Advanced Software Engineering - WS22

SUMMARY + LECTURE NOTES

Release Your Stuff 3 Times a Day

- **Dependency Management**
    - BAD example
        - managing dependencies by keeping the files in SCM or other file storage
        - Problems
            - loose version information (unless included in filename or package/manifest)
            - no standardized naming
            - loose trace of source (where downloaded…)
            - no info on transitive dependencies
            - updated versions need to be added manually
            - SCM is not built for versioning binaries (no diff, high resource usage,...)
    - Proper handling
        - Declare which libraries are used + version of the library
        - Declare context the library is used in (test / production)
        - Declare where these libraries are coming from
        - Used libraries declare which libraries they are using themselves
        - Automatically retrieve all required libraries from a repository
    - Tools:
        - Maven - also for build management, testing, release management, executing plugins,...

```
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>3.6.1</version>
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>

dependencies>
```

Group-ID: `com.google.protobuf`
Artifact-ID: `protobuf-java`
Version: `3.6.1`
Scope: `compile` (default value)

        - Gradle - DSL instead of XML, official build tool for Android
        - Apache Ivy - pure dependency management
    - Benefits of dependency management systems
        - Automated Bill-of-Materials (BoM) - if software is sold customer might want to know, what other libs,... are used (licensing)
        - CVE (Common Vulnerability & Exposure) Scanning
        - OSS License Compliance
            - check dependencies for appropriate licensing

        - no viral licenses “infect” your project (copy left,...)
        - take action to ensure compliance with individual licenses
  - Semantic Versioning
    - Set of rules for software with public API
    - is a best practice not a fixed rule
    - Pattern: X.Y.Z
      - X: Major version, incremented if backwards incompatible changes are introduced
      - Y: Minor version, incremented if new, backwards compatible features are introduced
      - Z: Patch version, incremented if only backwards compatible bug fixes are introduced
    - E.g. standard maven versioning
      - 3.0.0-SNAPSHOT (snapshot is the qualifier for nightly/local build; SNAPSHOT tells maven to not assume that this version will not change)
      - 2.0.0-RC3 (release candidate - not standardised!)
      - 2.0.4 (final versions usually have no qualifier)
    - do not change something and publish with same version number - just use a new one

- **Repository Management**
  - Tasks:
    - Manage all used (third party) dependencies & repositories (even the ones not readily available in public repos)
    - Proxy and cache remote repositories
      - results in faster builds
      - easy traceability
      - fault tolerance
      - enhanced security (supply chain attacks - exfiltrating system via third-party product that has been granted access)
    - Manage all artefacts created by your project (binaries, sources, documentation, configuration)
      - central location for all artefacts -> accessibility & easier backups
      - no need to always build complete project
      - archive for past releases
      - write once (should never change) - else this might confuse users
  - Tools:
    - jFrog Artifactory (Java)
      - open source and commercial version
    - Sonatype Nexus (Java)
      - open source version and commercial version (with support)
    - Apache Archiva (Java)
      - open source with fewer features (but full-fledged repo)
    - most ecosystems have native mechanisms
      - node.js - npm

            - python - PyPI
            - ruby and rails - RubyGems
            - Perl - CPAN
            - C/C++ is fragmented - e.g. Conan
        - Java does not have an official archive - but Maven Central is de facto standard

- **Build Management and Automation**
    - Tasks
        - Retrieve dependencies
        - Prepare resources
        - Compile source code to binary format
        - Package binaries & resources
        - Execute automated test cases
        - Execute static code analysis and reporting
        - Generate documentation
        - Run application locally
        - Deploy application
        - Release & publish artefacts
    - Build Management Tools
        - make
            - controls the generation of executables from source files
            - makefile determines how to build the program
            - only perform step (target) when source has changed
            - problems with makefiles
                - structuring is not predefined (e.g. you can write a file with 1000 lines or with 10)
                - everyone writes makefiles differently (has to be read just as source code)
                - configuration management does not exist -> pre-processing is necessary
        - Apache Ant / NAnt
        - Apache Maven
            - Convention over Configuration
                - every build is done in the same order (phases are ordered)
                - plugins enable work in phases (there is a standard config that of course can be changed)
                - you cannot force maven to execute phases in a different order
        - Gradle
        - MSBuild
        - Rake,...
- **Release Management**
    - goal: create stable reproducible artefacts
    - maven-release-plugin codifies best practices and provides safety nets
        - Step 1: maven release:prepare

            - verify no un-commited changes & no SNAPSHOT dependencies
            - build and execute tests
            - set release version number
            - commit to SCM with tag
            - increase version number, append -SNAPSHOT and update SCM section (new version to keep working while the release version is released)
            - commit to SCM
        - Step 2: maven release:perform
            - checkout previously created tag
            - build and deploy artefact to local and remote repo (release should include (java)doc and sources)
        - there is a roll-back feature (for step 1) - step 2 cannot be rolled-back as the code is already released publicly
    - changelog / release notes
        - content depends on recipient
            - technical - simple issue tracking report
            - non-technical - features and functional bugs
            - communicate to stakeholders (QA, PM, dependent projects, end user) - what has changed since last release
        - Tools - Issue Trackers (minimize overhead, align versioning between issue tracking and code base) -> discipline necessary

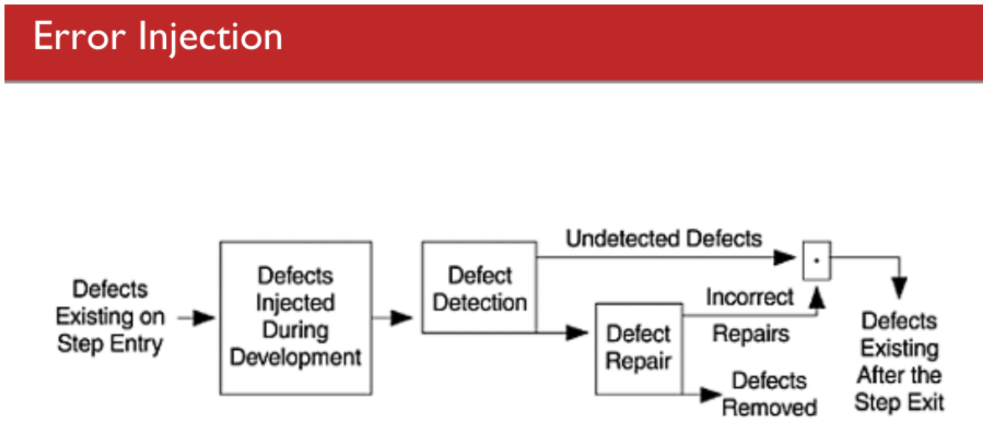
- **Continuous Integration**
    - Principles after Fowler: (10)
        - Maintain code repo
        - Automate Build
        - Make build self-testing
        - Everyone commits to the baseline every day
        - Every commit (to baseline) should be build
        - Keep the build fast
        - Test in a clone of the production environment
        - Make it easy to get latest deliverables
        - Everyone can see the results of the latest build
        - Automate deployment
    - Tasks:
        - Execute a full build of the project after every commit
        - always know & communicate the state of the repo
        - Publish your build artefacts (binaries, doc, config, reports)
        - Deploy and run your application
            - binaries and config have to fit together
            - usually not done continuously (as in every few minutes) but as a nightly build or when needed
    - Terminology
        - Continuous Integration (CI)
            - constantly merge development work with mainline
            - build and test automatically

            - Tools
                - Hudson/Jenkins
                - Apache Continuum
                - CruiseControl
                - Gitlab CI
            - Pipelines:
                - logically structure CI build into series of steps
                - information of CI config is stored alongside code (e.g gitlab-ci.yml, Jenkinsfile)
        - Continuous Delivery
            - continuously deliver your code to a staging environment
            - deployment to production requires manual interaction
        - Continuous Deployment
            - automatic deployment of code from SCM to PRODUCTION
            - requires CI and CD to be in place
- **Put it all together**
    - Enterprise
        - private setup - usually on premises
        - resource intensive - hardware (server, storage, network, rack,..) and human (admin, config, know-how,...)
        - high entry cost
        - full control and flexibility
            - integration with existing resources - e.g. LDAP/Active Directory
            - choose the tools you need
            - use your infrastructure (security!)
        - integration of tools often hard
        - scalability can be an issue for large organizations
        - support only for individual, commercial tools
    - Cloud
        - everything is hosted by external service provider(s)
        - easy setup (fully web-based config)
        - good integration of selected tools (provider is responsible for and supports complete tool chain)
        - easier scalable
        - source code leaves own servers/network/premises
            - often also the country (legal implications)
        - Tools
            - Source Code Management
                - GitHub
                - BitBucket
            - CI
                - Travis CI
                - CloudBees
            - Repository Management
                - BinTray
                - CloudBees
            - Development Server/PaaS
                - AWS

            - Google App Engine

Case Study - Vienna International Airport

- Complex software
    - Dependable Software
        - Attributes
            - availability - readiness for correct service
            - reliability - continuity of correct service
            - safety - absence of catastrophic consequences for the user(s) and the environment
            - confidentiality - absence of unauthorized disclosure of information
            - integrity - absence of improper system state alterations
            - maintainability - ability to undergo repairs and modifications
        - Means
            - Fault prevention
                - Quality control
                - Software design - structured programming, information hiding, modularization
            - Fault tolerance
                - Error detection and subsequent system recovery
                - Error handling - roll-back vs. roll-forward
                - Redundancy: fault masking, voting algorithms
                - Fault isolation
            - Fault removal
                - Verification (static, dynamic), diagnosis, correction
                - Fault injection (test error handling)
                - Corrective and preventive maintenance
            - Fault forecasting
                - Qualitative - identify, classify, rank
                - Quantitative - probability model (stochastic)
        - Threats
            - Faults - abnormal condition that can cause element to fail (e.g. uncaught null pointer exception)
            - Errors - discrepancy between observed and actual correct value or condition (e.g. null value when not valid)



            - Failures - termination of ability of an element (e.g. malfunction or crash of service - i.e. due to unhandled nullpointer)
        - Safety culture problems
            - management
                - diffusion of responsibility and authority
                - limited communication channels and poor information flow
            - technical
                - inadequate system and software engineering
                    - specifications
                    - unnecessary complexity and functionality
                    - software reuse or changes without appropriate safety analysis (think of Ariane; reusing old well-tested software is cost-efficient - however must be tested in new environment thoroughly as well)
                    - inadequate review activities
                    - ineffective system safety engineering
                    - flaws in tests and simulation (environment)
                    - inadequate human factors design for software
    - 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
            - docu and reviews
            - restructuring including partial replacement
            - plan for retirement and replacement (e.g. no hard-coded values - make stuff configurable)

- **System migration - example**
    - Technical strategy
        - migration type - e.g. 1:1 migration (feature-wise, not technical)
        - minimize changes in the legacy system (high risk)
        - incremental transfer of user groups into new system
        - migrate smallest possible size but coherent parts - technical little big bangs
        - parallel operations of both until ok to turn old system off completely (depends on how important it is to have no downtime)
    - Usability engineering
        - contextual enquiry of the working environment - How is the operative environment for each user set up?
        - individual design of user interface for each user group - functional replacement but with improved user interface
        - analysis of usage statistics of legacy system - how frequently is a function used by a user and what is the workflow; why is it that way?
        - mockups before implementation

Build for ten years and more

- **Planning for extended lifecycle**
    - Key factor is change -> design to minimize costs of change
        - Reuse
        - Extendability
        - Feature Changes
        - Scalability (Change in load / throughput)
        - Maintainability (Robustness of change)
        - Testing for regressions

- **Fundamental approach**
    - decomposition of system into independent parts
    - recomposition of parts into coherent system
        - context-aware
        - multiple system instances
        - static (build-time) vs. dynamic (runt-time)
    - component vs. service (after Fowler)
        - component: glob of software that is intended to be used without change (using application does not change the source code of the component; but may alter behaviour by extending it) by an application that is out of the control of the writers of the component
        - service: similar to component as it is used by a foreign application; main difference - component is used locally (jar file, assembly, dll, source import,...) and a service is used remotely through some remote interface either synchronous or asynchronous (web service, messaging system, RPC, socket,...)
    - example:
        - separation between user interface and business services (REST, support for future UI-technologies)
        - auto-refresh UIs (JS-polling)

                - customizable workflows for all business processes
                - customizable rules and layouts for the notification system
                - customizable templates for message sending
                - open-source based

- **Interfacing / Integration**
    - key design decisions
        - service (pull)
            - runtime (webservice) vs. build-time (java library)
            - general vs. specific interfaces
            - synchronous (request-response) vs. asynchronous (call-back)
            - ID vs. natural key object identifiers
            - primitive type parameter vs. DTOs
            - delta vs. full updates of data/information set
            - transformation (legacy interfaces/views)
            - versioning of interfaces
            - validations, return values, error codes
            - reuse of components / resources
        - message (push)
            - synchronous or asynchronous
            - event data models (payload)
            - internal vs. external events
            - primitive vs. complex (compound) event types
            - typical event payload
                - reference to a primary object
                - actual value(s)
                - previous value(s)
                - associated action
                - time of event creation
                - source of event creation
        - data coupling
            - separation of schemata
            - read access through views
            - write access through procedures
            - easiest type of integration to achieve and hardest to get rid of
- **Layered software design (API Design)**
    - Why cut software into layers / modules
        - Separation of concerns
        - abstraction
        - testability
        - error handling
        - transaction management (what is the exact transaction scope?)
        - reuse
            - frameworks
            - custom (DAOs in other projects)
    - How to cut software into layers / modules
        - separate UI from logic
        - separate model from logic

        - separate data access from logic (via a common interface)
        - separate connectors from logic (via a common interface)
    - Key questions for choosing the right level of modularization
        - fine-grained vs. business services
        - requirements on transactional capabilities
        - requirements on high availability & distribution
        - release and deployment scenarios
        - lifecycle (legacy connectors)
    - Forms of modularization
        - Build time
            - multiple JARs possible
            - single Bundle
            - update requires complete redeploy
            - easy operation
            - easy and fast intermodule communication
        - Runtime (single VM)
            - multiple JARs required
            - multiple bundles
            - update requires partial redeploy
            - medium complex operation
            - more complex but fast inter-module communication
        - Runtime (multi VM)
            - multiple JARs required
            - multiple bundles
            - update requires partial redeploy
            - highly complex operation
            - complex and possibly slow inter-module communication
    - Java technologies for modularization
        - OSGI (runtime)
            - initially created for the embedded systems domain
            - additional control over how classpath is constructed
            - targeted for single-VM operation
        - Maven (build time)
            - support for simultaneous assembly of multiple modules
            - management of direct and transitive dependencies
        - Project Jigsaw (language level modularization)

- **Single Service, Multiple-Consumers**
    - Callstack
        - Clients (GUI, external system, Telex)
        - REST Layer, JMS Consumer
        - Business Service
        - Data Access Layer
        - Tx-Boundry (commit)
        - Postprocessing
            - Notifications
            - Connected systems (data push)
            - Legacy system sync

- **Dependency Injection (DI)**
  - gluing of objects is separated from the implementation
  - all implementation is against the API
  - central definition and container that creates and binds objects together
  - DI supports code reuse and independently testing classes
  - DI support different bindings for different environments
  - DI supports lazy creation of objects (e.g. useful for limited memory environments)
  - DI framework provides the runtime services for Di (e.g. Spring framework)
    - Spring framework:
      - modular
      - allows to pick and choose modules that are applicable to your application
      - POJO’s (called beans) -> managed by Spring IoC container
      - container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application
      - DI helps in gluing loosely coupled classes together and at the same time keeping them independent
      - supports the utilization of existing frameworks (logging, ORM,...)
      - web model-view-controller (MVC)
      - coherent transaction management interface (JTA)
      - API for translating technology-specific exceptions (thrown by JDBC, Hibernate, JDO) into consistent, unchecked exceptions
      - inversion of control (IoC) containers are lightweight (beneficial for developing and deploying applications on computers with limited resources)
      - testing is simple because environment-dependent code is moved into this framework

- **Aspect Oriented Programming (AOP)**
  - class in OOP
  - cross-cutting concerns are the functions that span multiple points of an application
  - cross-cutting concerns are conceptually separate from the application’s business logic
  - AOP helps you decouple cross-cutting concerns from the objects that they affect
  - aspects are woven in at compile time or runtime

- **Event based architecture**
  - loose coupling
  - activator - after transaction commit
  - foundation for asynchronous processing
    - connected systems
    - messaging within the system

        - messaging to other systems
    - implementation with spring integration
        - enables persistent queuing (asynchronous processing)
        - existing producer-consumer pattern
    - eventing vs. batch
        - advantages:
            - fail-safe through retry
            - quicker transaction (user wait)
        - disadvantages
            - hard to trace
            - time-delay
            - testability
            - parallel operations of eventing vs batching

Problems you solve for every project

- cross-cutting concerns usually need tailoring
- handling these concerns separately from your business logic is a major factor for retaining clean, readable code

- **Contexts**
    - used to store state necessary to enable handling of cross-cutting concerns
    - associated with overarching scope
        - RequestContext
        - ThreadContext
        - ApplicationContext
    - initialized (and destroyed) by handlers/filters
    - stored in ThreadLocal variables
    - Example: authentication - user is already connected with context -> different context e.g. depending on how long they are needed (authentication for the whole session, for just one operation,...)

- **Transaction Management**
    - Models
        - describe the expected transactional behaviour
        - describe how transactions are implemented
        - who is responsible for the transaction?
            - Local transaction model
                - underlying database (auto commit)
                - connection based
                - this model just delegates -> send statement - connection established
                - is simple but limited -> not so often in use
            - Programmatic transaction model
                - developer (no auto commit)
                - transaction manager
                - developer is responsible and must handle transactions
            - Declarative transaction model (Container managed transactions - CMT)

            - developer specifies the behaviour
            - container handles transaction
            - code can be generated and configured using e.g. annotations
    - Strategies
        - describe how transactions are utilized
        - what is considered a unit of work? / at what level to handle transactions
            - client orchestration
                - for fine-grained (in-process) APIs
                - lower level
            - API Layer
                - for coarse grained methods
                - higher level
                - every call to API will be a transaction
            - Variation:
                - High Concurrency - optimizing each call individually
        - Distributed transactions (global transaction)
            - allow atomic behaviour over more than one resource (database, message queue, ...)
            - specified in XA (eXtended Architecture)
                - uses the 2-phase-commit (2PC) protocol to ensure atomic commits
                - Java Transaction API is based on XA standard
            - should only be used when absolutely necessary
                - not possible to cover all cases of (physical) failure - e.g. some race condition, non-determinism in distributed systems
                - many problems can be solved by fine-grained, manual control of commit sequence
        - declarative transactions - Spring example
            - @Transactional
            - Transactional Interceptor
                - Begin / commit transaction
                - Join existing transaction
                - Rollback in case of (unchecked) exception
            - Correct configuration of transactions is crucial
        - Choosing Transaction Management Strategies
            - If running inside Container: declarative transactions
            - Important: Understand managed persistence context
            - Managing Transactions manually results in a lot of code and is error-prone
            - CMT (container managed transaction) makes tests harder

- Logging & Auditing
    - Logging
        - Technical, text based output
        - for detecting and debugging problems

        - level can be configured
        - output not easily understandable
        - output not for automated processing
        - short term retention
        - Technical
            - Performance implications
                - avoid expensive operations (id instead of whole dataset)
                - avoid unnecessary concatenations
                - too much, too long log = bad performance
            - don’t write to stdout stderr (no proper separation by log-level & less control; sensitive data can be leaked to the system environment)
            - needs to be configured correctly
            - if you log something as an error it must be a SYSTEM error not a USER error - users just insert incorrect data that is to be expected and not an error
        - Reading logs
            - provide contextual information, especially for clustered or distributed systems TTTech!
            - use TOOLs (splunk, openSearch)
        - good log output
            - limit log levels (4 are usually enough, debug, info, warn & error)
            - clear rules when to use which log level
            - automatically adjust log level according to situation
                - use auto-adjustment for log-level (based on metric - e.g. a lot of TCP-session start requests)
            - provide contextual info
            - provide reference e.g. user (id=69) created
            - adjust log output when you gained more info
    - Auditing
        - Domain specific
        - fine-grained and structured output for tracing user activity
        - requirements are specified by legal and company policies
        - used by end user group (e.g. legal team)
        - long term retention (30+ 10+ years)
        - technical
            - no or little framework support available
            - requirements to diverse for generic solutions
            - needs quality assurance / specification
            - frequently depends on already existing (in house) product
            - use proper SLAs (service level agreement) for external product
            - work async as often as possible (performance)
    - how to correctly configure logging or auditing
        - Select good logging lib
        - Configure correctly (just enough information)
        - Simple rules for developers

                - Review log output on regular basis
                - Make logs accessible
                - make sure performance does not suffer

- Authentication & Authorization
    - Authentication
        - verifies identity
        - Types of Authentication
            - Username / Password
                - easy, use + implement
                - still mostly used, but more and more insecure
            - Token based / Single Sign On (SSO) / Deferred
                - SAML
                - OAUTH 2.0
            - Certificate based
                - Complex to roll out and manage
                - used in high security environments
            - Smart card, biometric
                - Bürgerkarte, Fingerprint Sensor
                - Usually needs client side support
                - Fronted to certificated based auth
    - Authorization
        - determines access rights
        - Types for Authorization
            - Role based access control (RBAC)
                - Frequently used for resource-based systems
                - easy to govern
                - well-supported by standard technologies
                - minimal performance implication
            - Permission based
                - simple action based
                - complex expressions
                - easy to govern
                - well-supported by standard technologies
                - minimal performance implication
            - Access control lists (ACL)
                - delivers fine-grained control on an (object) instance level
                - complex to maintain, but complexity is sometimes needed
                - significant performance implications
            - Rule based
                - suitable for complicated and frequently changing business requirements
                - complex to maintain, but complexity is sometimes needed
                - significant performance implications

    - Declarative Security
        - provided by container out of the box
        - @RolesAllowed
        - Spring extends mechanism -> expression based security checks
        - decide on scope for security (API level, client/user interface level)

    - Identity Managements
        - how to handle user related data
        - database, custom application
        - active directory / LDAP
        - managing users and granted authorization can become very complex with a growing number of users and actions
    - How to choose an appropriate access control mechanism
        - reuse existing infrastructure or frameworks / build your own
        - decouple authentication, authorization and identity management
        - keep business code clean of provider specific dependencies
        - adhere to organizational requirements
        - consider performance implications (complexer authorization methods)

- **Error Management**
    - user error vs. program error
    - program flow vs. exception
    - how and what to communicate to the end user
    - related to logging and UI as well as client side validation
    - Types of exceptions
        - checked exception
        - unchecked exception
        - error
    - how to error handling
        - does this method have enough info to handle exception?
            - yes -> handle it
            - no
                - does the caller have enough info
                    - if yes -> re-throw
                    - no
                        - does the caller need to specifically handle failures in operations from this component
                            - yes -> re-throw as nested within a component exception subclass
                            - no -> re-throw unchecked
        - do not expose lower level exceptions to upper layers (API bleeding)
        - higher layers should catch lower-level exceptions and wrap them in higher-level abstractions (e.g. database SQL error -> error getting data)
        - use interceptor/aspect + annotation
    - how to not to error handling

        - Log and throw -> do either one or the other
        - catching or throwing “exception”
        - destructive wrapping -> always pass the causing exception
        - catch and ignore
        - throw from within finally -> will swallow any other exception
    - 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

- **Internationalization** & **Localization**
    - Internationalization
        - The preparation of a product for use in the global market, usually done only once. (No source code changes necessary)
    - Localization
        - The Adaptation of a product to launch in a specific locale.
    - Focus Points
        - Language & Text
            - Char Encoding (UTF-8)
            - Orientation: Left to right
            - Sorting
            - Pluralization
                - “0 Personen” vs. “1 Person” vs. “5 Personen”
                - Only supported for easy languages in Java (eg not Polish)
            - Collation (Groß klein Schreibung)
                - Some languages don’t have a 1 to 1 mapping for collation (Turkisch 2 lowercase i)
        - Culture
            - Names and titles
            - Weights and measurements, paper sizes
            - Telephone, Addresses, Postal codes
        - Conventions
            - Currency format
            - Date, Time, Time-zone and calendar
            - Number format
    - Java Technologies
        - ResourceBundle
            - One file per supported language
            - string.format() or MessageFormat.format() for parameterized messages
        - Pitfalls
            - Property files are Latin-I
            - No type safety
            - No compiler checks
    - How to prepare your product for a global audience?

            - Consider Internationalization right from the beginning
                - Char Encoding
                - Locale & TimeZone
            - Know your target market to avoid overhead
            - III8n is not only translatable text
            - Make use of tools & frameworks

From Prototype to Product

- Project styles
    - waterfall style
        - plan, specify, design, build, test & deploy
        - no incentive to think about operation before testing
        - managers tend to micro-manage
    - Agile
        - Potentially shippable code every day
        - Integrate continuously
        - Deploy continuously
        - Not universal cure
            - Depends on team and organization
            - Requires trust

- What is DevOps
    - Designing Operational Aspects together
    - Considering operation from the beginning
    - Better communication between OPs and Devs
    - It's about knowing how the other side works
    - Shift left approach
        - Thinking about possible problems early on
        - Left = Dev | Right = OPs

- Configuration as Code
    - YAML
    - XML
    - Norway Problem
        - NO is parsed as False

- Configuration Management
    - Build Configuration
        - State of your source code
        - how to build
        - dependencies
        - state of your requirements
        - state of your defects
        - documentation of executed tests (test plans)
    - Product configuration
        - User config, as in config from a user perspective
    - Application server / database configuration

        - Often done in database or alongside source code
        - Keep config in as few places as possible
        - Application should handle wrong config
        - Clear distinction between data and config in database (namespaces)
    - OS configuration
    - System configuration
        - timezone
        - user language
        - memory assignment (java -Xmx)
        - avoid manually tinkering with the environment -> use libs / tools
        - infrastructure as a code (treat config like code)
        - use virtualization and containers to simulate environments

- Clustering vs. Load Balancing
    - Clustering
        - Application-Level (full/delta session replication)
        - Database-Level (Requests need to be stateless)
        - Reasons:
            - Server can't handle everything alone
            - Redundancy
            - Better Locality
        - Caching
            - In-Process Caching
                - One cache per process
                - maybe inconsistencies
                - higher memory usage
                - fast and easy to implement
            - Distributed Caching
                - slower due to overhead
                - more complex
                - scales better
                - no OutOfMem risk
                - does not use the memory needed for the program
        - Session Serialization
            - each session is replicated to all other nodes
            - Java: everything in session must be serializable
            - possibly a lot of network traffic
            - know what is in session
            - keep session small and stable
            - often used together with application layer clustering
    - Load Balancing
        - Sticky session
        - Round robin
        - Active / passive
        - Hardware vs. Software
    - Tradeoff between load distribution and fault-tolerant
    - Always perform fail-over tests on your setup (under load)
    - Master Node Election

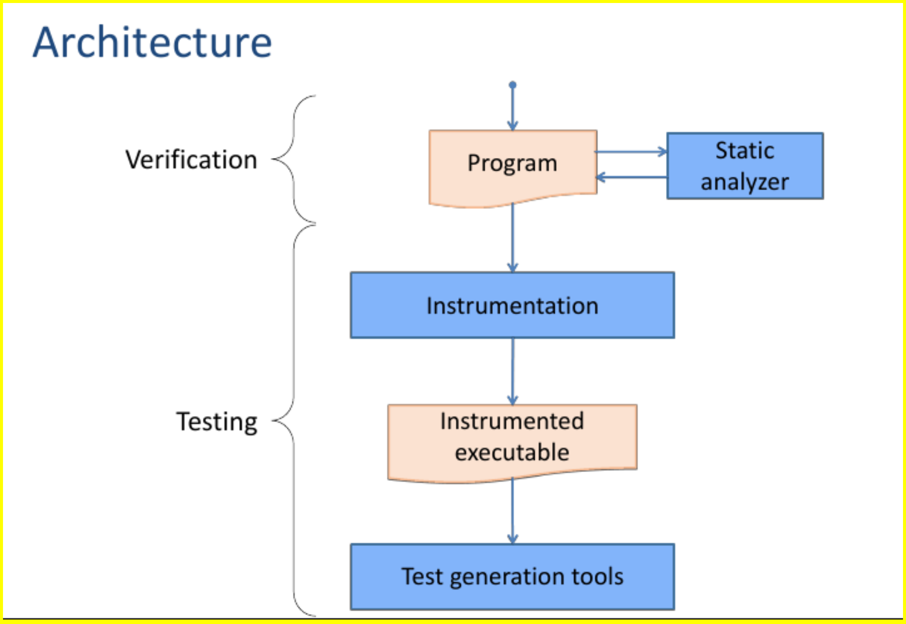
        - ensures something is only executed once
        - ensures messages are handled in correct order
        - used when one node has to mediate or delegate
        - automatic master node election is difficult (unless single resource for synching)
            - split brain problem - half of the total amount + 1 is needed to make a decision (majority)
        - manual master node election
            - might result in downtime
            - possibility of human error
- Performance
    - Test vs Development Team
        - Frequent internal (white box) know how / specific configuration required
        - QS-department often do not have the necessary skills
        - Generating load is hard
        - Best done in collaboration
    - Testing is only the “last” step to verify
        - Consider performance during design and development
    - Target potential bottlenecks first
        - limited thread/connection pools
        - frequently used pages (caching?)
    - Database
        - Use clone of production database
        - Think about resulting database queries (abstraction)
        - be careful when operating on lists / result sets (lazy loading, n+1 query problem)
        - Think about indices that fit your query pattern
        - Optimize based on data / facts
    - System
        - Beware of all calls that are “leaving your system”
            - are there SLAs?
            - make statements about actual performance
            - minimize round trips
        - How does your system react to timeouts?
            - Timeouts tend to bubble up
            - Some timeouts can’t be easily influenced (browser timeout)
        - Consider automatic retries if you can correctly detect specific errors
            - be aware of worst-case scenarios
            - long timeout (3 retries with 5min timeout = 15min timeout)
    - Tools
        - JVisualVM
        - YourKit
    - Profiling Modes
        - Tracing
            - Done through byte code instrumentation
            - Delivers Invocation counts
            - can influence performance

            - can’t be used in production environment
        - Sampling
            - Periodically queries stacks of running threads to estimate slowest part of the code
            - No invocation count
            - Almost no performance impact
        - Manual measuring
            - Good to see call duration
            - Good for runtime behaviour
            - Good for adaptive measuring / reporting
            - Bad if really done manually => too much boilerplate code
            - Bad for measuring “everything” (e.g. find needle in the haystack)
        - Pitfalls:
            - Always use System.currentNanos() for measurement
            - Also Interceptors can be used to measure time (@Measured)
- Monitoring
    - Often seen as a pure operations task
    - Difficult to detect application level problems
    - Basic monitoring is easy
        - System state (e.g. server down)
        - system resources (CPU usage)
        - Java behaviour (heap state)
        - Infrastructure state (e.g. queue sizes)
    - All the above only indicate “disaster” cases, no way to look inside the application
    - Goal: bring domain specific knowledge into operations
        - Vertical Health check (Heartbeat)
            - Is UI reachable
            - does UI reach Backend
            - …
        - Application specific
        - Often highly specific to the monitored application
        - A lot of application specific monitoring tasks can also be handled by database queries

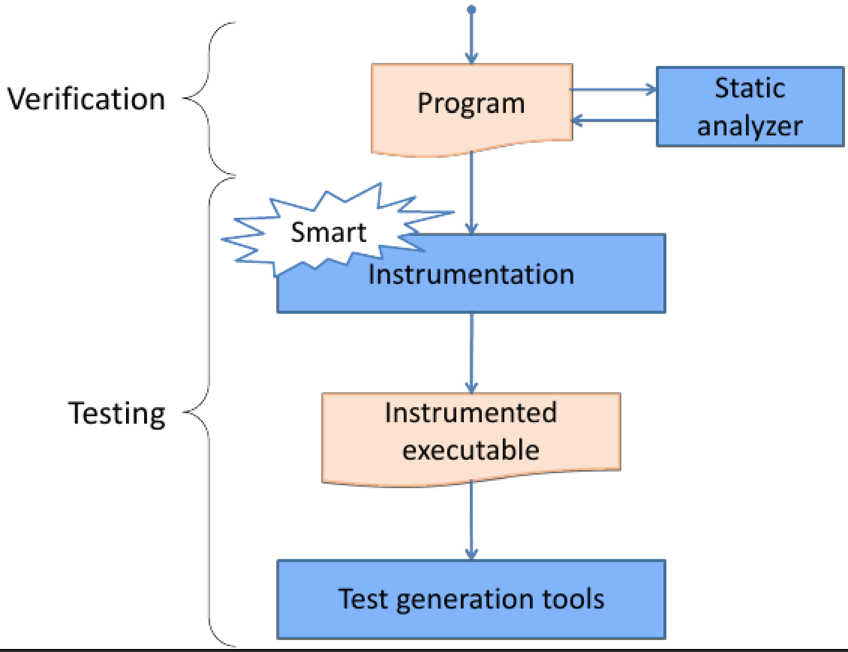
Systematic program analysis

- **What is program analysis**
    - how to build more reliable software while increasing developer productivity
    - Phases of program analysis
        - test generation
        - static analysis
        - software verification
        - human-computer interaction

- **Integrating program-analysis techniques to combine their strengths**
    - Static analysis
        - effective in detecting software errors

- increasingly applied in industry
- Compromises
  - reduce the annotation overhead
  - reduce the number of false positives
  - increase performance
  - preserve modularity
  - not checking program properties
  - making unjustified assumptions (e.g. this will never throw an exception)
  - being unsound
- Consequences
  - absence of errors not guaranteed
  - test effort not reduced
- Solutions
  - Annotations for assumptions
  - Instrumentation (instruments are used to monitor the values of variables or to detect assertion violation)
  - Dynamic test generation (different var assignments to get to different program sections
  - 

  - Problem about this architecture: all test cases for all branches still have to be generated as the assertion happens so late in the process ->



- Smart instrumentation
    - propagate conditions about unverified executions to higher up in the control flow
    - process
        - compute abstraction of program
        - infer conditions about unverified execution
        - instrument concrete program

- **Making program analysis more widely applicable**
    - Bias in machine learning
        - neural networks for criminal justice, health care, social welfare
        - concerns about fairness
        - neural networks may reproduce or even reinforce bias
    - Perfectly parallel certification of neural networks
        - fairness
            - given input features that are sensitive to bias (race, gender) a neural network is causally fair if the output classification is unaffected of sensitive features
                - e.g. credit rating algorithm is not influenced by age
            - check for fairness (naively) - certifying fairness
                - Analyse the neural network backwards (start at output)
                - forget value of sensitive feature
                - intersect the projected regions (non-empty intersection -> bias)
                - does not scale well
            - check for fairness advanced
                - forward and backward static analysis
                - forward: divide input space in independent partitions (reduce effort)
                    - not all inputs activate nodes in the network (not a 1-1 mapping or similar)

            - finding partitions by using
                - upper bound for number of nodes with unknown activation status
                - lower bound for size of dimension (features that divide into a lot of small groups instead of bigger groups are not as good)
            - partitions are made along NON-SENSITIVE features
            - characteristics
                - uses cheap abstract domain
                - balancing scalability and precision (with upper and lower bound - U and L)
                - may only consider a fraction of input space (e.g. hispanics over 45 years old discriminated against gender?)
        - backward: does naive approach for every partition (in parallel)
            - groups good partitions by abstract activation patterns
            - quantifies any bias
            - characteristics
                - expensive abstract domain
                - perfectly parallel
                - sound and in practice exact -> definite guarantees
        - certification fails -> biased region found

- **Testing program analysers for critical bugs**
    - why program analysers
        - wide applicability in software reliability
        - high degree of code complexity
        - severe consequences in case of errors
    - differential testing
        - compares analysis results on an input (multiple programs same input; not sure who is correct - not always the majority)
    - metamorphic testing
        - transforms an input such that the expected analysis is known (oracle is known)
        - metamorphic testing of datalog engines
            - datalog: declarative, logic-based query language (similar to ASP)
                - relations, facts and rules (head and subgoal)
                - engines:
                    - logicBlox
                    - DDlog
                    - bddbddb…

                    - may contain query bugs resulting in incorrect results (missing entries, including wrong entries)
                - given seed -> transform it such that new result contains old one OR is equivalent to old one OR is contained in old one
                    - detect bug: relation between old and new result does not hold
                - based on conjunctive queries (query containment)
            - metamorphic testing of SMT solvers
                - tools
                    - z3, STP,...

Ground Truth \ Solver result	SAT	UNSAT	UNKNOWN	Crash
SAT		A	C	D
UNSAT	B		C	D

A: Refutational unsoundness
B: Solution unsoundness
C: Incompleteness
D: Crash

                - given seed -> transform to generate SAT instances
                    - detect bug: solver returns UNSAT
            - metamorphic testing of Datalog engines and SMT solvers is effective in detecting fundamental correctness issues

Microservices

- **Cloud**
    - Pros
        - Scalability
        - Cheap for low traffic
        - Availability
        - Data security (cloud providers know what they are doing)
    - Cons
        - More expensive than dedicated hardware
        - Slower than bare metal
        - Complexity
        - Data security (legal and technical - you need to trust them, where are their servers located)

- **NoSQL**
    - Pros
        - Speed
        - Often tailored to specific task

        - Scalability
    - Cons
        - Not standardized
        - CAP Theorem - only two of
            - Availability
            - Consistency
            - Partition Tolerance

- **Machine learning**
    - Pros
        - Automation
        - Pattern recognition
        - Perpetual improvement
    - Cons
        - Training
        - Debugging
        - Accountability

- **Architectures**
    - Architecture
        - Abstraction of system
        - Helps communicating
        - improves maintainability
    - Distributed systems
        - Components are located on different machines
        - System appears as one
        - More scalable
        - more complex and harder to implement reliability
        - harder to deploy, debug and monitor
    - Monolith
        - One executable / deployable
        - hard to use different programming languages
        - complex environment setup
        - can be deployed manually
    - Service Oriented Architecture (SOA)
        - Distributed
        - Predecessor of microservices
        - Parts
            - Service broker
            - Service provider
            - Service consumer
        - Loose coupling of services

- **Microservices**
    - Definition
        - Small
            - Focussed on doing one thing well
            - Cohesion (everything that belongs together can be a service)

            - Small enough / no longer feels too big (when you break it into pieces, stop right before it is not useful any more after the split)
        - Independent
            - Communicate only over defined APIs
            - Independent deployment
            - Most changes affect only the service itself
        - Services
        - Work together
    - Pros
        - Distributed
        - Technical Heterogeneity
        - Quickly adapt to new technologies
        - Fault tolerance
        - Scalability
        - Deployment
        - Replaceable
        - Testing
        - Clear separation of ownership
    - Cons
        - Distributed
        - Technical Heterogeneity
        - Deployment (if there are a lot of services you are kinda forced to use DevOps)
        - Monitoring
        - Testing
        - Transactions
        - Reporting (e.g. joins for databases are often not possible as different databases are used)
    - Boundaries
        - You should have a well-defined border so that changing internals of the microservice does not affect other services
        - Domain Driven Design (Code Structure and Language matches Domain)
        - Bounded Contexts (Defines usage for a domain model) - often reflect departments of the business who talk a lot to each other
        - Conway's Law (ORGs design systems that mirror their own communication structure)
        - Loose Coupling
        - High Cohesion
        - Capabilities instead of data alone
    - Integration
        - Sync vs Async Communication
        - RPC, REST, HATEOAS
        - Binary vs XML vs JSON vs …
        - Message Queue
        - Orchestration vs Choreography
        - Shared code / client libraries
        - Breaking changes

- Deployment
  - Continuous Integration
  - Continuous Delivery
  - Continuous Deployment
  - DevOps
  - Configuration
- Testing
  - Unit test
  - Service tests
  - End-to-End tests (Integration Tests)
  - Dependencies may be mocked
  - Consumer-driven tests / contract tests (e.g. UI-Tests) - everyone that consumes a service provides tests that represent what they expect from the service -> enables to not break the consumers when the tests do not fail
  - Canary Releasing (rolling out changes gradually to a subset of users)
  - Non-functional Tests
- Monitoring
  - More services to monitor
  - Log aggregation
  - Metric aggregation
  - Correlation IDs (e.g. session id)
- Security
  - Authentication
  - Authorization
  - Single Sign-On
  - User to Service vs Service to Service Authentication & Authorization
  - TLS inside perimeter
  - SAML (Security Assertion Markup Language)
  - JWT (Java Web Token)
  - Client certificates
  - API Keys
- Conclusion
  - Many executables
  - Easy to use different programming languages
  - easy setup of environments, but many different needed
  - DevOps for deployment
  - Chatty microservices are undesirable
  - Need to get boundaries right, no breaking changes
  - Modelled after business domains DDD

- **From Monolith to Microservices**
  - Moving Code out of Monolith
    - Identify part
    - Define facades
    - Implement facades in monolith
    - use Facades
    - Move out of monolith

- Rewrite part as a microservice
    - Identify part
    - Define facades
    - Implement facades in monolith and new microservice
    - Switch to new implementation
    - Delete obsolete implementation

Free Open Source Software

- **Cargo cult programming:** just copying stuff of e.g. stackoverflow without understanding it
- **Benefits**
  - If bug found, one can just fix it, no workarounds
  - If a feature is missing one can just implement it (Custom Development)
  - Developing a generic Product costs more
- **Cons**
  - you NEED to fix stuff yourself or hope someone else does it

- **Ways to monetize OSS**
  - Adding commercially value on top of a base OSS offering
  - Professional Training
  - Embedding OSS into hardware
  - Service Contracts
  - Sharing the costs (pay a dev to help develop the OSS project as an organization; most OSS projects are frameworks and tools e.g. a database)
  - project consulting

- **What to avoid**
  - Don't sell the same product you give away for free
  - Respect freedom (respect community) - e.g. don't prevent people from forking
  - Don't rely only on a payroll (don't get influenced too much from a customer)
  - OSS project planing is different from company projects
  - Spread the influence across different companies

- **Legal Definitions**
  - Immaterialgüterrecht
    - Markenrecht
    - Musterrecht
    - Patentrecht
    - Urheberrecht
  - Sachenrecht vs Immaterialgüterrecht
    - Sachenrecht
      - bound to a physical thing
      - can only be traded exclusively
    - Immaterialgüterrecht
      - no physical representation
      - can be traded exclusively but also non-exclusively
        - e.g. A is allowed to sell books of it, B is allowed to sell books and films

- Copyright - summary term for different kinds of rights
    - Consists of many rights
    - difference between
        - Urheberrecht (nicht weitergebbar in Österreich)
        - verwertungsrecht
            - Werknutzungsrecht: exclusive
            - Werknutzungsberwilligung: non-exclusive
    - EU -> implicit
    - US -> rather explicit
- Threshold of Originality (Schöpfungshöhe)
    - only created original intellectual property if:
        - invented stuff yourself
        - it is not a trivial change
            - e.g not bugfix, not reformatting
- Authorship
    - author owns all the rights
        - Urheber eines Werkes ist, wer es geschaffen hat
        - Urheber hat mit bestimmten Beschränkungen aussschließliches Recht, das Werk zu verwerten
    - except when they are employed and do the work in their paid time (or sometimes it is sufficient that they use the resources of the company or their know-how - e.g. machines,...)
        - IP belongs to employer
        - different in US
            - depending on the country - even spare-time stuff might belong to the employer
- Code ownership
    - especially important to clarify in a customer relationship
        - state explicitly in contract
    - Zweckübertragungstheorie
        - Werkvertrag vs. Arbeitsvertrag (oder Arbeitskräfteüberlassungsvertrag)
        - nach der Zweckübertragungstheorie werden einem anderen nur die Rechte eingeräumt, die für den Verwendungszweck erforderlich sind (nicht mehr)
- IP and Open Source
    - make sure you really do own the IP
        - or make sure employer/customer is ok with you contributing source
        - trivial changes do NOT constitute IP
            - time compensation might still be needed
- CLA (Contributor License Agreement)
    - Make contributor aware of the legal impact
    - grant additional rights beyond the license
    - Symmetric vs Asymmetric CLA
        - asymmetric, often when owned by companies
            - company has extra rights, does not need to follow the license as everyone else

            - e.g. company does not need to publish stuff
        - iCLA vs cCLA
    - Code Provenance
        - where does the code come from?
        - important for big companies (in case of lawsuit)
            - prove the fact that you made the stuff and when

- **Open Source Licenses**
    - What is a License
        - Consensual Contract with rights and obligations (both contract partners know that you agree on the same thing)
        - Conditions under which someone can get rights to the code
        - Is not a contract, but close
        - Konkludente Verträge (durch handeln, e.g. ins Restaurant gehen, in die U-Bahn einsteigen)
        - Need to follow ALL the terms
    - Commercial Licenses
        - Hard to understand
        - bloated with exits and safety valves
    - MIT License
        - X11 License (other name)
        - Provides "as is" leave me alone if something blows up
        - rights to use copy modify merge publish distribute sublicense and or sell copies
        - need to include copyright info
    - BSD License
        - Allows copy change distribute (source + binary)
        - Copyright headers must be kept
        - Requires Berkley attribution
    - GPLv2
        - strong copy-left (applies in case of static and dynamic linking, not for just using)
        - distributing the results requires distributing modified sources
        - if you dont want to open the source
            - pay the IP holders (also after violating the license)
            - open source (well)
            - replace the thing you want to use or your stuff so no one can get your IP out of it
    - LGPL
        - GPL but allowed to use in dynamic linking
        - do what you want but if you change something you need to follow the license
    - Apache License v 2.0
        - Liberal open source software license
        - Business friendly
        - required redistributing NOTICE file
        - includes patent grant
        - can be sub-licensed (added code can be any license)

        - not re-licensing (allows to change license of existing code)
    - Not OSS
        - do-no-evil-license
        - beer-license
        - wtfpl-license
        - Facebook BSD + FB Patent License
            - React
            - RockDB
            - Not OSI approved
            - ASF does not allow it in Apache projects
        - Apache plus Commons Clause
            - Not OSI approved
            - Contradicts Apache License

- **Patents**
    - Some licenses contain a “patent grant”
        - License with patent grant:
            - ALv2
            - GPLv3
            - Mozilla Public License
        - Software Patents are allowed in the USA but not in the EU

- **Trademarks**
    - Name must be unique in your field (trademark classes)
    - Actively defend your mark
        - marks vanish if they are used often without attribution
    - Allow other people to build tools for your code (bla bla bla for Apache Foo)

Lost in Complexity

- **Software crisis** (term coined in 1968)
    - nothing really changed since then except that it is now a global problem and systems are more critical
    - Projects running over-budget
    - Projects running over-time
    - Software was very inefficient
    - Software was of low quality
    - Software often did not meet requirements
    - Software was never delivered
    - Projects were unmanageable and code difficult to maintain
    - maybe even more important now than it was back in 1968 as everything depends on software

- **Exogenous vs endogenous complexity**
    - Exogenous: defined by problem, domain, context
        - e.g. compare power plant management to customer service
    - Endogenous: defined by implementation, model, organization
        - software framework, testing,..

- **Why has complexity risen?**
    - distributed system
    - increasing complex external dependencies

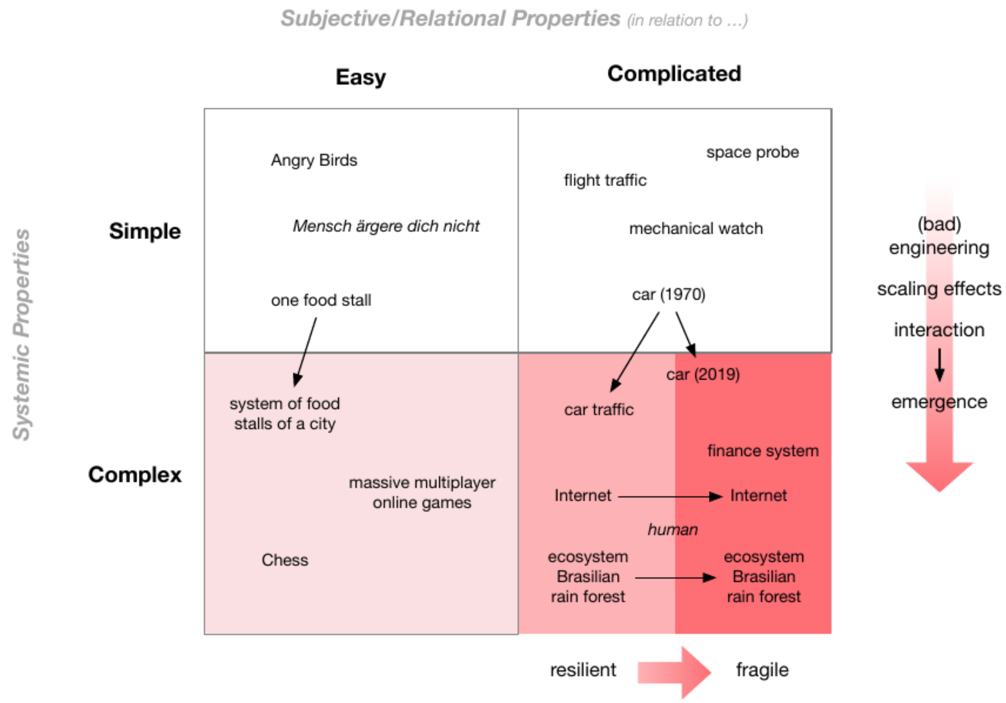
- **Consequences of increased complexity**
    - as ICT is a techno-social system
        - it enables nearly all important societal systems (e.g Health information)
        - it is itself dependent on most societal systems
    - there exist circular dependencies - power plants go out - communication goes out - communication needed for repairing the power plants

- **What is a System**
    - components - interaction parts, actors, input or interaction with other systems, environment
    - set of things, people, cells, molecules,..... interconnected in a way that they produce their **own pattern of behaviour over time**
    - systems have **defined borders** (what is part of the system and what is not)

- **System Principles?**
    - Stocks & Flows - Flows (trends) are more enlightening than stocks (counts); the measurement of the state of something is static at a point in time (a stock). flows change the value of that stock. you only change the state or value of the stock by influencing the flows.
        - compare with bank account - you can only change the total amount by changing the flow (how much you earn or how much you spend)
        - https://medium.com/natural-leadership/software-engineering-metrics-part-3-understanding-stocks-and-flows-71b2b859d992
    - Feedback Loops
    - Emergent Behavior - An emergent behavior is something that is a nonobvious side effect of bringing together a new combination of capabilities—whether related to goods or services.
    - Path- (History-) Dependence
    - Catalog disagreements (Any interesting system is sufficiently complex that different people will describe it differently)
    - Archetype:
        - describes personality types of developers???

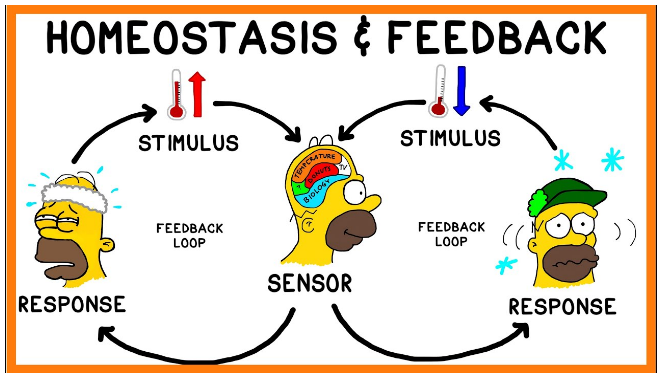
- **Wicked Problems**
    - no definition on what a wicked problem is
    - not a simple/easy problem
        - simple - one task, one role - systemically

- easy - for whom, depends on your knowledge



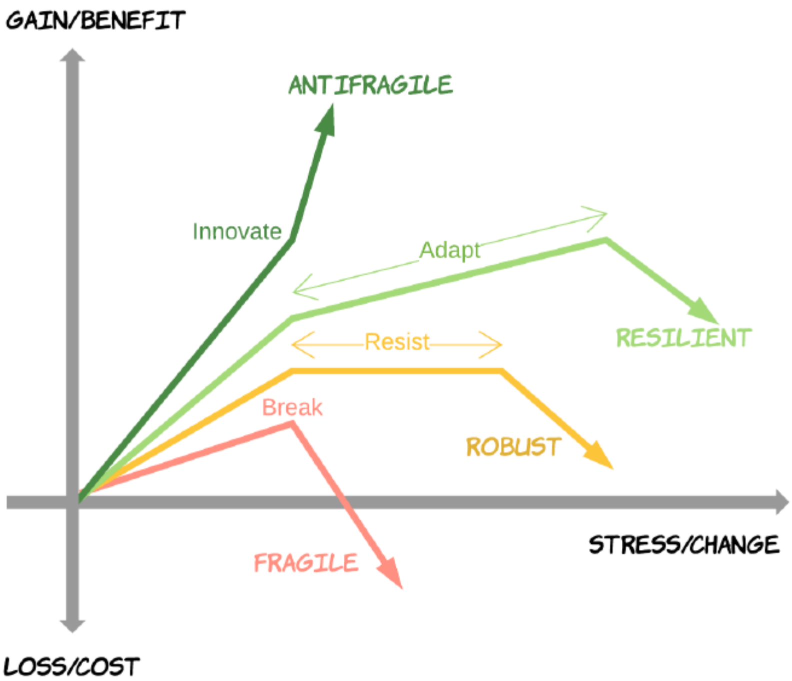
- no perfect solution
  - solution is stopped when resources run out
  - solution is good enough or better than before
- unique
  - no trail and error
  - one-shot operation

- **Control and prediction**
  - Predictability
    - Attractor state = systems always ends up here
      - e.g. a pendulum always goes through the middle
      - an attractor is a set of states toward which a system tends to evolve
      - Homeostasis - tendency to resist change to stay stable
        - e.g. temperatur control in the body



      - attractors show different stability to perturbation
    - In dynamic, complex systems there is no long term predictability

        - management schemes that predict will fail
            - compare communism
    - signs for failure (shift in attractor)
        - critical slowing
        - spatial resonance (pulses occurring in neighbouring parts of the web become synchronized)
  - Control over a system
    - Attractors stabilize a dynamic system because those points bring some predictability
    - Correct behaviour (steady state) in systems of systems
        - use system metrics
        - look from the outside on the entire system, not only on components
    - Some things cannot reasonably be controlled e.g. external modules, AI
  - Fragility, robustness, resilience
    - Many human-made systems are fragile as they did not have enough time to evolve (like natural ecosystems)
        - simplicity is a choice, complexity is your fault
    - resilience: what to do when everything goes downhill
        - resilience in software through e.g. graceful degradation (keep most important things running => shut down the rest)
    - fragility: how easy is it to influence correct behaviour
    - robustness: how many errors can a system tolerate



Bilgin Ibryam. Software: From fragile to antifragile. (2016)

- **Complexity in software**
  - Increase
    - Scaling (e.g. more components)
    - Interconnection (between systems)

        - Feedback loops
        - Speed
        - Number of stakeholders (forks) / users
        - design by committee
        - Software bloat and dependency madness
    - Decrease
        - small focused code
        - few dependencies
        - clean design made by few people
        - Compartmentalize, decouple
        - documentation and formal specification (of e.g. interfaces, protocols,..)
        - stateless programs (functional programs)
        - coding guidelines
    - Behaviour
        - follow attractors
            - self-healing
            - love randomness (small variations)
            - tipping points
        - multiple causes lead to failure (simple cause and effect analysis does not help) -> defect components cannot be changed easily
        - sometimes unexpected
    - How to deal with it
        - What does not work
            - Trial and Error (won't get you far)
            - Ignore it (abstraction)
            - Rationality - try to understand and predict
            - command and control - top down management
        - What does work
            - reduction to few criteria
            - Intuition
            - evolutionary adaption
            - sense and respond
            - resilience building - failure as standard procedure, not as catastrophe
            - split into sub-parts
    - Chaos engineering
        - some mechanism in the system randomly attacks the system (in production) to test its capabilities
            - e.g. Netflix Chaos Monkey
                - stress test in production

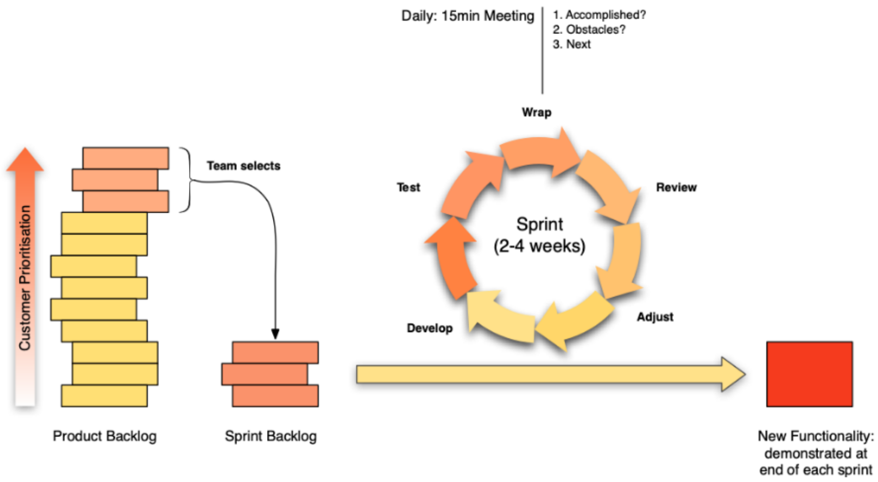
Agile Software Development in Corporate Environments

- **Software development strategies**
    - PDCA

Process	Waterfall Development	Iterative and Incremental	Agile Development
Measure of Success	Conformance to plan	→	Response to change, working code
Management Culture	Command and control	→	Leadership/ collaborative
Requirements and Design	Big and up front	→	Continuous/emergent/ just-in-time
Coding and Implementation	Code all features in parallel/test later	→	Code and unit test, deliver serially
Test and Quality Assurance	Big, planned/ test late	→	Continuous/concurrent/ test early
Planning and Scheduling	PERT/detailed/fix scope, estimate time and resource	→	Two-level plan/fix date, estimate scope

  -
  - Agile Manifesto
    - Individuals and Interaction over Processes and Tools
    - Working Software / Business Results over Comprehensive Documentation
    - Customer Collaboration over Contract Negotiation
    - Responding to Change over Following a Plan

- **Agile Practices**
  - Process-oriented (like a process description)
    - SCRUM
      - team size < 10 people (5-10)
      - customer tightly integrated
      - realistic estimations
        - user stories -> backlogs -> planning poker (team estimations)
      - splitting up the code or tasks (no collective code ownership)
        - helps making development more efficient
          - is in a way the removing of redundancy (in knowledge)
            - not everyone needs to know how a certain thing can be done
            - however this can be a problem in the long run (someone leaves the company)
            - short-term efficiency < long-term stability



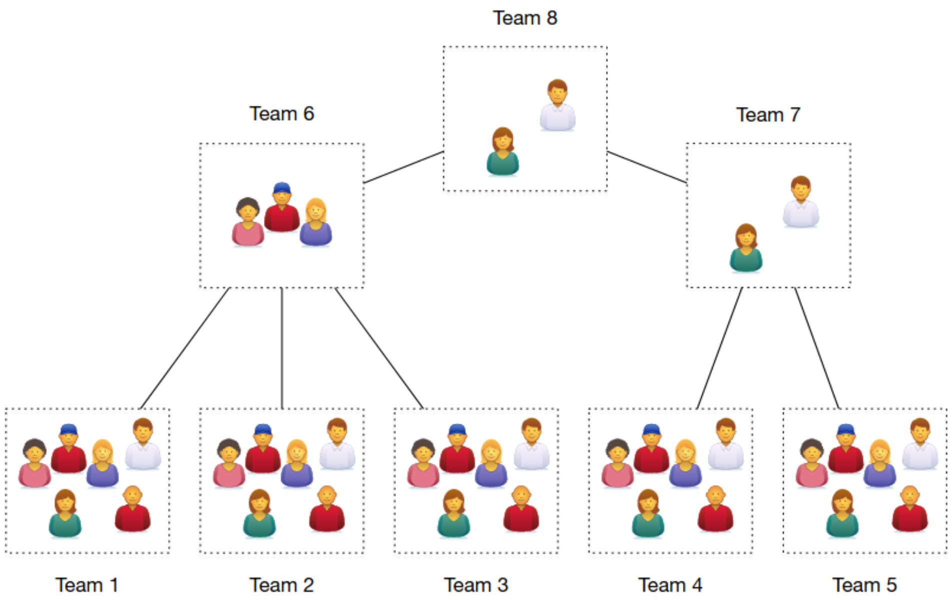
-
- product backlog: is taken care of by the product owner that interacts with the team and the customer
- sprint backlog: what is to do in the next sprint
- management usually wants predictability

■ Software Kanban
- not with iterations but as a whole iteration
    - Continuously taking stuff from the backlog
    - stuff gets added to the backlog continuously
- tries to resolve bottle necks of SCRUM (too much to do at once or not enough to do - missing resources)
- uses real-time metrics
    - average lead time
    - cumulative flow diagrams: cycle time

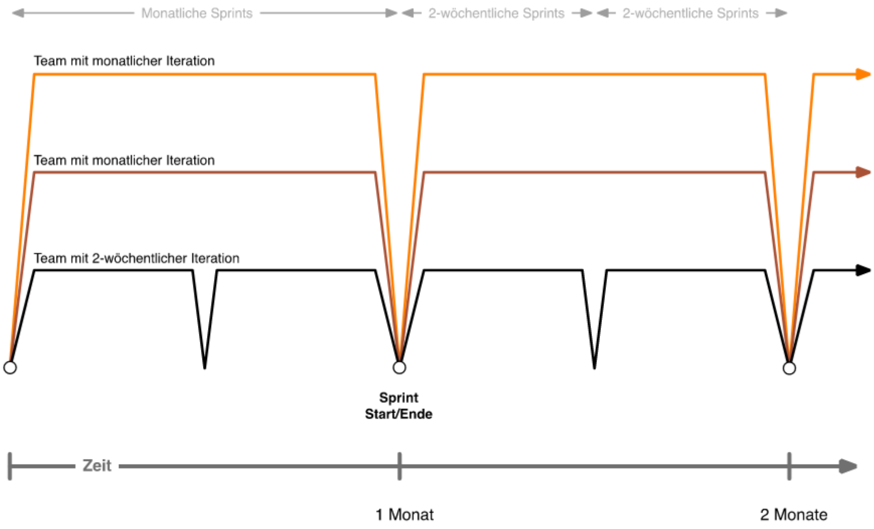
- Methodical building blocks (like a toolbox with practices)
    - XTreme Programming
        - Communication / Collaboration / Architecture
            - Planning Game
                - Release Planning
                    - Customer collects user-stories (story creation)
                - Iteration Planning
                    - User-Stories -> Tasks
            - Metaphor
                - Each chunk of code get own name, so that the customer (who is part of the xp team) can understand them
            - Simple Design
        - Process
            - Small Releases
            - Pair Programming
                - bad decisions and mistakes caught
            - Collective Code Ownership
                - everyone is responsible for the code base

            - no separation of knowledge (think about someone leaving)
        - 40-hrs Week
        - On-Site Customer
    - Technical
        - Coding Standards
        - Testing (Test-Driven-Development)
        - Continuous Integration
        - Refactoring

- **Challenges in Corporate Environments**
    - Problems
        - multiple teams
        - many devs
        - large projects vs. perfective maintenance
        - prioritization
        - projects are partly internal and partly external
        - management levels
        - budgeting and planning cycles
        - reporting and controlling
    - the agile approaches work well for single teams but if there are like 20 scrum teams the product owners & teams need to coordinate
        - core of agile software
            - flexible self-organisation of teams
            - lean and efficient work in small teams with short iteration cycles
    - Scrum of Scrums

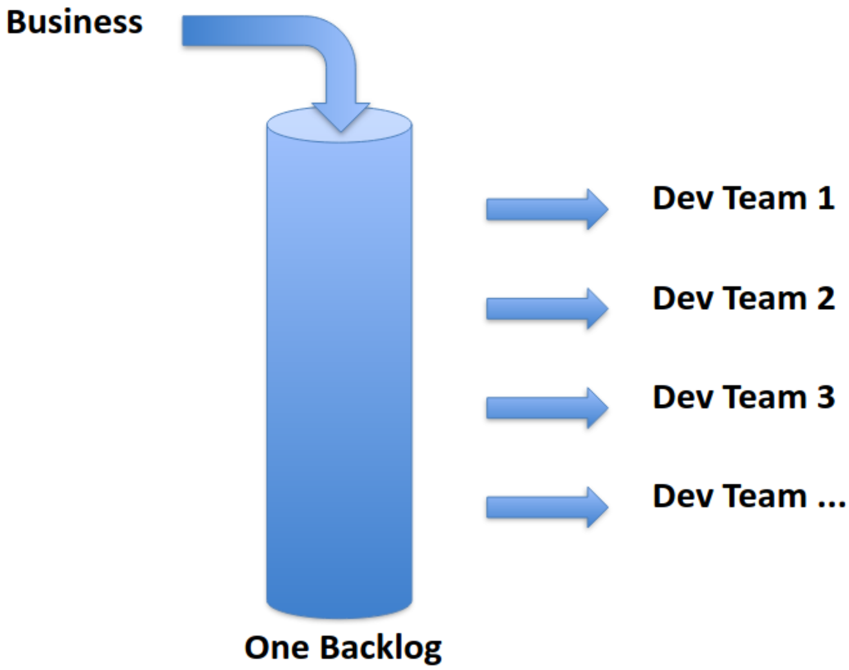


        -
        - e.g. product owners build their own scrum teams
        - all teams in an organization must have the same sprint synch (if the need to coordinate)



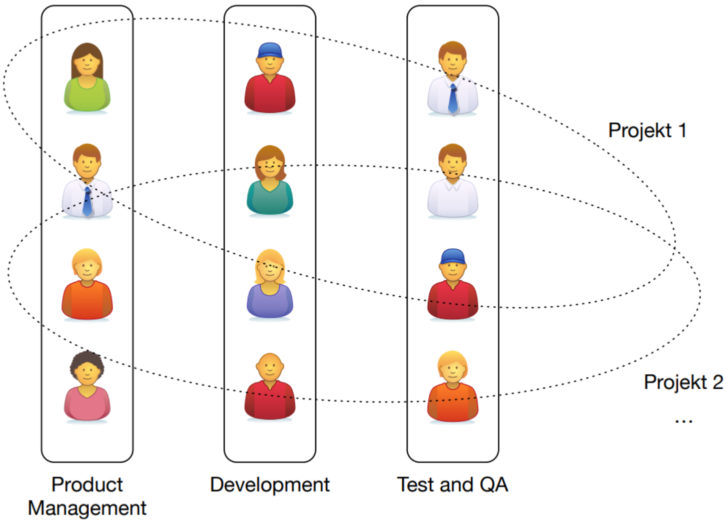
- else one team is working while the other is planning and vice versa
- everything is ready at same time
- shifting people from team to team is easier (think of security experts… that are not part of a fixed team but work for one sprint with a team)

○ Factory Approach



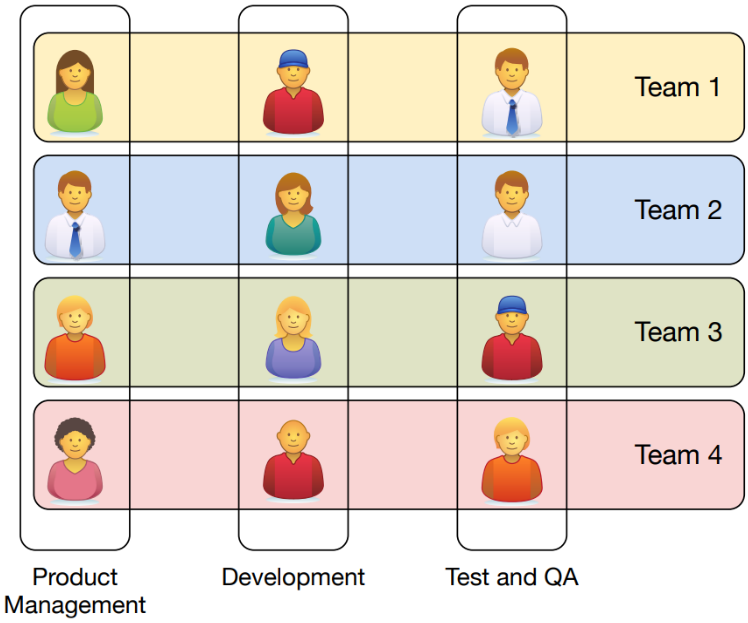
- if all teams are the same & products are quite small

- **Team Organization**
  - Functional Silos



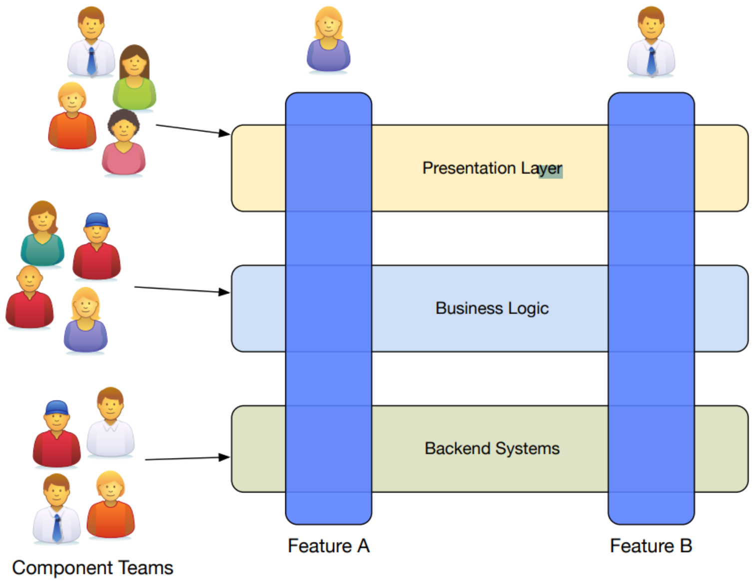
■

○ Agile Teams

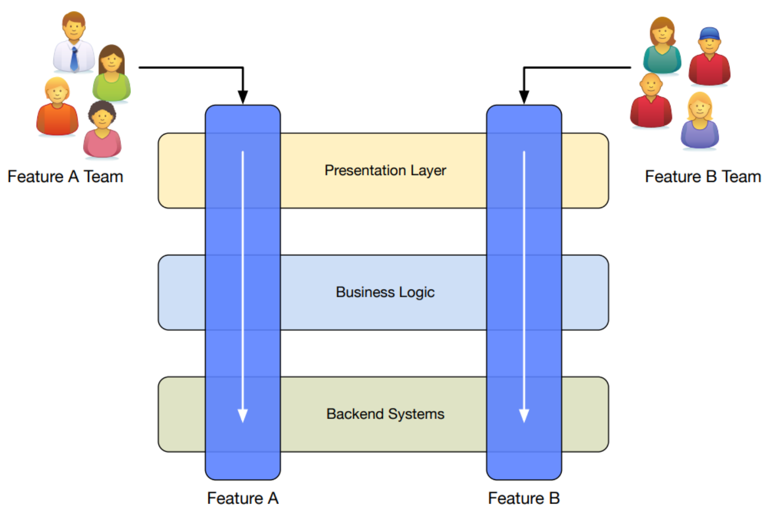


■

○ Organization Following Component



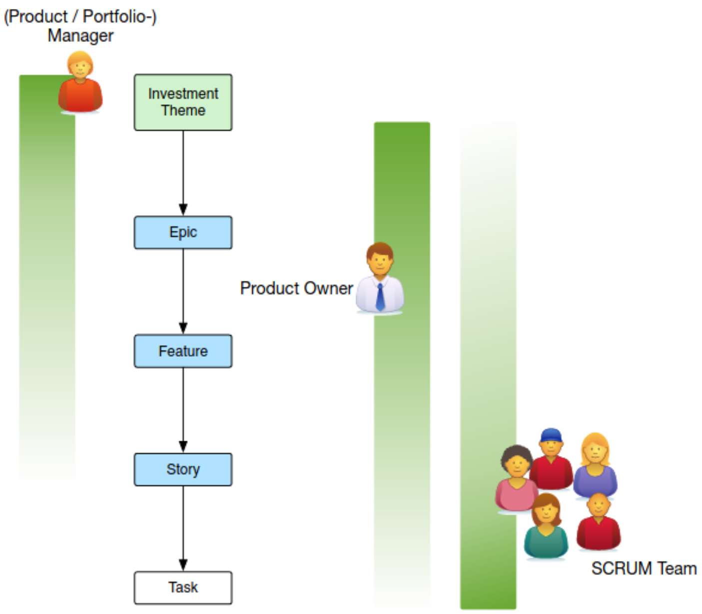
- Agile Team Organisation (Following Feature / Processes / Services)



- **SAFE Framework**
  - scaled agile framework
  - epic vs user story
    - epic consists of multiple user stories (complete feature) (user management page)
    - user story is “just” a part of a feature (deleting user)
  - enabler vs. stories
    - enabler - work that cannot be attributed to a story but needs to be done to enable working (e.g. CI-pipeline setup, refactoring, setup of development environment)
    - story - need to have business value in the end

- **Customer Responsibility**
  - product owner
  - roles and process responsibilities have to be clarified

    - lack of clear roles and respos is often the main factor for failure in agile projects

- **Requirement engineering**
    - User stories - as WHO I want WHAT so that WHY
        - small - one card
        - parts
            - acceptance criteria
            - role & description
        - needs to be estimable (not in absolute values but in relative values that represent complexity)
            - Scrum poker
                - Fibonacci numbers (bigger numbers get harder to categorize - bigger steps)
                - what is the difference between 10 or 11? hard to say
                    - difference between 1 and 2 is easy
                - also the uncertainty gets bigger
                - avoid anchoring (looking at what other people say it takes to finish it) by letting everyone choose in private
        - INVEST
            - Independent
            - Negotiable
            - Valuable
            - Estimable
            - Small
            - Testable
    - Roles and "Stories"
        - 

- **Transparency**
    - Burn Down Charts
    - Agile Metrics

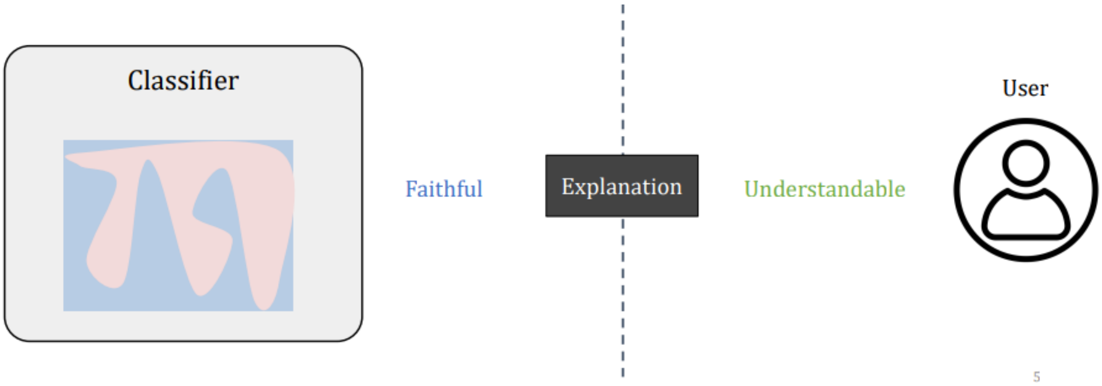
        - Burn-Down-Charts, process-flow visualization, cumulative flow diagrams
        - Velocity (items per iteration), velocity per work type, cycle time (average completion time of one item), identification of bottlenecks (queue length), defect rates
        - Metrics and performance indicators are sometimes a bad thing
            - e.g. metric that measures productivity of a team in their completed story points -> could lead to bad effects
                - teams just do the easy things that are quick and bring them points fast instead of doing all the work that should be done
                - teams start overestimating to pump up the indicator values
                - teams could keep estimating as before, but work with less quality to keep up
                - -> simplistic metrics are a problem
    - Progress / Cost and Budget
        - Reporting of progress is often difficult
        - Progress according to defined scope is comparatively easy
        - Is development in budget?
            - Time recorded
            - Cost per day per employee
            - Internal external members
            - other cost (licenses)
        - Actuals vs planning (who does what? opposite of agile)
        - administration task for dev teams become all but lean and self-organised
    - Challenges and Risks
        - Lack of trust
        - Lack of transparency
        - Cost/backlash of transparency
        - Complexity of architecture and systems
        - Team structure not clear enough (or still focused on silos)

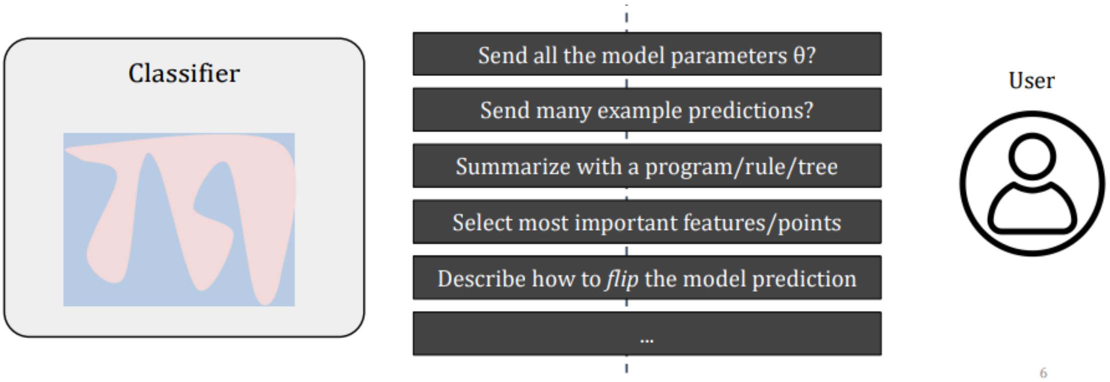
Explaining Machine Learning Models

- **What are machine learning models used for**
    - Vulnerability Detection
    - Semantic Code Labelling (Label methods based on instructions in the methods)
    - Performance Regression Detection
    - Testplan Quality Assessment
    - Taint Propagation Detection (privacy leak detection, how data flows within the program)

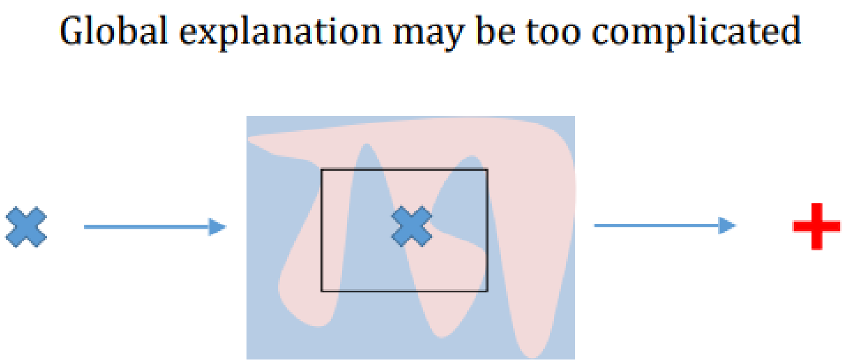
- **What is an explanation?**
    - Definition: Interpretation description of the model behaviour (in a target neighborhood)
    - Help understanding WHY a machine model has come to some result

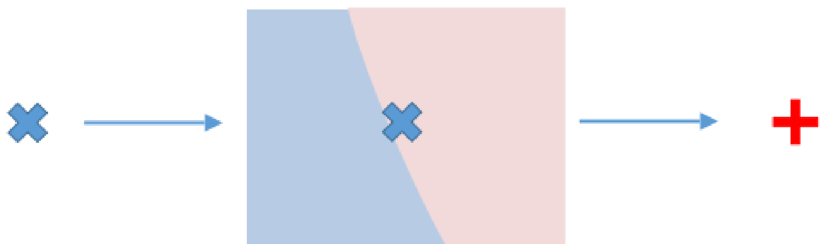
- e.g. why does the model think there is a security issue?





- Global explanation vs local explanation





- Global
  - explaining and understanding the whole model behaviour
  - shed light on big picture biases
  - help check if model at high level is suitable for deployment
  - usually it is easier to get only an area of the input

                - used more as a debugging tool
            - Local
                - explain individual predictions
                - help unearth biases in the local neighbourhood of a given instance
                - help check if individual predictions are being made for the right reasons

- **Counterfactual explanations**
    - Counterfactuals: alternate “world” where prior circumstances are changed to see what the consequences of this change would be
        - e.g. I slipped and fell on the rainy street and broke my leg. - Counterfactual: If today wasn’t rainy, would I still have slipped and broken my leg?
        - e.g. If you had called genSimple instead of genHandle, your code would not be classified as causing a performance regression

```
-   $store_handle = await SomethingStore::genStoreHandle($vc);
+   $store_handle = await SomethingStore::genHandle($vc) ;
                          SomethingStore::genSimple($vc)
+   ... other code ...
```

        - demonstrate how the model’s prediction would have changed had the program been modified in a certain way
        - what-if questions
    - Problem statement

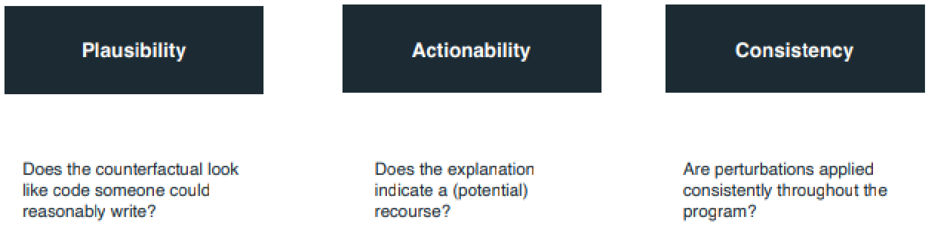
Terms:

P: the original program, which is the input to the model **M**

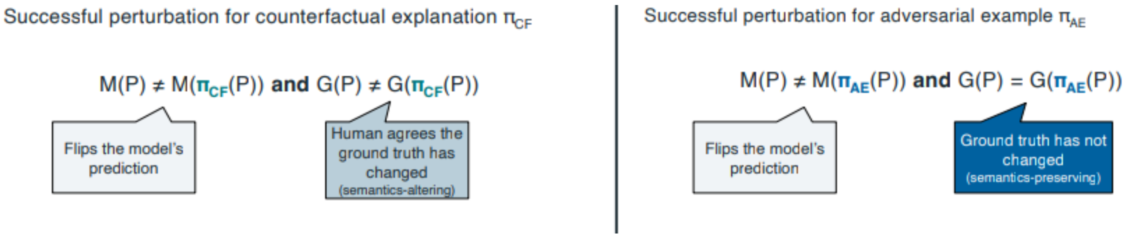
π: a perturbation to the program

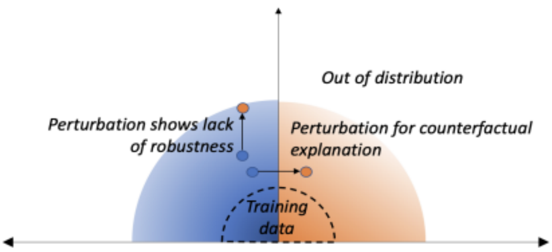
G: ground truth outcome

        -
        - Ground truth: the value the output should have; that is the reality you want your model to predict
            - e.g. we know that the data we gave to the model results in a performance regression - compared to the output -> is the model able to detect the regression?
    - Plausibility - Actionability - Consistency



- ■
- ■ Plausibility (naturalness)
    - e.g. the model says the diff is the problem instead of the root cause - the expression -> this is a bad counterfactual
    - when perturbed inputs are out-of-distribution
        - model predictions can be unreliable
        - counterfactual explanation is uninformative
        - user does not believe explanation

- **Robustness vs. Counterfactual Explanations**
    - Adversarial examples (from robustness research)
        - robustness - how stable is the prediction model against changes - e.g. Panda
        - semantics-preserving
        - the input is changed slightly so that the model classifies it as a wrong output, but humans do not see the difference
            - e.g. Panda example
    - Counterfactual explanation
        - humans agree that the ground truth has changed
    - 

    - 

- **Where did my model go wrong**
    - Challenges
        - High-dimensional input space (many vars)
        - Opaque models (want to see inside but usually blackbox)
        - Manual Hypothesis Testing not scalable
    - Misprediction diagnoser (MD)

            - Goal: explaining ML models by systematically identifying subsets of input space on which the model mispredicts

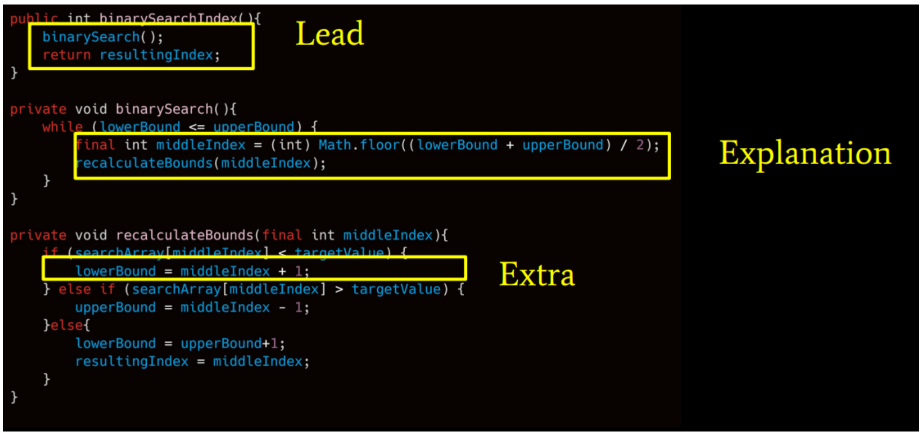
How to achieve clean code

- **Technical Debt**
    - A shortcut that helps you in the short term but will cost you more in the long term
    - Technische Schulden

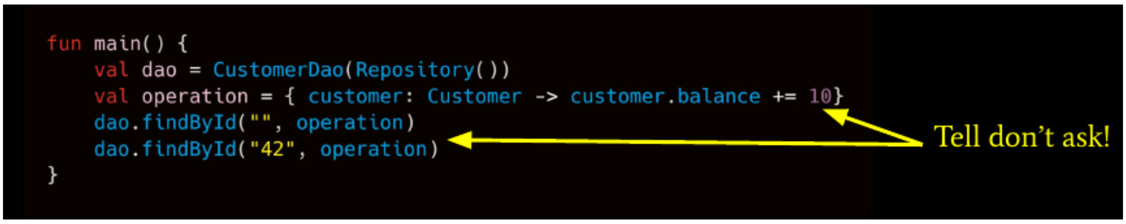
- **Clean code**
    - What is clean code
        - Readable - Simple
        - Tested
        - Practiced - it is a mentality
        - Continuously refactored
    - Allows you to
        - change high-level functionality and low-level implementations even in late stage of project
        - postpone harder decisions to later stages of a project
        - makes the basis for good architecture and design
    - Developer Maturity Levels
        - L0 - Black - Interest
        - L1 - Red - Attitude
        - L2 - Orange - Fundamentals
        - L3 - Yellow - Testing
        - L4 - Green - Automatization
        - L5 - Blue - Deployment & Architecture
        - L5 - White - Awareness of CCD Values

- **Naming**
    - methods should do the thing you expect them to after reading their name
    - 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

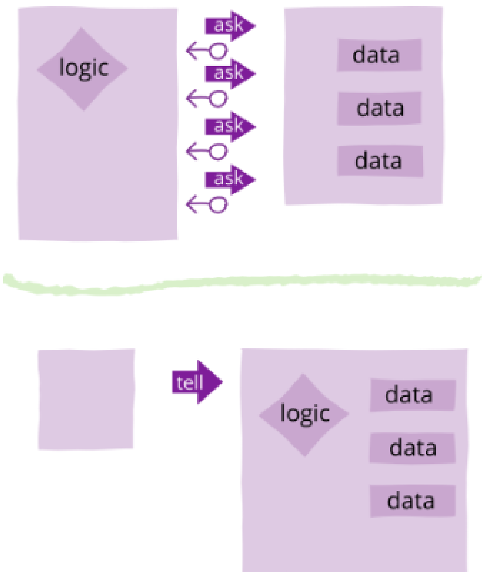
- **Functions**
    - should do one thing only
        - one level of abstraction
        - one level of indentation (loops, branches)
    - Build your functions like a newspaper article
        - Lead Paragraph = public interface of class
            - get the most important information across in the thing everyone has a look at first
        - Explanation = high level “routing” (call stack)
        - Extra = low-level implementations



- Use a max of 3 arguments in your method's signature, best none
  - arguments are hard to interpret
  - argument are different levels of abstractions
- Beware of boolean arguments - they do more than one thing
- Tell - don't ask - e.g. search user in a list - tell the object that it should give you something instead of asking for the find and doing it yourself



- Tell - Dont Ask



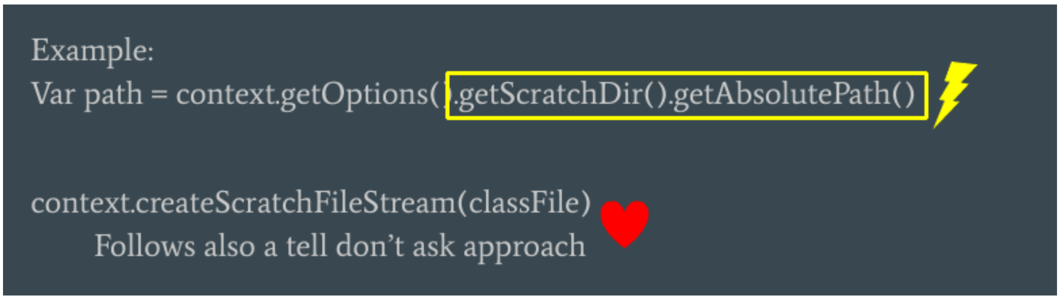
- **Error Handling**
  - Make exception names clearer, more concise and part of your domain
  - Putting "Exception" in the name is not very helpful
  - Always
    - write try - catch - finally first

    - Never
        - Pass or return null (use Optionals, Null-Objects, Empty Lists instead)
        - Hide behind errors
        - use errors to influence the control flow
        - destructive wrapping (pass causing exception instead)
    - Either
        - Log XOR throw
        - Handle it XOR pass it on

- **Comments**
    - People don't read comments - neither do compilers
    - lie, because only code contains the truth
    - do not make up for bad code
    - Good Devs may not write good comments
    - Consider if it is comment worthy or should be refactored
    - Don't use comments as documentation
        - too specific
        - too detailed
        - too quickly outdated
    - Use documentation techniques
        - meaningful interface documentation (JavaDoc)
        - Mock Press Releases
        - Versioned documentation (readme)
        - API documentation (SWAGGER)
        - Documentation as part of your tests (Spring)

- **Classes**
    - Step down rule
        - List of variables
            - Public static constants
            - private static vars
            - private instance vars
            - (public var)
        - Public functions
            - Constructor
            - Private functions called by a public function right after the call
                - keep callee and caller close together
    - Name hints for unfortunate aggregations (bad cohesion - class should focus on one thing)
        - e.g. managers, processors, super usually do more than one thing and have multiple responsibilities
    - One responsibility
        - Comply to needs of ONE stakeholder group
        - have many small classes (single responsibility)
        - not few large doing multiple things
    - Dependency Inversion Principle
        - Depend upon abstraction - not implemenation

        - Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen. - Wikipedia
        - Open Closed Principle - open for extension, closed for modification

- **Objects & Data Structures**
    - Make it hard to wrongly use your object
        - Define constructors adequately
        - maybe overload them
        - don’t require setter to be called after instantiation
        - user creational patterns for complex instantiation
            - factory
            - builder
            - prototype
    - Law of Demeter - a method f(x) oc Class C should only call
        - C
        - on object created / passed by / to f(x)
        - Instance objects of C
        - 

        - makes explicit what you are doing - tell don’t ask

- **Clean Test Code**
    - 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. You shall not
        - write production code until you have written a failing unit test
        - write more of a unit test than is sufficient to fail (dont add unnecessary stuff to your test)
        - write more production code than needed to pass the currently failing test
    - Designing a unit test
        - Build up test data
            - have enough data
        - operate on data
        - check that operation yielded expected result
            - only one assert per test

                - only one thing per test
    - Clean Code !== Clean Test Code
        - One functions contains all relevant aspects
        - keep the reader in the test function
        - test methods should be self contained
        - accept redundancy if it supports simplicity
        - dont bury critical information
        - test methods are never called so use descriptive names

- **Tools for clean code**
    - Formatter / Checkstyle
    - Static Code Analysis
    - Continuous Integration

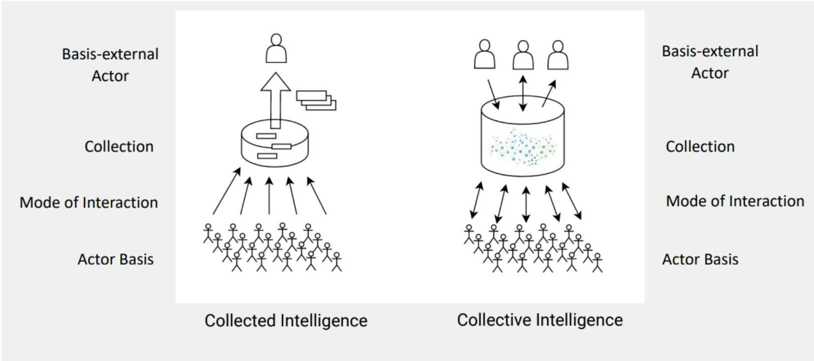
Software Architecture for Collective Intelligence Systems

- **Collective Intelligence**
    - Group intelligence that emerges from group collaboration, collective action and competition of individuals
    - Examples:
        - Swarm formation of drones
        - Intelligent routing of traffic
        - online social network + co creation platforms
    - It is achieved by hybrid systems in which humans and computers interoperate and complement each other
    - It has a potential for creating highly effective collection of hard to access knowledge
    - used for social web / media and social computing

- **Collective Intelligence Systems**
    - Definition: Collective intelligence of connected groups of people by providing a web-based environment to share, distribute and retrieve topic-specific information
        - socio-technical multi-agent system
        - mediates human interaction
        - provides support for distributed cognitive processes
        - driven by users who contribute content
        - distribution of consolidated info back to the users (give and take)
    - Examples
        - Social network services (Facebook, Twitter, Snapchat,...)
        - Media / Content Sharing (YouTube, Soundcloud,...)
        - Knowledge Creation (Wikipedia, Stack Overflow, Fandom,...)#

- **Nature of Intelligence**
    - Steps
        - Collection
        - Processing and Exploitation
        - Analysis and Production
    - Foundations

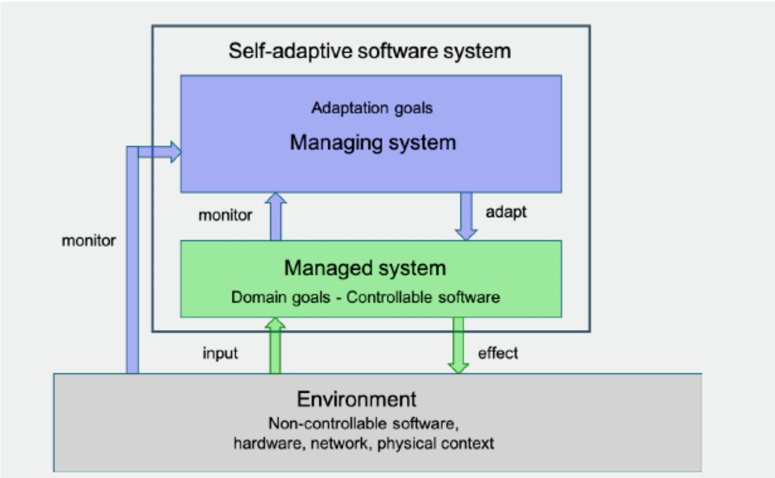
- Data needs to be processed to become information
- Information needs to be compared to other information to draw conclusions
- Intelligence arises from information that is related to environment and past experiences
- Intelligence allows prediction and planning

- Collected Intelligence vs Collective Intelligence
  - 

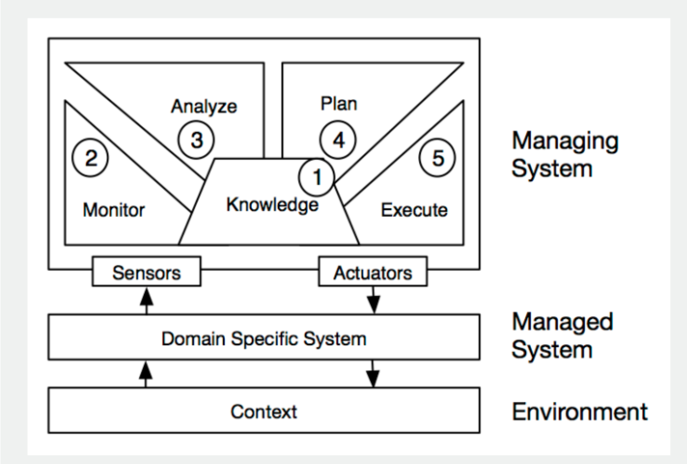
  - Definitions:
    - Actor Basis: Group of agents who are the data source
    - Collection: Organized aggregate of structured/unstructured data and information
    - Basis-external Actor: Agents have access to the collection and are not members of the actor basis
    - Intelligence Beneficiary: Group of agents who gain intelligence from the collection

- **Key Stakeholders & Benefits of CIS**
  - Users
    - effective bottom-up communication
    - awareness (new developments, changes, trends)
    - building upon content (knowledge) of others
    - be able to work on a common topic (that needs contribution from dispersed users)
  - Platform providers
    - Network effect (more people use it - so more people use it) - more valuable over time
    - Building up an active user base is time intensive and hard to replicate by competitors
    - Data collected is valuable

- **Foundational Concepts of CIS**
  - Coordination Models for Swarms
    - Swarm Formation - e.g. birds
      - direct interaction
      - communication is in a direct way
      - collective movement
      - global: stay in the group
      - local: do not hit other birds

            - global attraction but local repulsion
        - Stigmergy - e.g. ants
            - indirect interaction over the environment
            - communication is in an indirect way (environment)
            - dynamic construction of trials (collective foraging)
    - Self Adaptation
        - a way to deal with uncertainties
        - uncertainties affect qualities
        - uncertainties are difficult to anticipate
        - idea
            - gather info at runtime and use it to reason about itself and change the plan accordingly
        - Dimensions of Uncertainties
            - Location (what is effected from uncertainty)
            - Nature (what causes the uncertainty; is it due to imperfection of knowledge or due to inherent variability)
            - Level / Spectrum (how uncertain am I)
            - Emerging Time (when is it acknowledged or appeared)
            - Sources
            - Model uncertainty
            - Adaptation functions uncertainty
            - Goal uncertainty
            - Environment uncertainty
            - Resource uncertainty (are resources available, do resources change)
        - 

        - Example MAPE(-K) Model

- 

- 1 Knowledge - e.g. logs, rules/policies, metrics, topologies,...
- 2 Monitor - collect data
- 3 Analyse - analysis and reasoning on data from 2
- 4 Plan - creates workflows depending on analysed data and goals
- 5 Execute - execute workflows

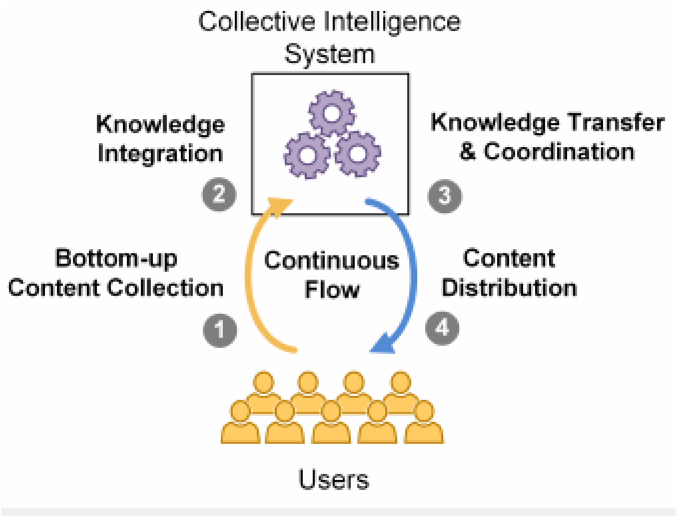
  - Socio-technical Systems
    - interaction between humans, machines and the environmental aspects
    - are composed of 2 sub-systems
      - social system - humans with knowledge, skills and relationships who participate content
      - technical system - technology and technological artifacts to perform tasks to the overall purpose

- **Architecting CIS**
  - Approaches to the Architecture of self organising systems
    - Multi-Agent Systems (MAS)
      - Socio-technical system where agents interact with each other and environment to satisfy their goals
      - Agent-Oriented Software Engineering (AOSE)
      - Environment architectures (Environment-mediated Coordination)
        - Coordination Models
        - Environment
          - coordination infrastructure
        - Artefact
          - Coordination medium (abstraction of environment)
        - Stigmergy
    - CI-adapted Coordination Models
      - feedback loops, self organization and self adaption
    - Software Architecture
      - is the set of structures needed to reason about system (software elements, relations, properties,...)

        - is the set of architectural design decisions - consists of:
            - rationale - reason behind design decision
            - design rules - what is allowed in further designs
            - design constraints - what is not allowed in future design
            - additional requirements
        - Standard-based software architecture frameworks and reference architectures
        - Concepts
            - Environment
                - Every system is situated in the context of a defined environment
            - Stakeholder
                - Individuals, groups, orgs, define a system's purpose and have interests in a system
            - System / Stakeholder Concern
                - Specific interest of stakeholders in a system
- ISO-Standards
    - 42020 - Architecture Processes (Governance, Management, Conceptualization, Evaluation, Elaboration, Enablement), Information Flows
    - 42010 - 2011: Architecture Description Language (ADL), Architecture Framework, Correspondences, Architecture description
        - Architecture Description
            - documents one possible architecture (design decisions)
            - identifies stakeholders and their concerns
            - describes needs
        - Architecture View
            - describes system from a chosen viewpoint
        - Architecture Viewpoint
            - promotes reuse of best practices
        - Correspondences
            - express architecture relations
        - Correspondence Rules
            - governs correspondences and enforces relations within architecture description
        - Architecture Framework
            - Defines conventions, principles and common practices
            - Specifies
                - addressed concerns
                - stakeholders having those concerns
                - architecture viewpoints that frame those concerns
                - correspondence rules integrating those viewpoints
        - Architecture Description Language (ADL)
            - Form of expression
            - Specifies
                - addressed concerns

            - stakeholders having those concerns
            - model kinds
            - any architecture viewpoints
            - any correspondence rules
    - Viewpoint
        - Context Viewpoint
            - Designs CI-specific system capabilities and defines models for new CIS construction and capture of design decisions
            - Stakeholders
                - Architect
                - Owner
                - Actors
            - Concerns
                - Usefulness
                - Perpetuality
            - Model Kinds
                - MK1 - As-Is Workflow
                - MK2 - Stigmergic Coordination
                - MK3 - To-Be Workflow
        - Technical Realization Viewpoint
            - CIS realization and defines models to model collective knowledge, the aggregation of data and stigmergy-based dissemination of knowledge
            - Stakeholders
                - Architect
                - Owner
                - Builder
                - Actor
            - Concerns
                - Data Aggregation
                - Knowledge
                - Dissemination
                - Interactivity
            - Model Kinds
                - MK1 - Artifact Definition (artifact structure, linking, and operations to interact with artifact content)
                - MK2 - Aggregation (describes actor activities, logging, data aggreation)
                - MK3 - Dissemination
        - Operation Viewpoint
            - CIS operation startup and defines models to identify initial content, actor groups, and measures for CIS aggregation and dissemination performance.
            - Stakeholder
                - Manager
                - Analyst
            - Concerns
                - Kickstart

        - Monitoring
    - Model Kinds
        - MK1 - Initial Content Acquisition
        - MK2 - CI Analytics

- **CIS Concerns**
    - Environment-mediated coordination and indirect communication with feedback loop (2,3)
    - Information Aggregation (1)
    - Knowledge Dissemination (4)
    - Perpetual Feedback Loop
    - 

- **Using CI during a Software Engineering Project**
    - Internal perspective
        - Coordination of collective development efforts
        - Awareness about progress (changes, issues)
        - Discoverability of locally distributed knowledge and software artifacts
    - External perspective
        - thriving on the work and knowledge of communities instead of reinventing the wheel
            - open source
            - going platform / ecosystem
            - accessing quality-assured knowledge of crowds
    - CIS helping your SE Tasks
        - Issue Tracking (internal / external) - Jira,...
        - Knowledge Management (Internal / external) - Confluence,...
        - Programming Q&As (External) - Stack overflow,...
        - Code Review Tools (Internal / external) - gerrit, crucible,...
        - Container registries (internal / external) - docker hub
        - Extension portals (external) - rubygems.org, vs marketplace
        - Collaborative Code repositories (external) - GitHub
        - Digital Distribution and Updates (external) - App Stores, Steam,...

- **Key Design Assumptions**

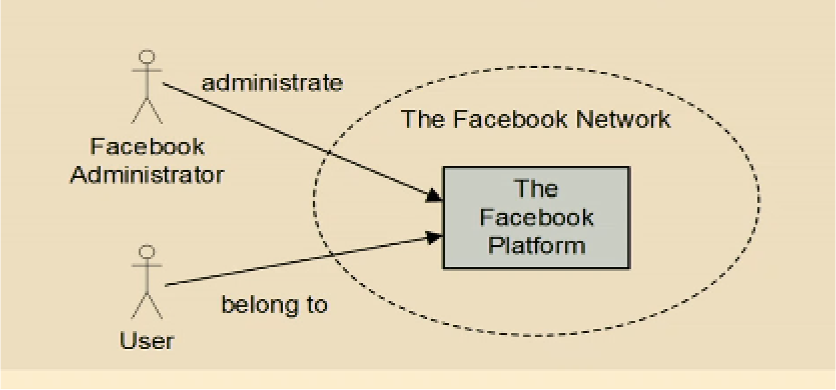
- User-driven Content Generation
- Big Data Processing and Management
    - Issue: a lot of data is needed to be processed
- (Real-Time) Data Analysis
    - Analysis Paralysis -> too much data, you cannot find something useful
    - need assistance to see what is there
- Scalability
    - pricing concerns
    - architecture dependant
- High Availability (24/7)
    - or only in core hours
    - but then it should be stable

- **Common Misconceptions**
    - We are in a perpetual beta, so we just start with the development and do the system design as we go
        - WRONG, though a well-rounded system architecture of the “core” system and its user-machine workflows is key
    - If we built it, they will come
        - if our system is cool, someone will use it
            - WRONG - a strategy for every initial user group is needed
    - Scaling has to be considered from the very beginning of the system design
        - WRONG - depends on the system design
        - if you go server-less you have the scaling given already
        - if you go on premise there needs to be more thought, but it depends on how much people will use the system
    - CICs utility and its ability to keep users engaged is related to using the right technology framework and libraries
        - WRONG - effectiveness depends on the ability of CIS to keep users engaged, also about content moderation, social aspects, privacy, security -> to a degree independent of the technology
        - e.g.: Whatsapp - belongs to Facebook; Facebook has privacy problems but still some people do not leave because more people use Whatsapp and this is the reason they do not want to leave (network effect)
        - Black Swan moments: Twitter -> Mastodon (because of Musk)

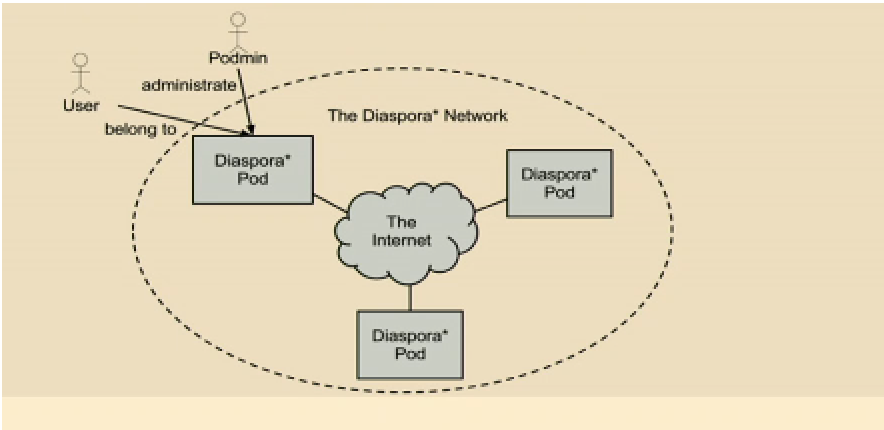
- **Success and Risk Factors**
    - Success
        - Choosing the right type of CIS
        - Appropriate set of CI design patterns
            - e.g. Youtube - got rid of down-vote button (only for video creators)
        - Provide low friction, easy to use means on contributing content
            - e.g. one-click-mechanisms
        - effective feedback mechanisms - which make users aware about activities of other users
    - Risk

        - CIS will not be used if it is not integrated in user workflow
            - if it is too complicated, people will not use it
            - design workflows according to natural flows
        - neglecting the user-base side
            - too strict too loose content moderation
        - cannibalization of user activity by other CIS
            - all people are somewhere already
            - platforms are trying to steal each other's user based
            - consider UX and UI
        - handling of security and privacy of user data
            - people are more sensitive now to privacy
- **Challenges**
    - Designing the right functional architecture
        - requirement elicitation of users needs and optimization potential
        - getting the basic workflows right
    - Perpetual beta
        - continuous delivery
    - Fostering an active community of contributors
        - users are scarce resource - competition
        - engagement (incentives, motivation)
    - Scaling
        - Big data and Machine Learning
        - Cloud computing
        - Global software dev
            - team around the globe - always someone that is live and working
        - Hyperscaler

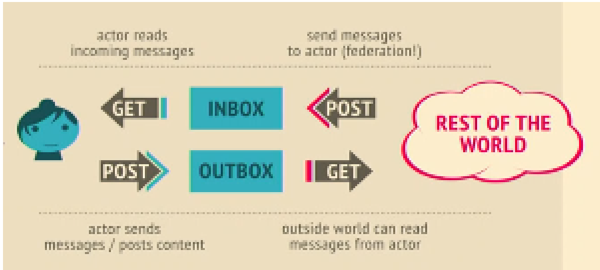
- **Centralized CIS vs Decentralized CIS**
    - Centralized
        - One Platform
        - One Provider
        - central admin, dev and content curation
        - Data in one single system
        - e.g. Youtube,...



    - Decentralized and federated CIS
        - Most are open source

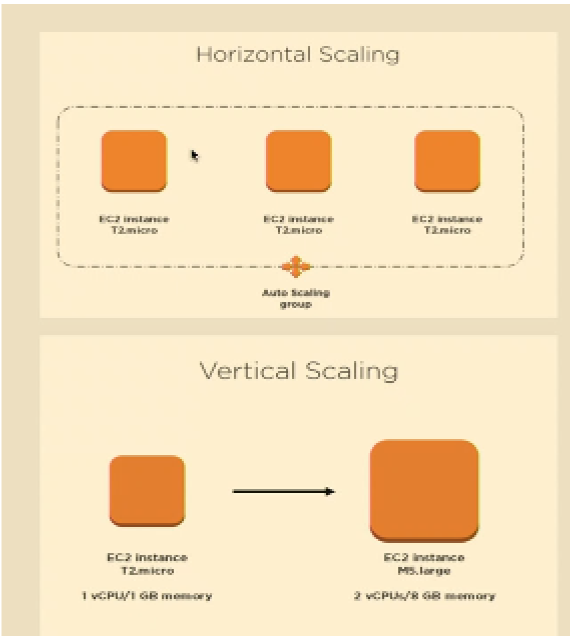
- Different nodes where instances of the systems are deployed
- Challenge:
    - A lot of different operators with different setups (server hardware)
    - quality differs widely
    - Examples: all FOSS
        - Mastodon (Twitter)
            - microblogging
            - nodes = instances (with own policies for privacy, content, moderation, ….)
            - ruby on rails - back, react.js - front, PostgreSQL, redis (caching)
        - PeerTube (Youtube)
            - content via web torrent
            - Postgres, redis, Express/NodeJS
        - Pixelfed (Instagram)
            - image sharing
            - tech: php, nodeJS, MariaDB / PostgreSQL, Redis
        - GNU Social
            - microblogging
            - tech: php, OStatus, XMPP
        - Diaspora (Facebook)
            - social networking service + personal web server (Unicorn)
            - diaspora network is build out of a network of individual diaspora system instances (pods)
            - tech: ruby on rails, unicorn, backbone.js
- 

- ActivityPub Protocol
    - Open Protocol
    - Based on Activity Stream and linked Data
    - Main integrative protocol for platforms in the Fediverse
    - Does
        - Communicate, follow, like with users and content on other instances / platforms that support ActivityPub

        - Does NOT
            - Discovery: no mechanisms for this -> need to use WebFinger URI e.g.
            - Simple: layered integration of W3C specs leads to verbose responses; difficult in handling
            - Certification: Platform decides if/how they follow the protocol (out of spec behaviour)
            - 

- Trade-Off Centralized / Distributed CIS
    - Going centralized or distributed comes with trade-off
    - Always one central node required
        - for quality control
        - finding other nodes
        - etc
    - Pros (centralized)
        - Constant quality of service
        -
        - Single point of access
        - More resources for system maintenance, security, evolution
        - Accountable entity (privacy issue, lawsuit)
        - Effective information exchange due to recommender systems
    - Cons (centralized)
        - single point of failure (privacy, security, governance)
        - prone to censorship and systematic infiltration by governments
        - often closed / proprietary system code
        - influence concentrated in one organization
    - Pros (Decentralized)
        - Multiple points of access
            - e.g. different pods
        - More robust
            - e.g. if a pod is hacked then the others a maybe still safe
        - often open source
        - Easy to host new instances
            - if developers considered it (was not the case with mastodon)
        - Individual nodes cost less
    - Cons (Decentralized)
        - quality of service depends on individual node
            - software updates
            - hardware specs
            - firewall systems
            - who moderates the content (stricter looser)

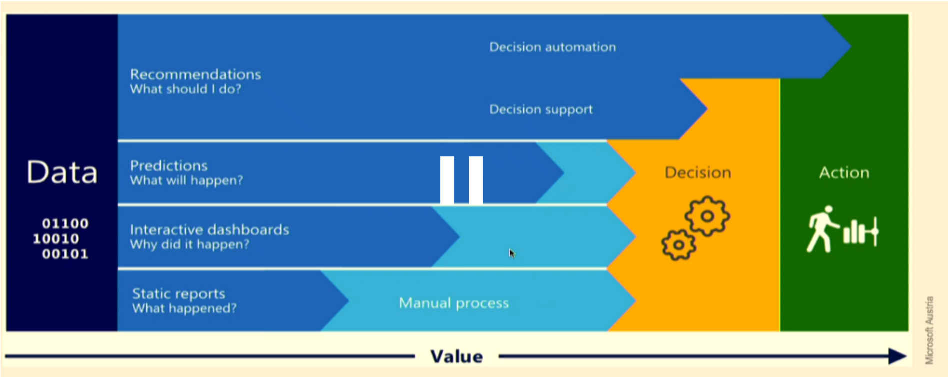
            - each node is responsible for its maintenance and data security
            - less effective info exchange because of fragmentation of user base
            - little to no recommender systems
            - user contributions stored on an individual node
        - for decentralized systems the CIS can also be a publish/subscribe implementation

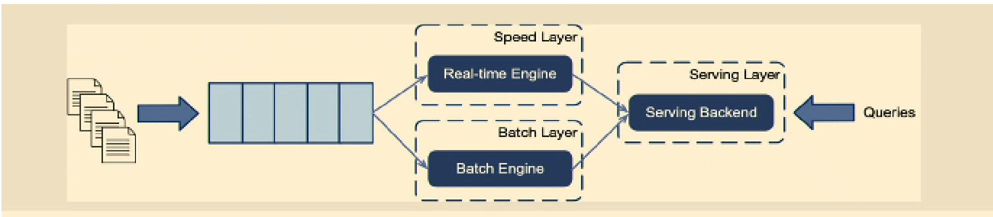
- **Technology Stack**
    - Front End
        - Web client
            - Angular JS
            - Ruby on Rails (RESTful, MVC pattern, bundler - maintains consistent environment)
        - Desktop client
            - Electron (Chromium browser engine + node.js)
        - App client
        - Wearable client
            - e.g. activity tracker, smartwatch
    - Back End
        - Spring Framework / Spring boot
    - Hyperscaler
        - cloud computing system
        - Can handle very small and very large volumes of data / computing load / traffic
            - Example: AWS, Azure
        - Horizontal Scaling - scaling out
            - Virtual machines, more storage, memory, networking
        - Vertical Scaling - scaling up
            - Upgrade capacity with better hardware
        - hyperscalers are expensive
        - 

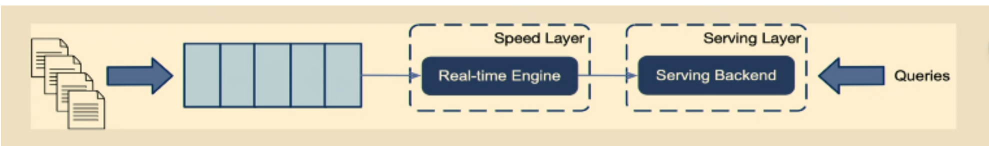
        - Constituents

        - Computing
            - Virtual Servers
            - Machine learning
            - Analytics
            - Serverless
        - Storage
            - managed databases - NoSQL
            - hyperscalable databases - AWS aurora
            - object storage - aws s3
            - backups
        - Networking
            - Content Deliver Network (CDN), Load Balancing
            - Virutal Private Clouds
            - Gateways and Service Orchestrators (REST, Microservices, API-Gateways)
        - Security & Compliance - often overlooked
            - certifications (C5 - europe/germany)
            - firewalls, DDoS/Traffic Protection, Detection Services, Access / Identity Management
            - Governance, Auditing and Reporting Services
            - compliance is more important -> we certify the hyperscaler, if your application is completely on the hyperscaler, than the application is also compliant
                - think finance, health or government organizations
                - e.g. hetzner in germany, aws,...
    - Architecture Concerns
        - New solution design and development
            - design and implementation strategy -> reliability requirements
            - design for business continuity
            - design for performance objectives
                - inbound / outbound processing
            - deployment strategy
                - how to handle source code, deployment,...
        - Resilience
            - multi-tiered architecture
            - high availability and/or fault-tolerance
            - Decoupled granular Service Organization
            - Resilient Storage (Decade+)
                - different variants
                - use pricing calculators - who much and when
        - High-Performance
            - hot or cold storage
            - elastic and scalable computing, storage and network workload handling
        - Security and Compliance
            - secured application tiers and networks

- mechanisms for resource access and data security
    -
- Cost-Handling and Cost-Optimization
    - identification, selection, implementation and review of cost-effective compute, storage and networking solutions
    - design implement review controls

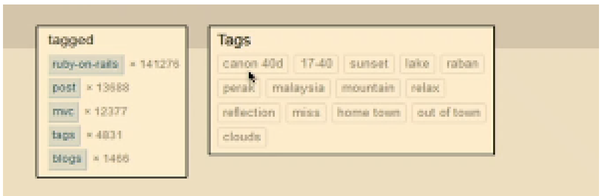
- **From data to intelligence to decisions to action**
    - 

- **Big Data Processing: Lambda Architecture**
    - different kind of data
        - old data - batch layer
        - new data = real time - speed layer
        - Combination of old and new - service layer
        - 

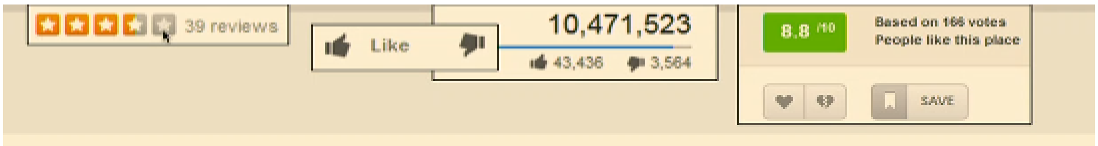
    - Pros
        - batch layer manages historical data - at least something can be served
        - balance between of reliability and speed
        - good scalability
    - Cons
        - Coding overhead due to involvement of comprehensive processing
        - Reprocessing every path cycle not suited for certain scenarios
        - Data may be difficult to migrate/reorganize
- Kappa Architecture
    - 

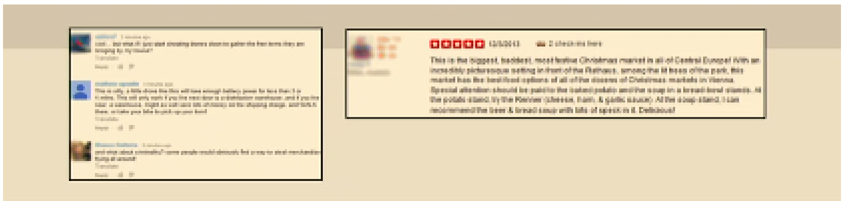
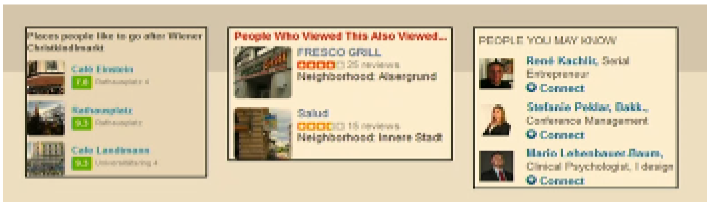
    - pros

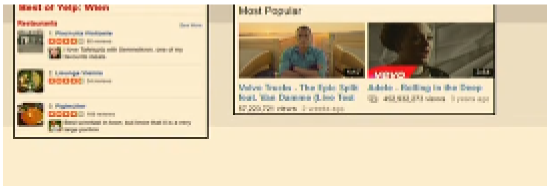
            - suited for system that depend on hot, online data and no cold storage (batch layer)
            - suited for horizontal scalable systems
            - pre-processing is only if code changes
            - fixed memory deployment
        - cons
            - lack of batch layer increases risk of errors during data processing or database updates / reconciliation
            - more expensive
        - for data where there is a little error - because later data is used; newest data is the most important
- **Separation of Data Storage and Processing**
    - Collect all data
    - store all raw data
        - storage technology
            - relational database management system
            - in-memory database system / caching
            - graph databases
                - graph structures for semantic queries
                - can be used together with relational database
            - BLOB /object storage
                - storing massive amounts of unstructured data like images, video, docs, audio
                - e.g. AWS Amazon S3
                - e.g. use for client-centric web-applications
    - process and analyse data
        - use analytic engines to perform analysis on collected and stored data
            - batch queries, interactive queries, real-time analysis, machine learning
            - Apache Kafka, Azure ML
    - apply and provide results:

- **Trade-oFF Analysis Hyperscaler Example IaaS and Serverless**
    - IaaS
    - TODO
- CI Design Patterns
    - Tagging
        - 

        - Problem: Information is dispersed and not grouped
        - Solution: It enables users to categorizes content on their own
    - Rating

- 

- Comments
  - 
- Hashtags
- Recommendations
  - 

- Generated Lists
  - 

- Follow Subscribe
  - 

- Activity Indicator (e.g. Github)
  - 

- User-generated Collections