well-defined, light-weight mechanism to serve a business goal.

Advantages:

- Distributed system
- Technical Heterogeneity - choose the right technology for the job
- Quickly adapt to new technologies
- Fault tolerance
- Scalability
- Deployment
- Replaceable
- Testing
- Clear separation of ownership

Disadvantages:

- Distributed system
- Technical Heterogeneity
- Deployment
- Monitoring
- Testing
- Transactions
- Reporting

Challenges:

- Are MicroServices really independent of each other?
- How about versioning?
- How to detect if a feature is unused?

in comparison a Monolith:

- One executable / deployable
- Hard(er) to use different programming languages
- Often complex setup of environment
- Can reasonably be deployed by hand

5 From Prototype to Production

5.1 Idea behind the DevOps Movement

The DevOps movement is a philosophical idea where traditional boundaries between development and operations are removed. It is more about breaking down walls than classic "who does what". It is not about who does it, but when and how it is done. It is about considering operations from the beginning. It is about knowing how "the other side"works. It is about facilitating communication!

5.2 Configuration Management (CM)

Configuration Management is a management process for establishing and maintaining consistency of a product's performance, its functional and physical attributes, with its requirements, design and operational information, throughout its life.

Configuration that need to be managed:

- Build configuration (e.g. state of code, how to build, dependencies, documentation)
- Product configuration
- Application server/database configuration (using YAML)
- OS configuration
- System configuration (*infrastructure as code* like Chef or Puppet)

5.3 Clustering VS Load balancing - variants of clustering and load balancing

Clustering means you run a program on several machines (nodes). One reason why you want to do this is: Load balancing. If you have too much load/work to do for a single machine you can use a cluster of machines instead. A load balancer then can distribute the load over the nodes in the cluster.

Different clustering modes, different implications

- Application level clustering: full/delta session replication
- Database level clustering (e.g. Oracle RAC)

Load balancing

- Sticky session
- Round robin
- Active/passive
- Hardware vs. software

Trade-off between load distribution and fault tolerance
Always perform fail-over tests on your setup (under load)

5.4 Types of Caching in a cluster setup?

A clustered setup has strong implications on your caching strategies. There are two types of Caching:

- **In-Process Caching**
  - One cache per-process (higher overall memory usage)

  - Possibilities for inconsistencies between individual caches
  - Be extra careful with cache size in on-heap scenarios

- **Distributed Caching**
  - Slower due to additional overhead (network latency, object serialization)
  - More complex to operate
  - Scales much better
  - No risk of taking down the main application with OutOfMem

5.5 Tracing VS Sampling in Performance profiling

- **Tracing**
  - Usually done through byte code instrumentation
  - Delivers invocation counts
  - Can significantly influence runtime performance
  - Not suitable for production environments

- **Sampling**
  - Periodically queries stacks of running threads to estimate the slowest parts of the code.
  - No invocation counts
  - Negligible performance impact

6 Lost in Complexity

6.1 What is a System?

1. A system is a set of things-people, cells, molecules, or whatever- interconnected in such a way that they produce their own pattern of behavior over time.

2. The system, to a large extent, causes its own behavior! An outside event may unleash that behavior, but the same outside event applied to a different system is likely to produce a different result

3. a system has a defined border that defines what is part of the system and what is not. It interacts with it's environment or other systems.

6.2 Characteristics and Behavior of Complex Systems

Characteristics

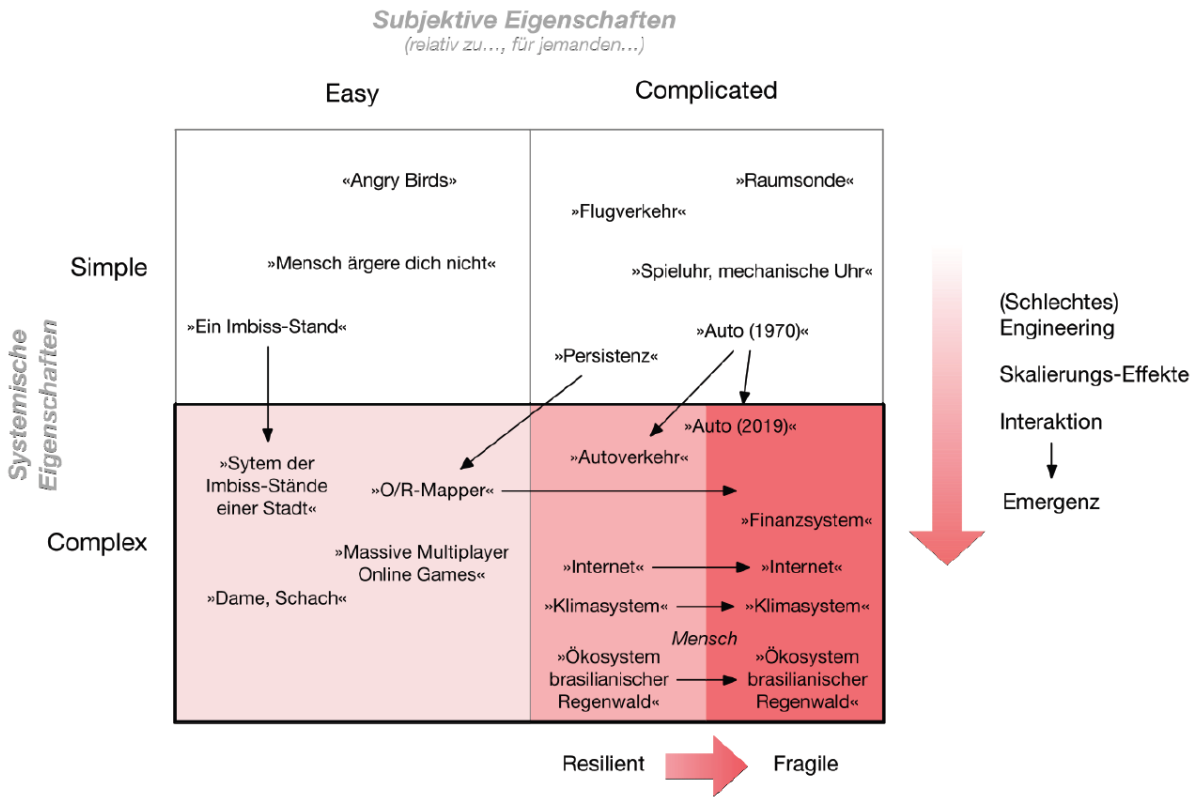
- Interdependent (Variables depend on their past, other variables and past of other variables)
- Complex interactions between many parts / feedback loops
- Interaction can change the part(s)
- dynamic, network connections
- not predictable by *formula*
- Often: systems of systems (ecology)

Behavior

- Follow attractors (self-healing, randomness, recovery between stressors, tipping points)
- Cause-Effect analysis makes not much sense
- *Defect* components cannot be changed easily
- Show unexpected behavior
- self-organising systems are nearly always complex systems; not all complex systems are self-organising

6.3 Easy VS Simple / Complex VS Complicated

Simple: one taks, one role, one concept, one dimension (opposite is complex)
Easy: Near at hand, familiar, near to our capabilities, relative (easy for whom?) (Opposite is Complicated)



Remember examples: *Sausage joint VS Network of Sausage joints, Car VS Traffic*

6.4 Wicked Problems

- **There is no definitive formulation of a wicked problem**
  hard to decide what actually is the problem, hard to test if it is solved
- **No simple stopping rule, no true/false solution**
  Stop with improvement when it is *good enough* or resources run out
- **Every wicked problem is unique**
  One-shot operation, no trial and error possible
- **A wicked problem extends laterally and vertically**
  can be a symptom of another wicked problem (vertical) or consist of many wicked problems (horizontal). Solution might cause a new wicked problem. Don't get stuck in analyses paralysis!

6.5 How to deal with complexity in Software?

- Agile development (long term planning never works out)
- Chaos Engineering, Chaos Monkey (Experiment with System, produce unwanted behavior in production e.g. unsync clocs, kill services, inject latency)

- Canary roll out (roll out to small subset of users to find errors)

7 Free and Open Source Software (FOSS)

7.1 What is the benefit if you hit a bug in FOSS?

You don't need workarounds in your own code, instead you can fix bugs directly in the OSS libraries you use.

7.2 How to monetize FOSS - business models

- Adding commercially available value on top of a base OSS offering (e.g. RedHat JBoss AS, IBM WebSphere Liberty)
- Professional Training
- Embedding OSS into hardware (e.g. most routers use Linux, iptables, Android)
- Service Contracts (i.e. 'insurance' for your project)
- Sharing the costs - saving money!
- Project Consulting (i.e. Help other companies to save money by using OSS.)

things to avoid:

- Don't try to sell the same product you give away for free
- Respect the freedom! (i.e. Don't force/restrict others to do something only because you need it. Respect the community
- Don't rely (only) on a payroll (i.e. don't push a project into one corner just because a customer likes it that way)
- OSS project planing is different than company projects
- Spread the influence across different companies

7.3 Contributor License Agreement (CLA)

A Contributor License Agreement defines the terms under which intellectual property has been contributed to a company/project, typically software under an open source license. There are symmetric CLAs where every contributor has the same rights, and asymmetric CLAs that grant additional rights to a subset of developers. Reasons to use a CLA are:

- Make the Contributor aware of the legal impact
- Grant additional rights beyond the License

7.4 What is a License?

A software licence is a conglomerate of conditions under which a Licensee gets to use code. It is consensual, and although it is juristically viewed not a contract, it is very close. It matters because it is a form of securing for your own code.

7.5 License flavors

- **Commercial Licenses:** bloated with exits and safety valves for the vendor. Company is allowed to do anything.
- **MIT License:** code provided as is. Rights to use, copy, modify, merge, publish, ... Do what you want and leave me alone.
- **BSD License:** Allows to copy, change and distribute source and binaries. Requires to keep the original Copyright Header.
- **GPLv2:** Strong copy-left. Your give up all rights. All changes and linking to GPL software are also under GPL. Viral behavior!
- **LGPL:** Like GPL but you can use it as library without publishing your source. Use it but don't touch it.
- **ALv2:** Apache License. Requires a NOTICE file, includes a patent grant and can be sub-licensed (add something and publish that part under new license).

7.6 Patent Grant

Currently software patents are allowed in the US but not the EU. A patent grant makes sure you own the patent if there will ever be a software patent. Licenses that include a patent grant are **ALv2, GPLv3, MPL (Mozilla Public License)**.

8 Agile Software Development

8.1 The Agile Manifesto

1. **Individuals and interactions** ... over processes and tools.
2. **Working software business results** ... over comprehensive documentation.
3. **Customer collaboration** ... over contract negotiation.
4. **Responding to chang**e ... over following a plan.

8.2 Extreme Programming Praktiken

Communication/Collaboration/Architecure

- Planning Game
- Metaphor
- Simple Design

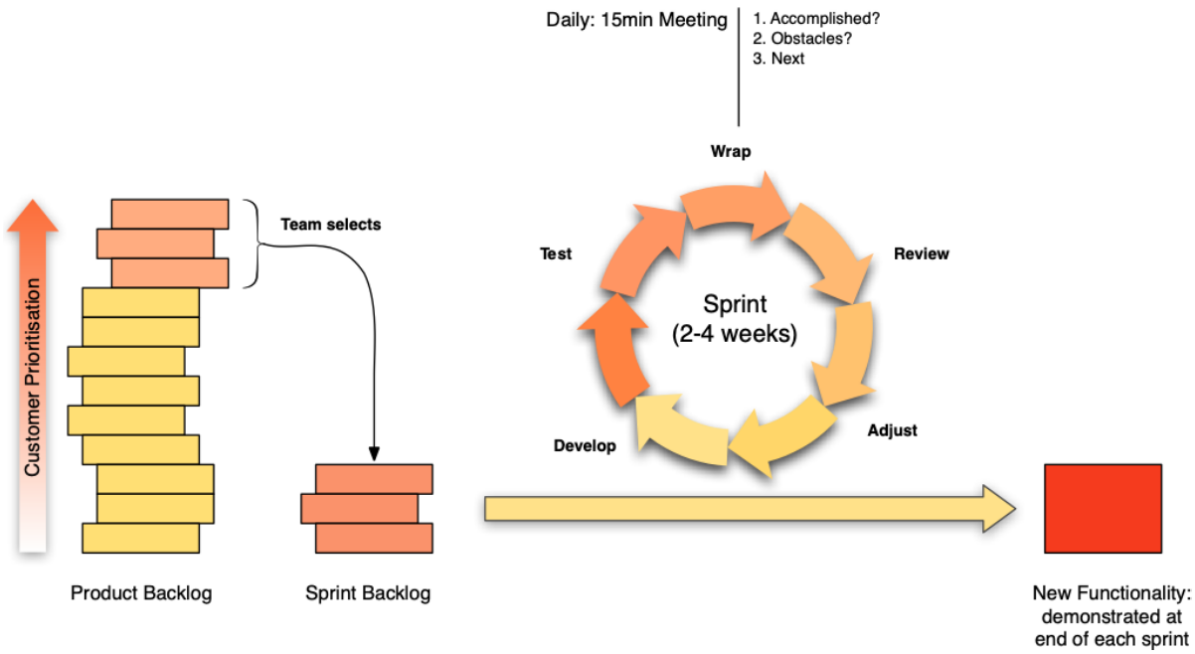
Process

- Small Releases
- Pair Programming
- Collective Code Ownership
- 40-hrs Week
- On-Site Customer

Technical

- Coding Standards
- Testing
- Continuous Integration
- Refactoring

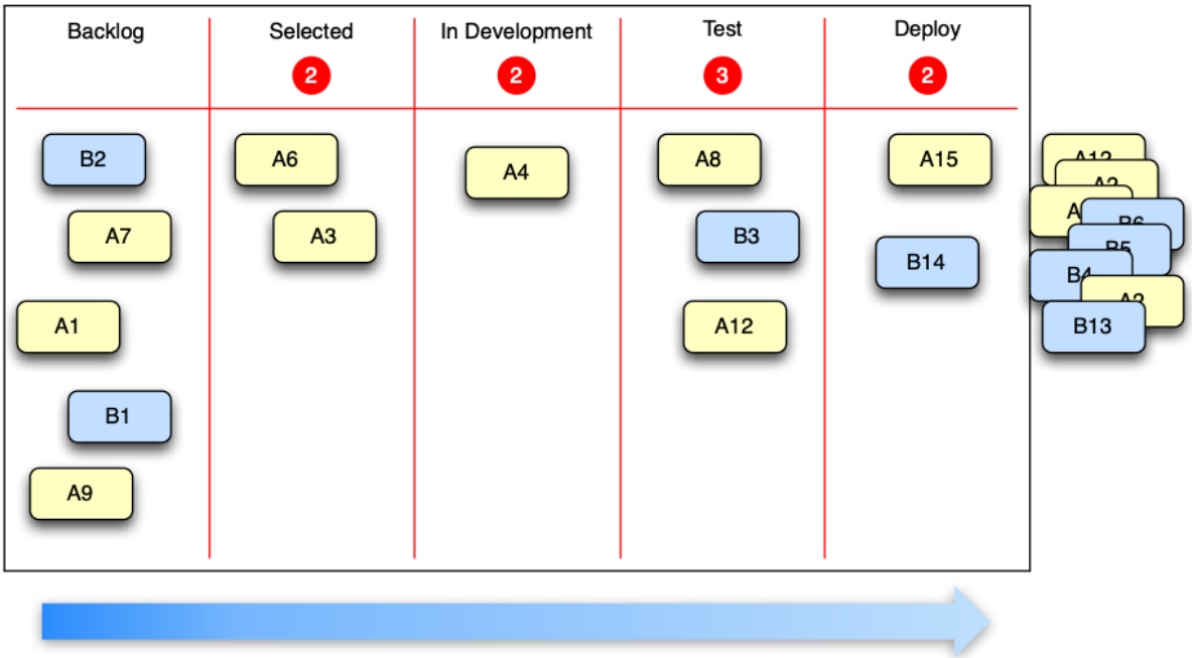
8.3 SCRUM



8.4 Scrum poker

Scrum poker, is a consensus-based, gamified technique for estimating, mostly used to estimate effort or relative size of development goals in software development. In planning poker, members of the group make estimates by playing numbered cards face-down to the table, instead of speaking them aloud. The cards are revealed, and the estimates are then discussed. By hiding the figures in this way, the group can avoid the cognitive bias of anchoring, where the first number spoken aloud sets a precedent for subsequent estimates.

8.5 Kanban



Basics

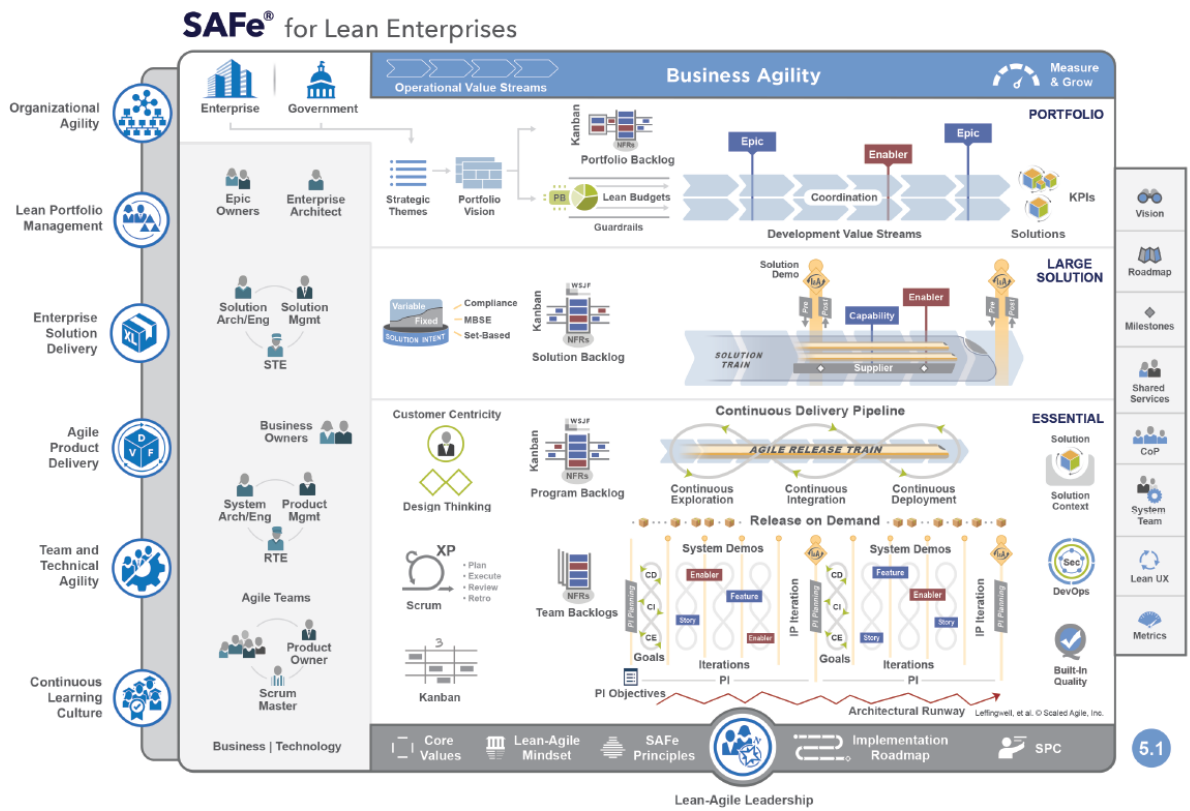
- Kanban is flow oriented and does not force a time frame like Scrum does with sprints.
- Kanban does not require specific roles like Scrum with Project owner, Scrum Master etc.
- Prioritization and changes in a project can be done on-demand in Kanban, Scrum discourages significantly changing a sprint.

Kanban VS Scrum

- **Kanban**: No different roles.
  Scrum: each team member has a predefined role
- **Kanban**: Continuous delivery.
  Scrum: deliverables are defined by a sprint
- **Kanban**: best for projects with widely varying priorities.
  Scrum: best for teams with stable priorities that don't change much over time.

8.6 SAFE Framework

Is a scaled SCRUM version.



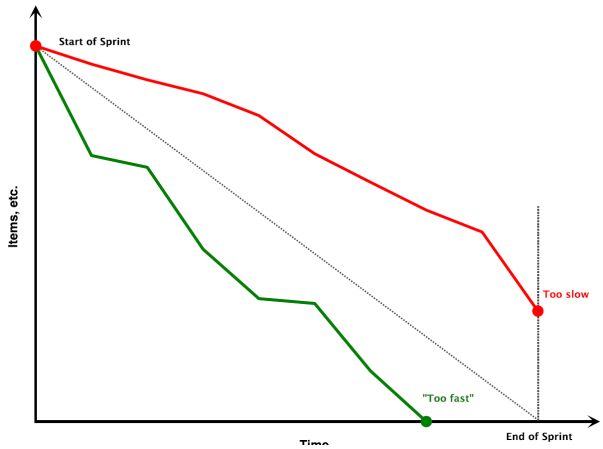
8.7 User Stories

User Stories define *as WHO I want WHAT so that WHY*.
They also provide a *acceptance Criteria* and should have the following properties (INVEST):

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

8.8 Burn Down Chart

Chart that provides a visualization of progress. Y-axis holds features (story points), X-axis represents time.



9 Collective Intelligence Systems

9.1 What is Collective Intelligence?

Ancient Phenomenon: Group intelligence that emerges from collaboration, collective action, and competition of many individuals.

Groups of individuals doing things collectively that seem intelligent.

Examples: Drone Swarms, Smart Mobility, Social Networks, Co-Creation Platforms

9.2 What are Collective Intelligence Systems (CIS)? Concepts and Concerns?

Harness the collective intelligence of connected groups of people by providing a web-based environment to share, distribute and retrieve topic-specific information.

Foundational Concepts

- **Coordination Models for Swarms**
  Flexible, adaptive and collective behavior found in swarms, flocks, herds and other groups of social animals. (*Swarm formation, Stigmergy*)
- **Self-Adaptation**
  Deals with uncertainties and business continuity.
  Feedback loops, self-organization and self-adaption approaches.
- **Socio-technical Systems**
  Combination of social aspects of people and society and technical aspects of organizational structure and processes.

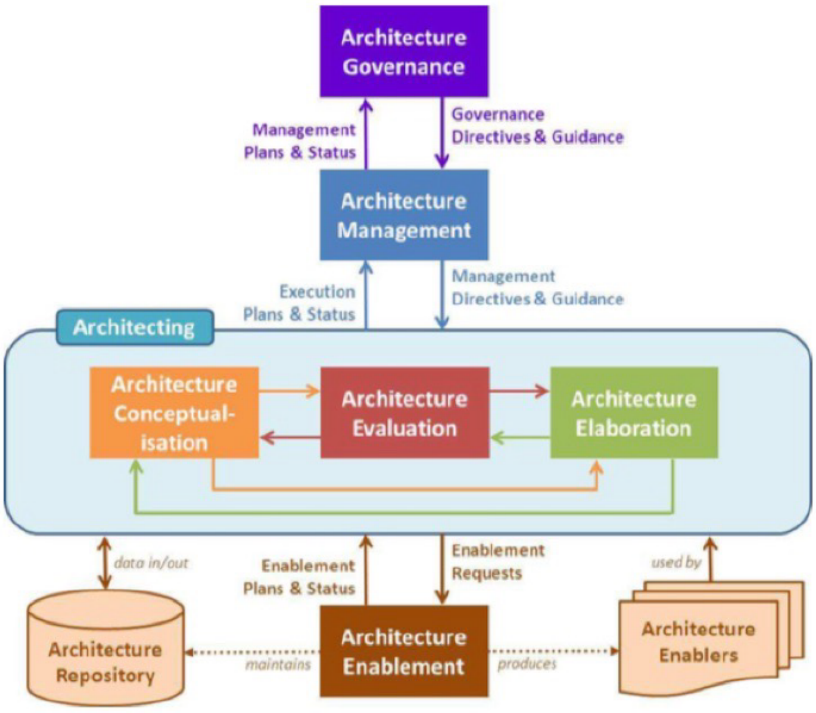
Concerns

- Environment-mediated coordination and indirect communication between actors with feedback loop.
- Information Aggregation. Bottom-up sharing of content through actors.
- Knowledge Dissemination. Distributing content among actors.
- Perpetual Feedback Loop. Continuous flow of user contributions.

9.3 Types of CIS

- **Social network services** (Facebook, Twitter, LinkedIn, Instagram)
- **Media / Content Sharing** (YouTube, Soundcloud, Flickr, Slideshare)
- **Knowledge Creation** (Wikipedia, Stack Overflow, Yelp)

9.4 Architecture Process



- **Architecture Governance**
  Establish standards and policies related to one or more architectures of interest and their development, and to monitor and facilitate the alignment of the architecture(s) to stakeholder concerns, policies and standards, including organizational and environmental constraints.

- **Architecture Management**
  Ensure execution of directives for development of the architectures, to ensure that the development runs according to these directives, to the expected timetables, to the assigned budgets, and that the architecture satisfies its objectives.

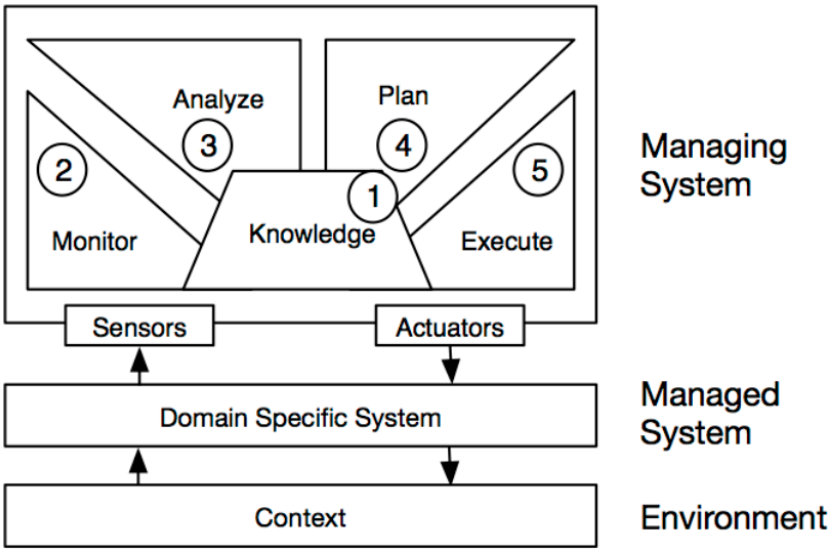
- **Architecture Conceptualization**
  Generate architecture alternatives, to select one or more alternatives that address stakeholder concerns and meet relevant requirements, and to express them in a set of consistent views.

- **Architecture Evaluation**
  Determine the degree to which the architecture meets architecture objectives and addresses stakeholder concerns.

- **Architecture Elaboration**
  Create one or more architecture descriptions in a form that uses established notations and languages and captures this in a set of consistent views and models.

9.5 MAPE (-K) Model

Is a design model for self-organized systems.



1. **Knowledge**
   Data and information shared among the monitor, analyse, plan and execute stages. Examples: logs, rules/policies, metrics, topologies.

2. **Monitor**
   Aggregates, cleans and filters data from different types of sensors (virtual, physical, humans) from the Managed System.

3. **Analyze**
   Performs data analytics and subsequent reasoning based on the data provided by the monitoring stage and the existing knowledge data.

4. **Plan**
   Relates the data from the Analyze stage w.r.t. to defined objectives and goals. Selects/creates commands/workflows that would manipulate the Managed System in order to adapt towards desired goals.

5. **Execute**
   Performs the commands / workflows using actuators (e.g. service endpoints) of the Managed System. The changes are based on the instructions of the Plan stage.

9.6 Crowdsourcing

Outsourcing a job traditionally performed by a designated agent to an undefined, generally large group of people in the form of an open call

1. **Task Creation**
   Definition of task/job by requester. In case of big task, breakdown in micro tasks. Definition of worker requirements and qualification.

2. **Allocation**
   Task Allocation: Based on worker qualification and collected meta data, appropriate and available tasks are recommended to workers. Worker Allocation: Based on task description, appropriate and available workers are identified.

3. **Task in Progress**
   After accepting a task, worker starts to work on it.

4. **Evaluation of Work**
   After finishing a task and providing the result, the work is evaluated by the requester.

5. **Payment of Work**
   After positive evaluation of the work, worker gets paid by the requester.

6. **Worker Profiling**
   After each work on a task, the worker profile is updated based on the meta data collected during the whole process. The updated worker profile is used for new allocation of tasks and workers.

9.7 Success and Risk factors for CI

Success Factors:

- Choosing the right type of CIS

- Appropriate set of CI design patterns
- Provide low friction, easy to use means on contributing content
- Effective feedback mechanisms, which make users aware about activities of other users

Risk Factors:

- If CIS is not well integrated in users' workflow routine, the CIS will not be used
- Focusing more on the software side, and neglecting the user base side
- Cannibalization of user activity by other CIS
- Handling of security and privacy of user data (logs)

9.8 Challenges in CI

- **Designing** the "right" functional architecture.
  Requirement elicitation of user needs and optimization potential. Getting the basic workflows right.
- **Perpetual beta** due to constant evolution of capabilities.
  Continuous delivery
- Fostering an **active community of contributors**.
  Users are scarce resource: Competition with existing platforms. Engagement (incentives, motivation).
- **Scaling** up to (ultra) large-scale proportions.
  Big data and Machine learning, Cloud computing, Global software development

9.9 Human based computing VS CIS + examples

- Human Based Computing utilizes human processing power to solve problems that computers cannot (yet) solve. Example: ReCaptcha, select images to help AI,..
- CIS harness the collective intelligence of connected groups of people by providing a web-based environment to share, distribute and retrieve topic-specific information.
- Differences: In HBC the "humans serve the computatio" and in CIS the "computation serves the human". Both are (more or less) driven by users who contribute.
- Two types and examples: 1. Social Network Services (Twitter, Facebook), 2. Media/Content Sharing (Wikipedia, Youtube)

9.10 CI design patterns

- **Tagging**: Enables users to categorize content on their own using keywords.
- **Rating**: Enables users to express their like or dislike about the content so that other users can benefit.
- **Comments**: Enable users to express their own opinion to a topic and encourages discussions with others.
- **Hashtags**: Enables users to add keywords/tags using # symbol among text to facilitate discoverability later.
- **Recommendations**: Seek to predict content the user might be interested in based on other users' behavior and decisions.
- **Generated lists**: Enable users to get fast status overview of trending topics and news.
- **Follow/subscribe**: Enable users to define specific content they are interested in and that they would like to monitor.
- **Activity indicator**: Enables users to assess how relevant the content is based on other users' actions.
- **User-generated Collections**: enable users to create their own collections of curated content and to provide them other users

9.11 Advantages and Disadvantages of centralized and decentralized CIS

Centralized CIS Acts as one platform and is operated by a single provider. Data is concentrated in a single system. (e.g. Facebook, Wikipedia, ...)

- **PRO:**
  - Constant quality of service for all participants.
  - Single point of access.
  - More resources for system maintenance, data security, platform evolution.
  - Single organization to hold accountable (e.g. privacy issues).
  - Effective information exchange due to very-large scale user base.
- **CON:**
  - Single point of failure.
  - Prone to censorship and systematic infiltration by government organizations.
  - Often closed/proprietary system source code.
  - Lots of influence concentrated in a single organization

Decentralized/Federated CIS Is decentralized and distributed across multiple providers. Consists of independent CIS instances which are able to communicate with each other. (e.g. GNU Social, Diaspora)

- **PRO:**
  - Multiple points of access.
  - Robust: Infiltration only of individual points; Easy hosting of new nodes.
  - Often system source code is open source (= auditable).
  - Easier to operate on non-profit basis due to lower resource needs of individual nodes.
- **CON:**
  - Quality of service dependent on individual node.
  - Each node is responsible for its maintenance and data security.
  - Less effective information exchange due to fragmentation of user base.
  - User contributions are stored on an individual node.

10 Product Line Engineering

10.1 Software Pruduct Line

Definitions

- A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.
- A set of programs is considered to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.
- Software product line engineering refers to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production.

So IntelliJ, PyCharme... is a product line of JetBrains.?

Characteristics

- collection of similar software systems - family
- shared set of software assets - reuse of commonalities
- for a specific domain - domain engineering

Benefits

- Improved productivity
- Increased quality
- Increased reliability
- Decreased cost
- Decreased certification costs
- Decreased labor needs
- Decreased time to market

10.2 Time VS Space Variablility

Variability in Time - Releases Different versions of an artifact that are valid from a different point in time

Variability in Space - Variants An artifact that comes in different forms at the same time

10.3 Variability by Time

Variability can happen at different Times:

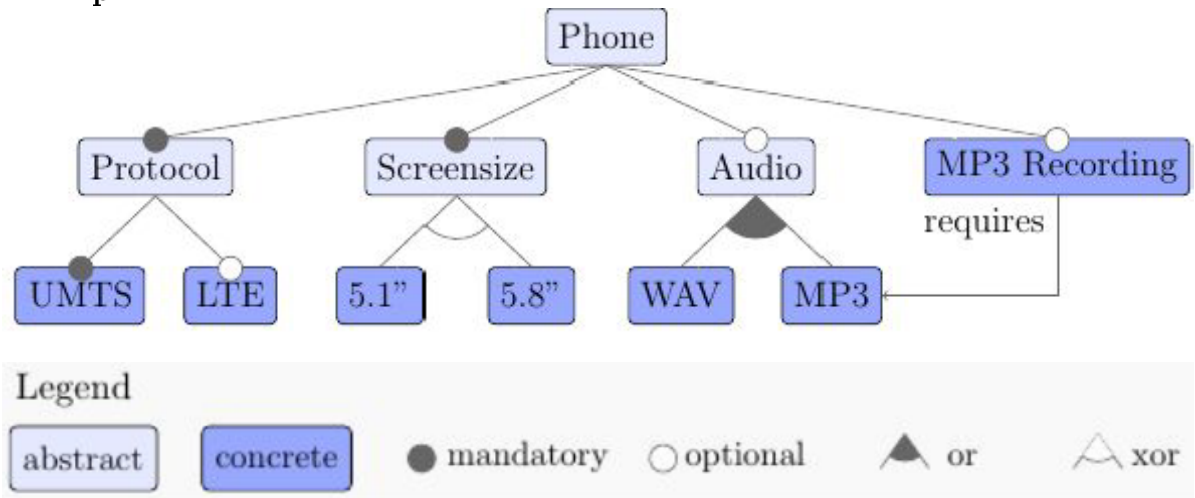
- **Compile Time** - e.g. variability within the source code
- **Build Time** - e.g. variability via build tools
- **Linking/Binding Time** - e.g. variability via Dependency Injection
- **Runtime** - e.g. Microservice Architecture

10.4 Ways to implement Variability

- Parameterization (e.g. command line arguments, build tool)
- Configuration Constants
- Object Orientation (e.g. interface, factory pattern)
- Dependency Injection
- Aspect Orientation

10.5 Feature Models

Visual description that allows to describe a set of features and their dependencies.
Example: Feature Tree



11 Clean Code

11.1 Naming conventions

- If you need a comment you are doing something wrong!
- Descriptive (long) name > short name
- Precise names for small classes > generic names for large classes
- Clarity is king
- Length of a name should correspond to the size of its scope

11.2 Function conventions

- One level of abstraction per function
- One level of indentation
- Tell a story with a function
- Use three arguments in a method signature as maximum - best none
- Beware of boolean arguments - they do more than one thing!
- Tell - don't ask e.g. pass a function that gets executed

11.3 Error Handling conventions

- Always write try-catch-finally first
- Never pass or return null (use empty list, optionals, ...)
- Never hide behind errors
- Never use errors to influence control flow
- Never use destructive wrapping (better pass causing exception)
- Log XOR throw, handle XOR pass on
- Make exceptions part of your domain

11.4 Comments conventions

- Good code does not need comments
- Comments are not documentation, don't use them as documentation
- Better don't use comments at all

11.5 Class conventions

SOLID

- **S – Single Responsibility principle**
  a class or module should have one, and only one, reason to change
- **O – Open-Closed principle**
  A software artifact should be open for extension, but closed for modification
- **L – Liskov Substitution principle**
  Derived classes must be substitutable for their base classes
- **I – Interface Segregation principle**
  Many client-specific interfaces are better than one general purpose interface
- **D – Dependency Inversion principle**
  Depend upon abstractions - not implementations

11.6 Clean Test Code

A test is not a unit test if

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run correctly at the same time as any of your other unit tests
- You have to do special things to your environment to run it

Three laws of test-driven development

- Don't write production code until you have written a failing unit test.
- Don't write more of a unit test than is sufficient to fail.
- Don't write more production code than needed to pass the currently failing test.