C4

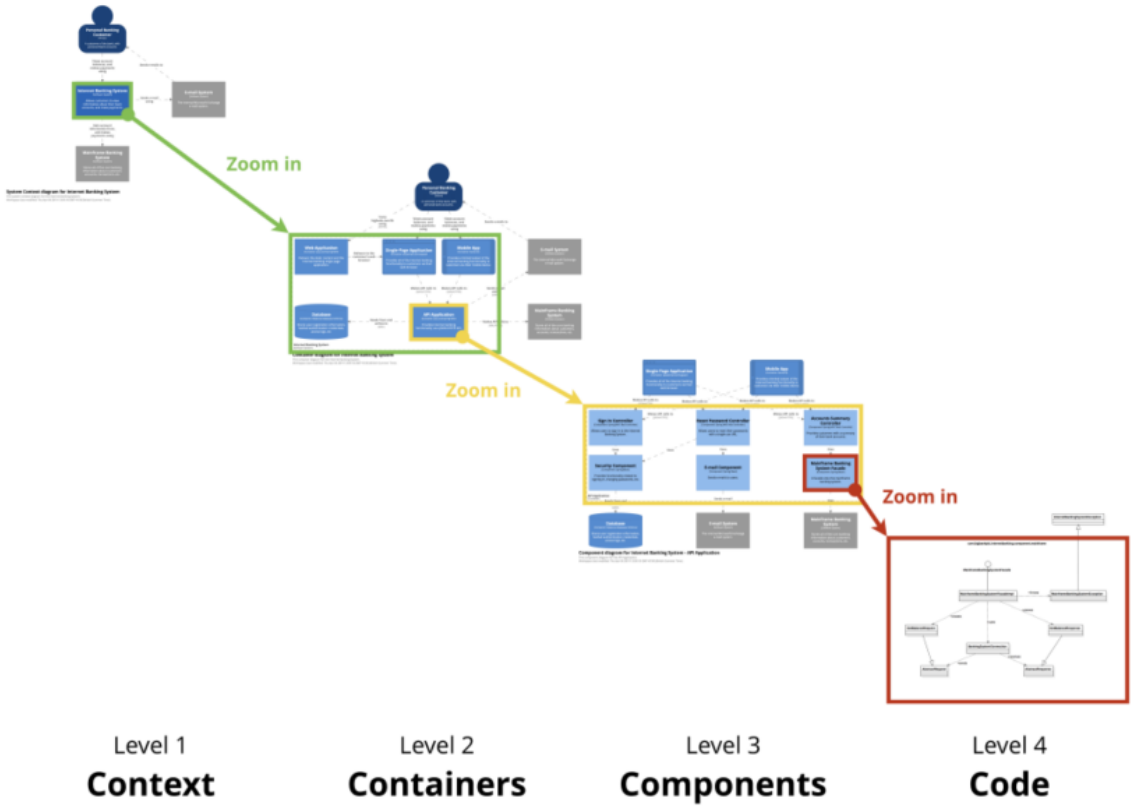


Figure 1: image-20230427100758442

Context

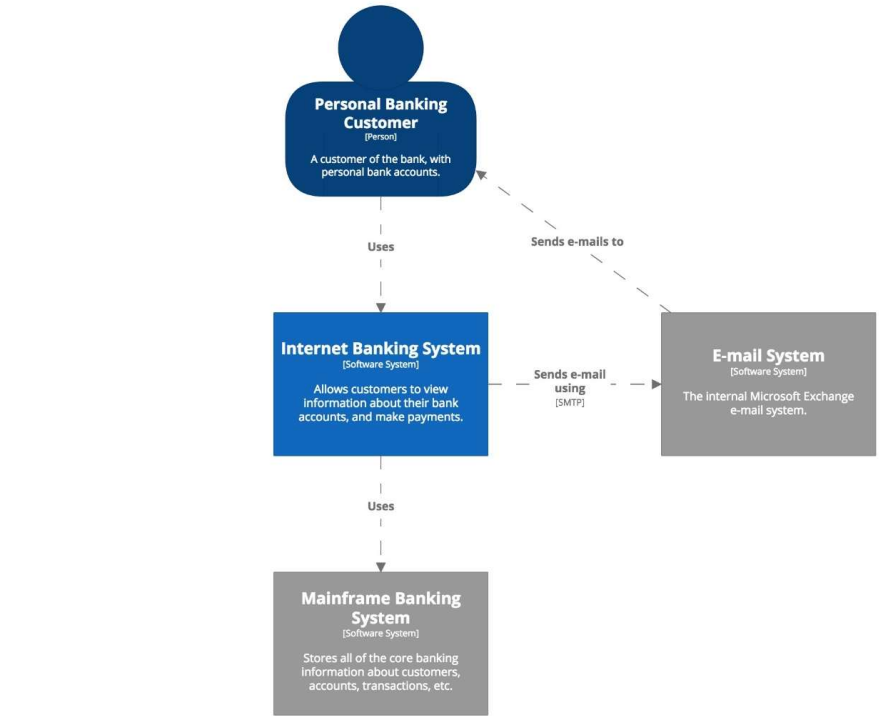
Ein Systemkontextdiagramm hilft, folgende Fragen zu beantworten:

1. Was ist das Software-System, das gebaut wird?
2. Von wem wird es verwendet?
3. Wie passt es in die bestehende Umgebung?

Containers

Ein Containerdiagramm hilft, folgende Fragen zu beantworten:

1. Wie ist die Gesamtform des Softwaresystems?
2. Was sind die High-Level-Technologieentscheidungen?
3. Wie sind die erantwortlichkeiten im System verteilt?
4. Wie kommunizieren die Container miteinander?
5. Wo muss der Entwickler Code schreiben, um Funktionen zu implementieren?



System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Last modified: Wednesday 02 May 2018 13:46 UTC

Figure 2: image-20230621125601571

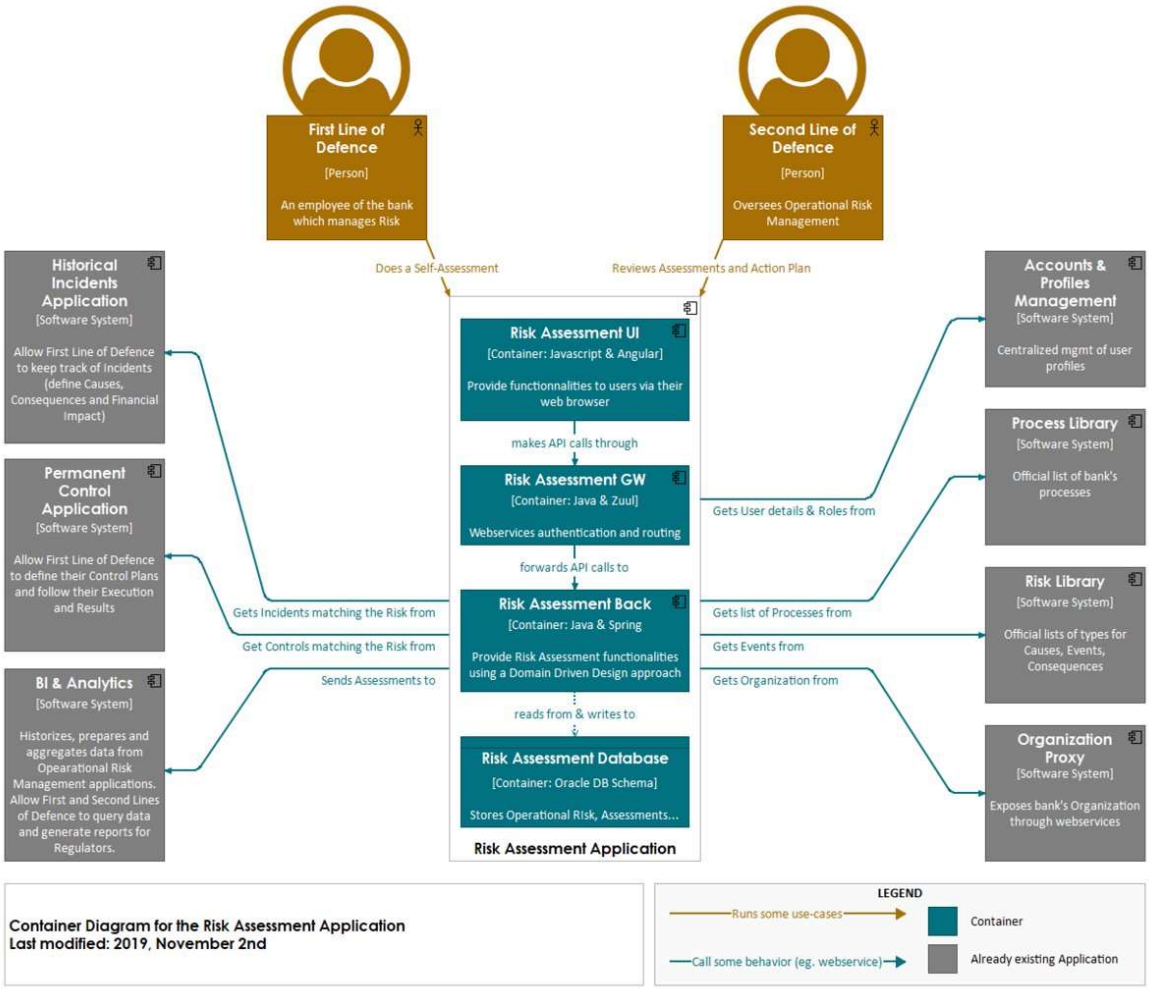


Figure 3: image-20230621125614920

Component

Ein Komponentendiagramm hilft, die folgenden Fragen zu beantworten. 1. Aus welchen Komponenten setzt sich jeder Container zusammen? 2. Haben alle Komponenten ein Zuhause (d. h. sie befinden sich in einem Container)? 3. Ist es klar, wie die Software auf hoher Ebene funktioniert?

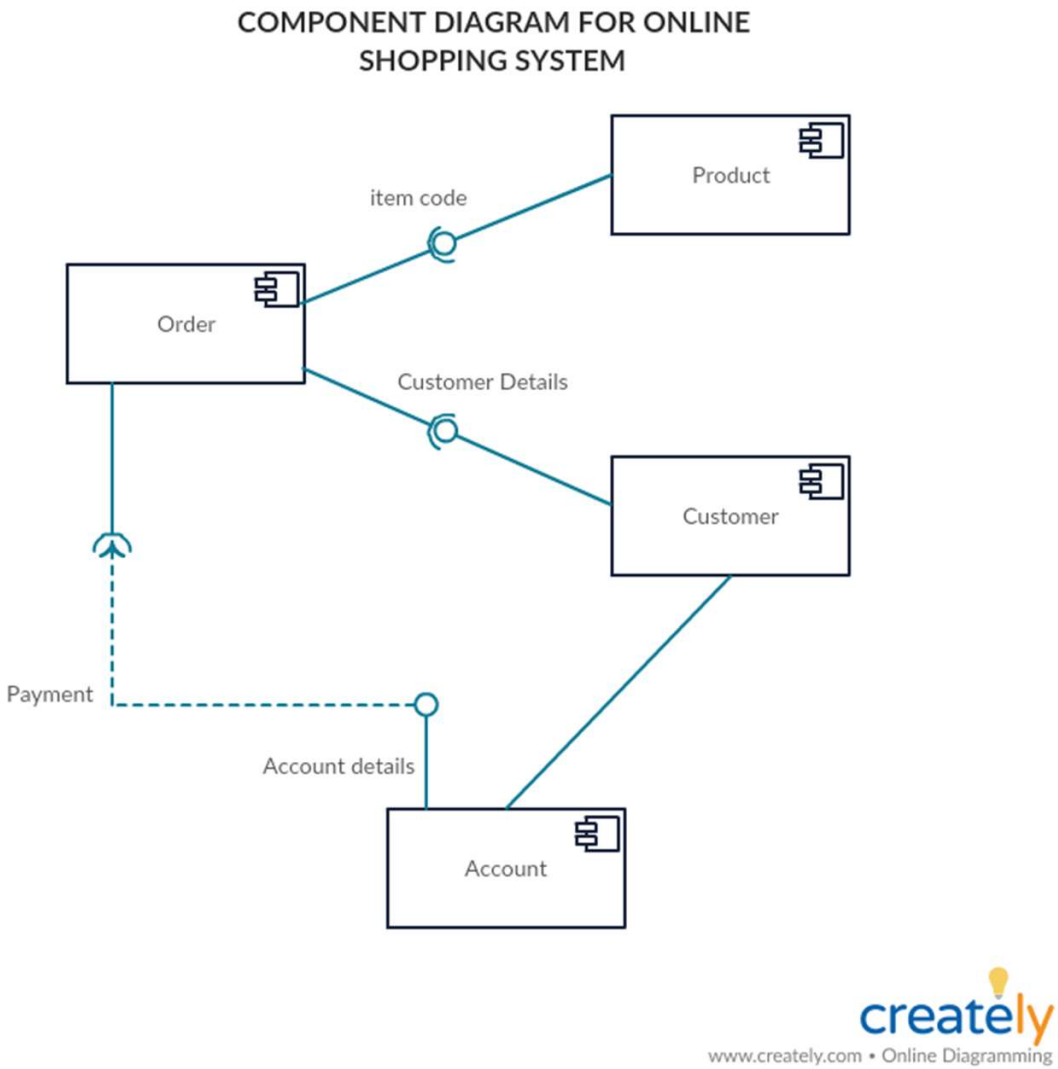


Figure 4: image-20230621125624928

CI

- On-Demand run a script or press a button
- Scheduled at certain times → nightly builds
- Triggered on certain events → commit/push to VCS

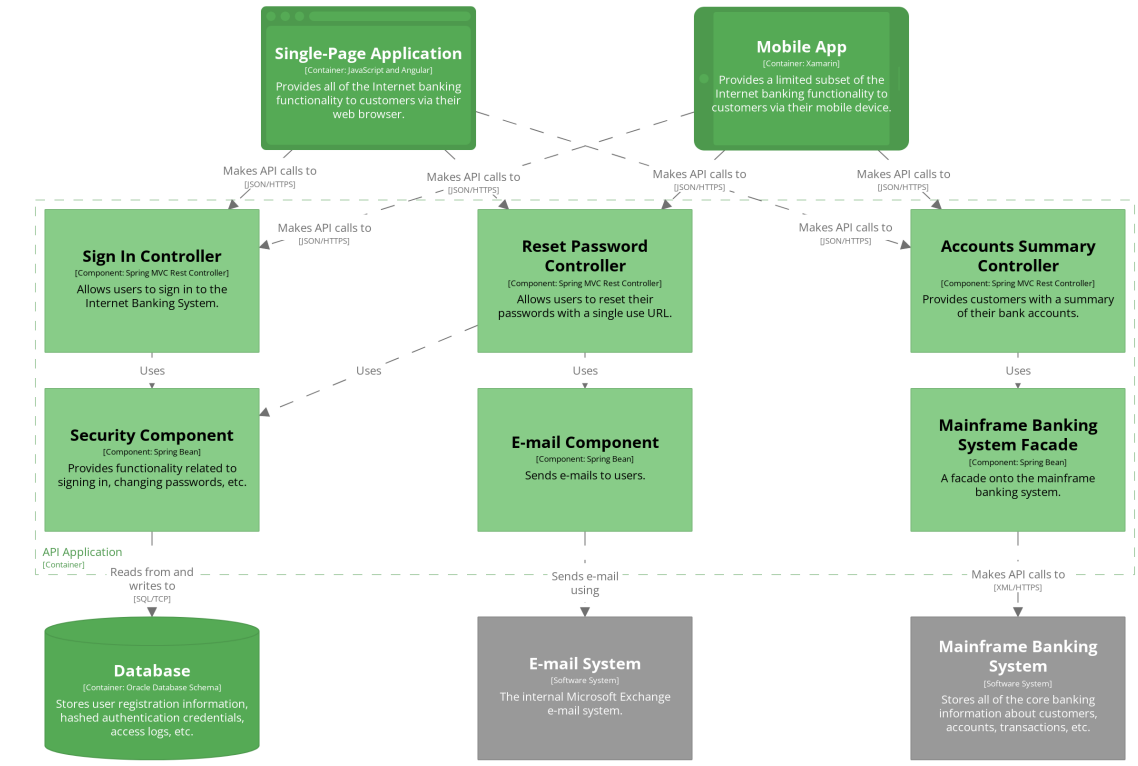


Figure 5: image-20230621125653174

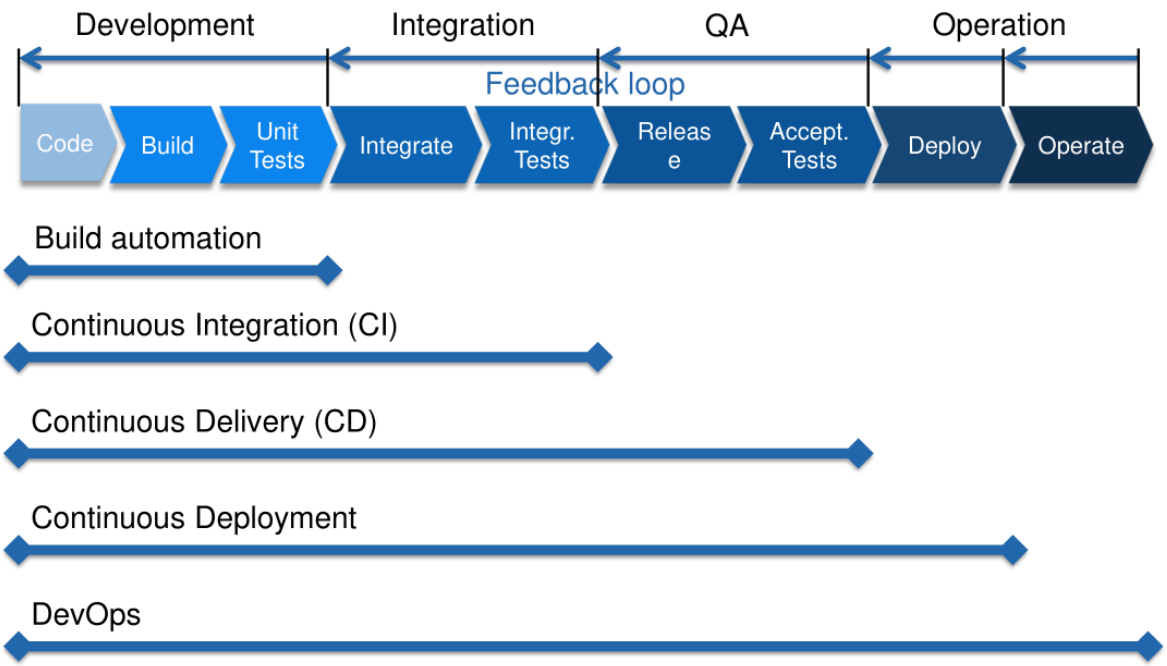


Figure 6: image-20230612153748431

Types of Automation

- Build Automation Thinks like building, packaging or creating documenation
- Test Automation Automated execution of unit-, integration- and acceptance-test
- Deployment Automation Automatically deploy to production or testing
- Operation Automation Automatically provision infrastructure, monitoring, ...

Automation has a lot of benefits like:

- developers can't reliable built software locally
- unit testing consistent
- transparent deployment
- documentation from Jenkinsfile or similar
- manual work is error-prone
- ...

Best Practices

- Maintain a single source repository
- Automate the build
- Make your build self-testing
- Every commit should build on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening
- Automate deployment

Cynefin

Cynefin is not a model (represents reality), but a sense making tool (a way to look at reality).

- **Simple/Clear**: There can exist a step-by-step instructions
- **Complicated:** In a complicated system, one needs to consult an expert, but there is still an ordered system
- **Complex**: A complex system can only be understand by probing it and interacting with it
- **Chaotic**: A transistion state in which a system has no rules. The brain strugles dealing with this. In an chaotic system, you do something and see what happens, then you do it again and something else happens.
- **Disorder**: A transition state where you don't know in which state you actually are and just use the model which is most familiar.
- **Collapse from complacency**: When a system is overconstraint and looks nice and simple, but then collapses when the reality isn't that simple.

Between complicated, complex and chaotic isn't a strict line.

Codefin

Exaptation

Exaptation, also **radical repurposing**, is the taking of an idea, concept, tool, method, framework, etc., intended to address one thing, and using it to address a different thing, often in another domain.

An example is the re-purposing of snorkelling mask as oxygen masks during the Covid-19 crises. Similarly, a Ukrainian web app designed for use for transport, parking and paying utility bills has been

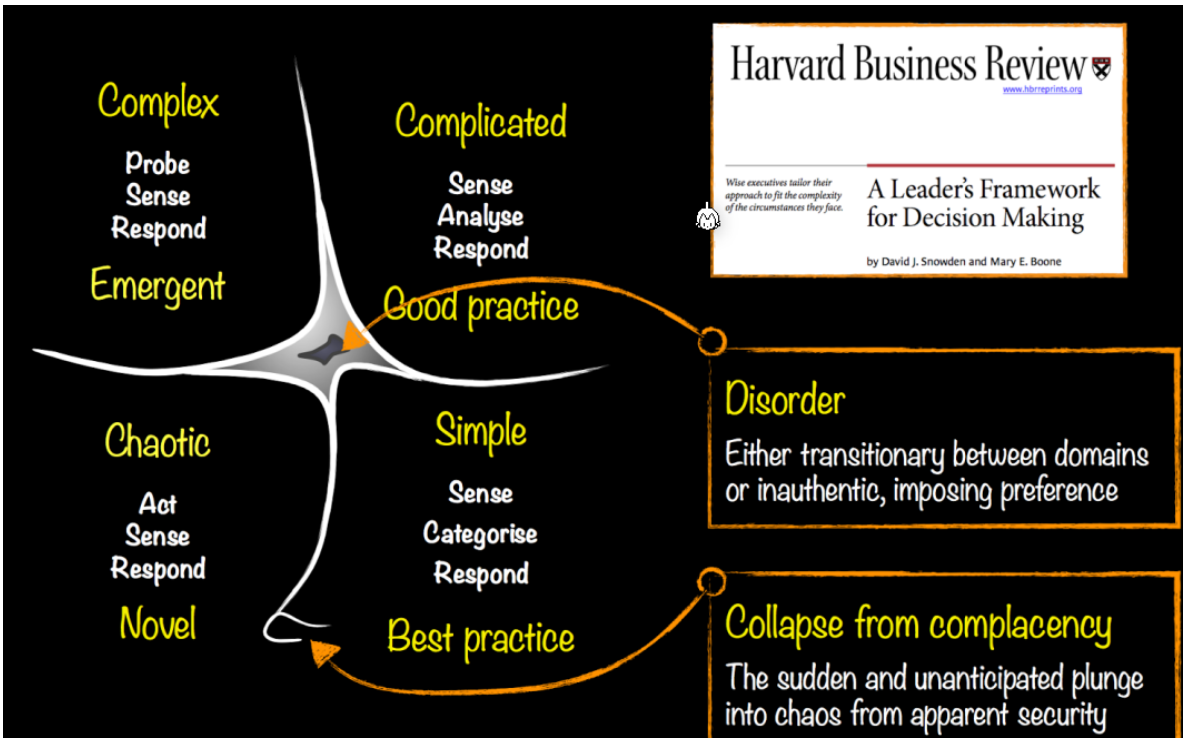


Figure 7: image-20230504090919828

re-purposed to warn of air raids, and directs people to bomb shelters.

Constraints

Constraints limit the number of options one have. In the cynefin framework, different stages have different kinds of constraints:

- **Simple Domain**: Rigid constraints
- **Complicated Domain**: Governing constraints
- **Complex Domain:** Enabling constraints
- **Chaotic Domain:** Absence of constraints

Governing/Enabling Constraints

Laws, rules and codes create a governing constraint and give a sense of stability. However, they are sensitive to change.

Internal/External Constraints

(Unsure what this means. . .)

Connections, like hashtags in knowledge management and links in networks, provide a flexible and adaptive structure but at the cost of visibility and control. Containers, like categories, spreadsheets cells, and departments, provide clear, reassuring boundary conditions. Changing connections between people and organisational units is less costly than trying to restructure or reorganize departments. As new connections start to provide new ways of dealing with issues, then the constraints can be tightened and eventually formalized into new units and departments.

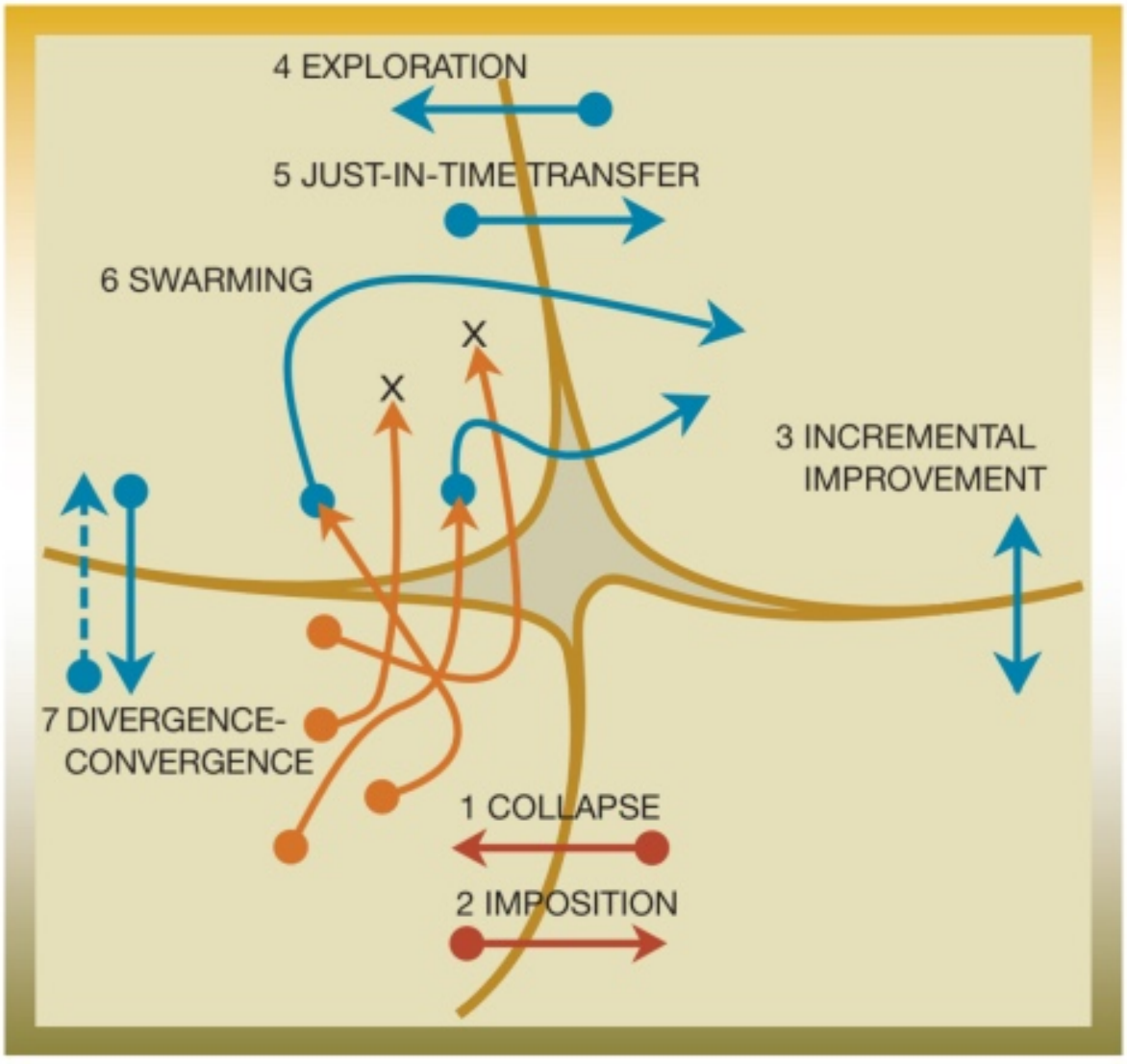


Figure 8: image-20230504092946875

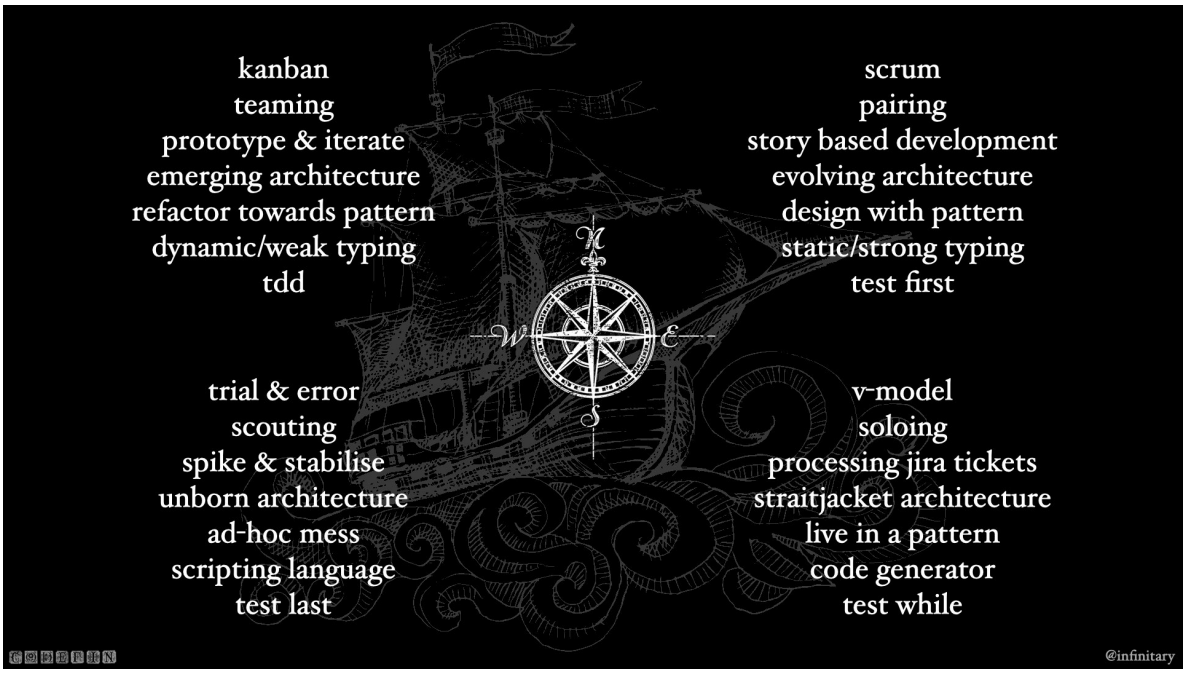


Figure 9: image-20230504092758899

Connecting/Containing Constraints

Connections between people or between departments are flexible can be easily changed but invisible and hard to control effectively. On the other side, categories, containers and departments provide a clear structure, but changing them is costly.

One way is to let connections form and then later tighten the constraints and formalise the connections into categories, departments and rules.

Rigid, Flexible and Permeable Constraints

Rigid constraints cannot be changed. If rigid structures are pressured enough, they fail catastrophically.

Flexible constraints adjust to pressures constantly.

Both can be enhanced with permeability, which is a special condition that allows for exceptions. This, however, comes with the danger of too many people applying for the exception.

Dark Constraints

Dark constraints are not officially defined and cannot be seen, only their effects can be observed. [Apparently,] narratives are powerful antidotes against dark constraints.

Dark constraints can somewhat be meassured by looking how much of the past can be explained with the known constraints. The more unexplainable things, the more likely are unexpected and potentially catastrophic surprises.

Lean

The Toyota Production System (TPS) has two pillars: **Just-in-Time** and **Jidoka**.

TPS/Lean focuses on reducing the waste (unnecessary code, meetings, documentation, bugs, ...) in a system and produce a higher value for the final customer.

The followings are the principles of lean:

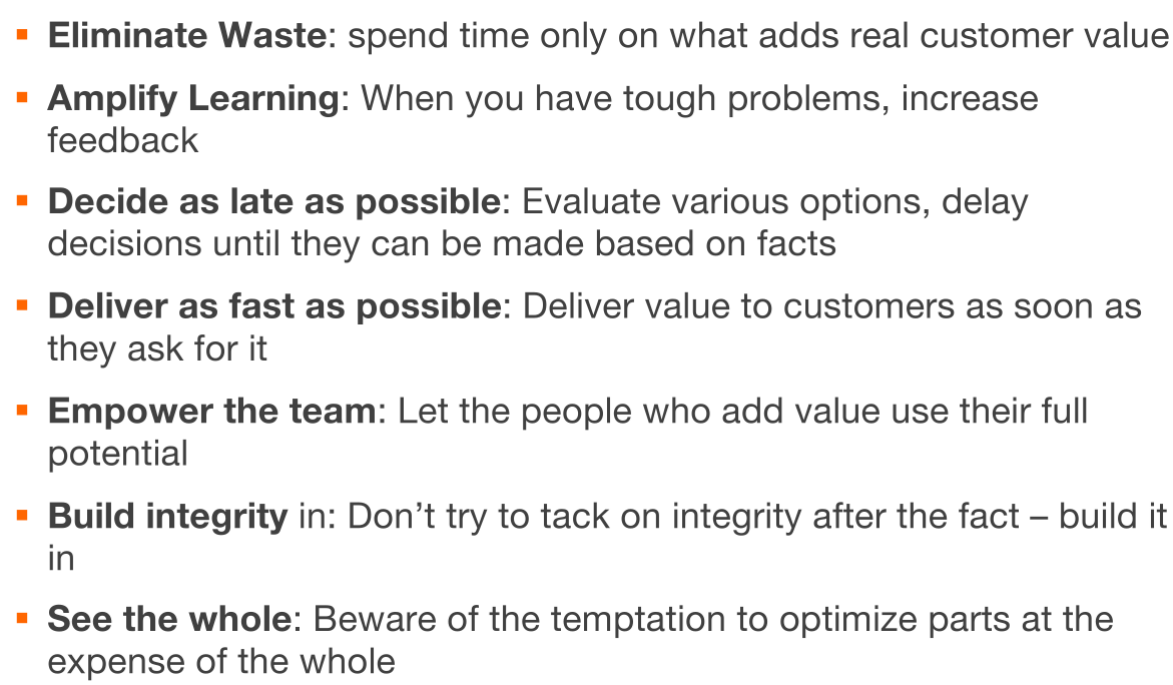


Figure 10: image-20230525084856677

There are three types of waste:

- Wasted code Code which is either partially completed, outdated, unnecessary or defected
- Project management Processes which aren't necessary, hand offs (there is a loss of knowledge), or extra features
- Waste in work force
  - Task switching slows people down
  - When people wait for instructions or information

SAFe

Traditional management optimises for efficiency and stability, while agile optimises for innovation speed. SAFe tries to inject innovation speed into traditional management.

SAFe keeps the traditional hierarchy, but adds agileness into individual blocks in the hierarchy.

The following is an overview of SAFe:

SAFe Core Values

1. Alignment: Leaders communicate the mission by establishing and expressing the portfolio strategy and solution vision, determining business value during planning, and guiding the adjustment of scope to ensure that demand matches capacity.

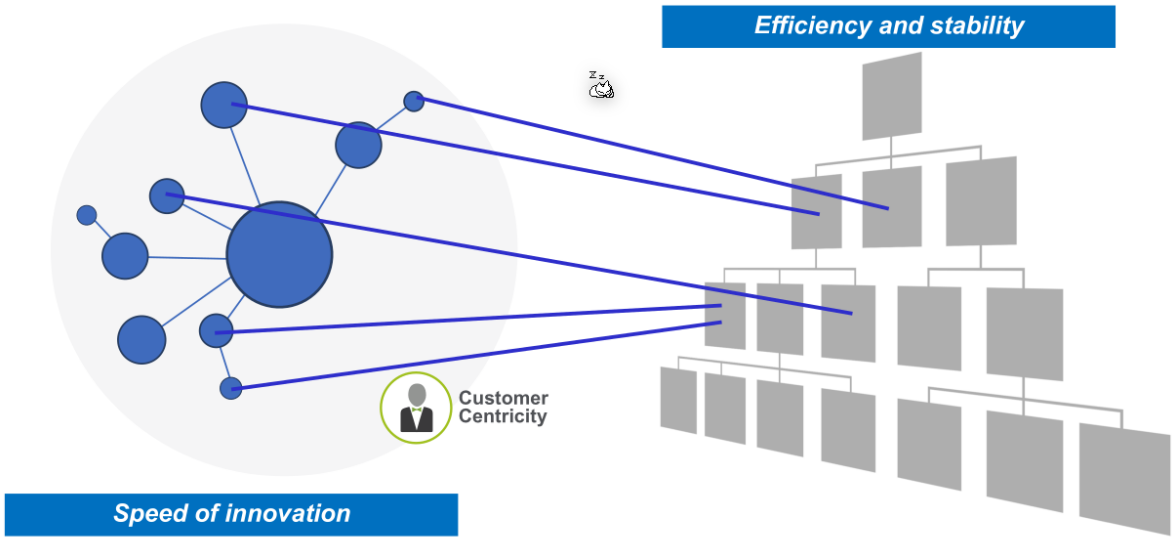


Figure 11: image-20230406090241961

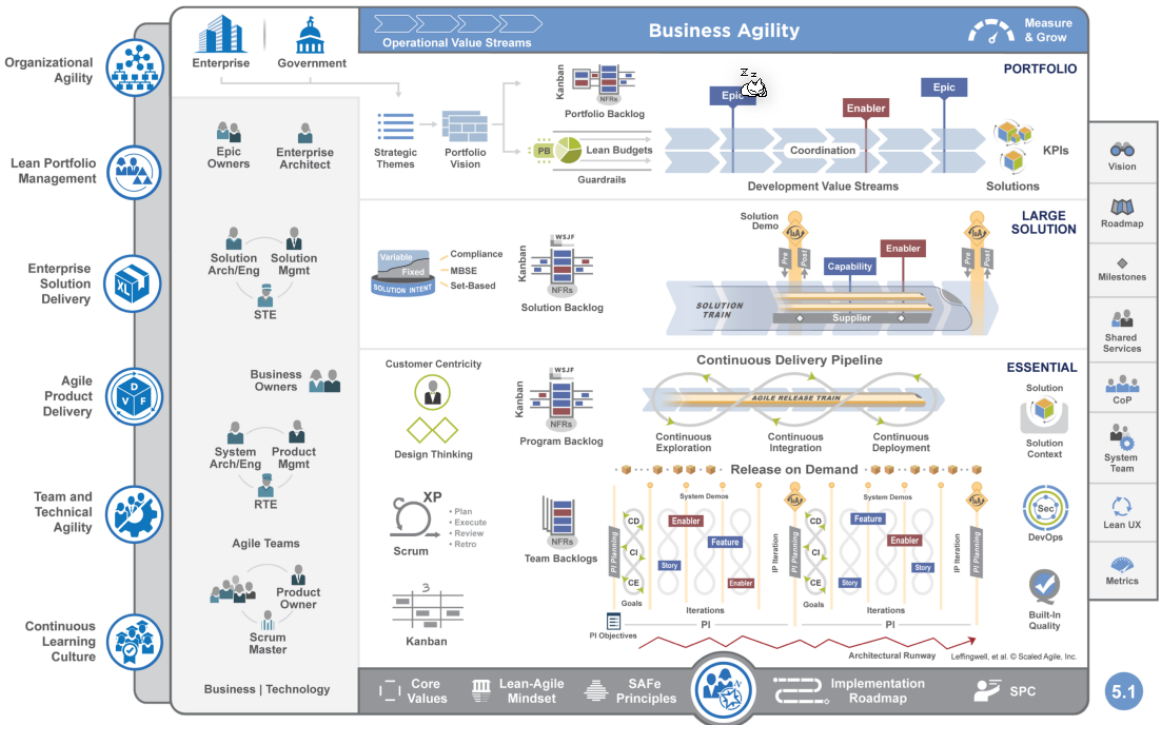


Figure 12: image-20230406091242467

2. Built-in quality: Leaders change the system and demonstrate commitment by creating an environment where built-in quality becomes the standard.
3. Transparency: Leaders foster the visualization of all relevant work and create an environment where "... the facts are always friendly, every bit of evidence one can acquire, in any area, leads one that much closer to what is true."1
4. Program execution: Leaders participate as Business Owners in PI planning and execution, celebrating high-quality PIs while aggressively removing impediments and demotivators.

SAFe Principles

1. Take an economic view
2. Apply systems thinking
3. Assume variability; preserve options
4. Build incrementally with fast, integrated learning cycles
5. Base milestones on objective evaluation of working systems
6. Visualize and limit WIP (Work-in-Process), reduce batch sizes, and manage queue lengths
7. Apply cadence, synchronise with cross-domain planning
8. Unlock the intrinsic motivation of knowledge workers
9. Decentralise decision-making
10. Organise around value

SAFe Levels

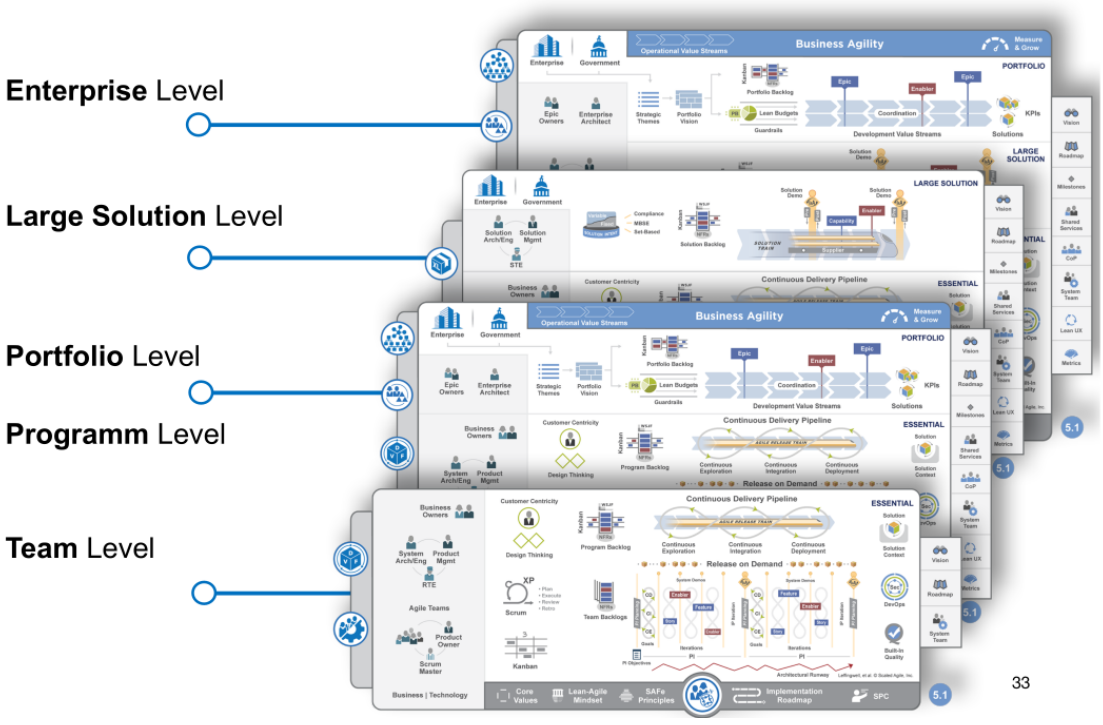


Figure 13: image-20230406091700167

SAFe Roles

(Product Manager, System Architect, Release Train Engineer, Business Owner, Scrum Roles)

- The **Product Manager** is responsible for the higher-level Product Backlog, which is then called Program Backlog
- The **System Architect** is responsible for the system design
- The **Release Train Engineer** takes care of the collaboration and dependencies between teams as a kind of cross-team Scrum Master
- The **Business Owner** keeps an eye on ROI and value maximization as well as balancing costs and benefits in product development at the system level; thus relieving the Product Manager somewhat in this regard
- **AND:** the Scrum roles (**Product Owner, Scrum Master, Team**)

Figure 14: image-20230406092207231

Agile Release Train (ART)

An agile release train is a long-lived team of agile teams that incrementally develops, delivers, and often operates one or more solutions in a value stream.

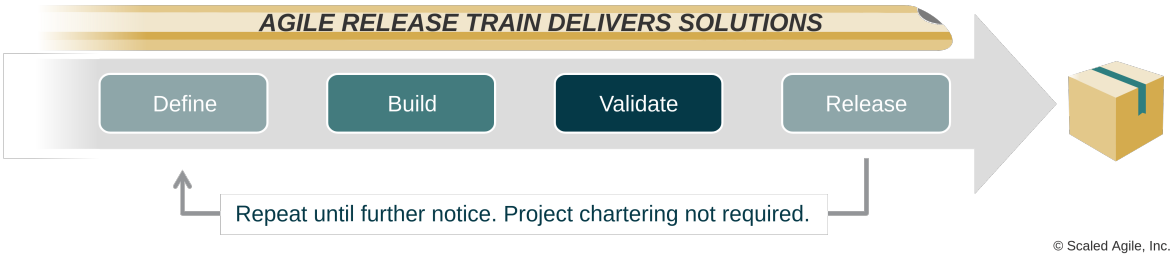


Figure 15: image-20230621123757323

Software Architecture

> Big design up front is dumb. Doing no design up front is even dumber.
>
> Or do just enough...

Software architecture can be defined as:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relations among them by Simon Brown

Application Architecture

The main concern is the organisation of code. It matters how the code is split into modules, components, layers, ...

Usual questions in the application architectures are:

- Cross-cutting concerns: Issues which span multiple layers/components, like exception handling, logging, authorisation & authentication, ...
- Security
- Performance, scalability
- Auditing, regulatory requirements
- Real-word constraints
- Interoperability and integration into other software
- Operational, support and maintenance requirement
- Structural consistency and integrity. Things, like classes, modules, etc, should be where people expect them

Non-Functional Behaviour

Non-functional behaviour are the properties of a software. Some of these can be measured while run-time, and some can't.

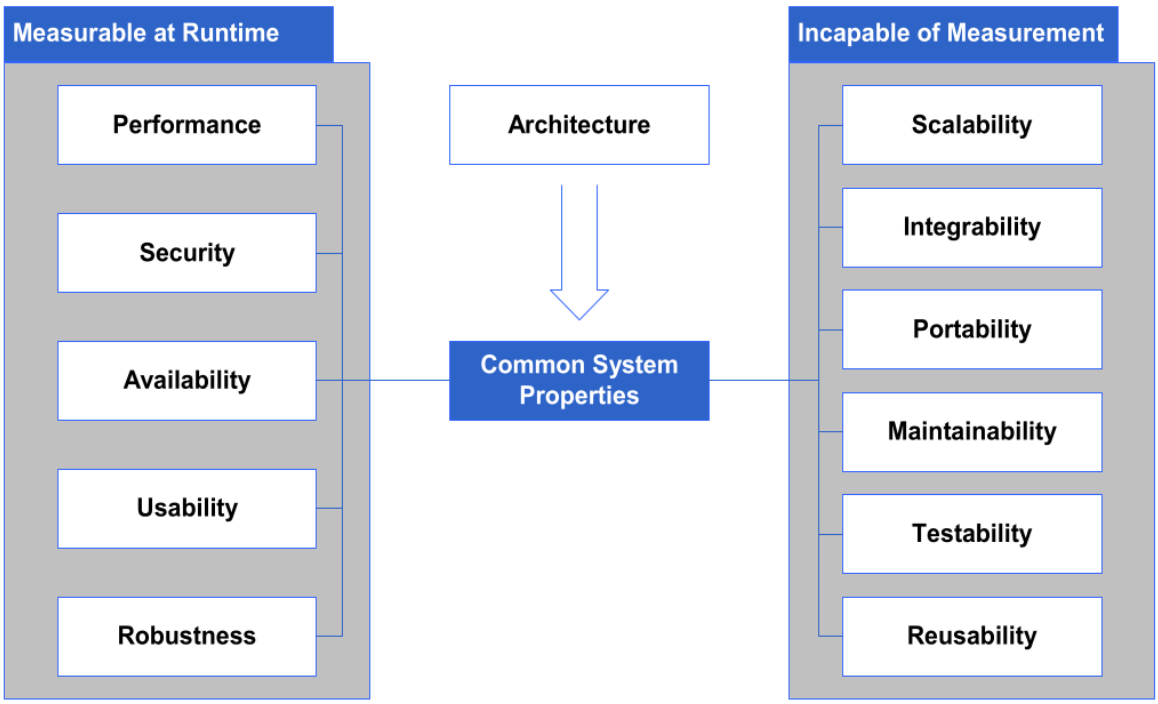


Figure 16: image-20230420085521967

- Performance (response time / throughput): A system must guarantee the required response times. Not least because system response times have a significant influence on work productivity.
- Security: A system must be secure against unauthorized access and wanton destruction.
- Availability: A system must be available and meet defined minimum requirements.

- Usability: A system must be usable for its intended purpose.
- Robustness: A system must run stably and must not partially or completely stop its service under load.
- Scalability: A system must be able to be scaled (out or up).
- Integrability: A system must fit seamlessly into an existing environment.
- Portability: A system must support different platforms.
- Maintainability: A system must have defined maintenance interfaces and a clear and concise structure.
- Testability: A system must be testable as a whole and in its individual components. The testing of the system must be supported by the system itself through tools (automated tests, logs, traces).
- Reusability: System components must be reusable for other systems.

The divide between measurable and immeasurable isn't as binary is it is made up to be in the slides.

Principle of Software Design

Software should be modular, portable, changable.

- Modularity: Components of a design should consist of easily exchangeable, understandable and self-contained parts (e.g. ACL, DB Access, Validator, Rule Engine).
- Portability: Portability is given when software or parts of it are designed to run in other environments,
- Changeability: The more malleable a system is, the easier it is to make changes (e.g. by separating domain specifics from cross-cutting concerns).
- Conceptual Integrity: Those parts of a system that contain similar functions should also be designed similarly (industry standards, reference architecture). Or, if you don't build a rocket, the architecture doesn't need to be rocket science.
- Intellectual Control: A software design should be understood in detail by those responsible in terms of form, content and complexity (interface, scope). If the people working on the software don't understand its architecture then the architecture won't be implemented
- Buildability: A software design must specify a target system in such a way that it can be realized by a given team in a given time (know-how, technology).

Coupling and Cohesion

There are different kind of coupling and cohesion. The basic rules, that higher cohesion will generally result in lower coupling.

- Data Coupling: data is exchanged between modules of a system
- Stamp Coupling: Data structures are exchanged between modules of a system
- Control Coupling: The exchange of data between modules controls the control flow
- Common Coupling: Different modules access the same data (Shared Data)
- Content Coupling: One module modifies the internal data of another module
- Coincidental Cohesion: The grouping of the functionality of a module is done by chance.
- Logical Cohesion: The functionality is grouped in a module, but they do not refer to each other.
- Temporal Cohesion: The time of the use determines the grouping of the functions.

- Procedural Cohesion: The call sequence of the functions determines the grouping.
- Communications Cohesion: The grouping of the functionality is determined by the common I/O.
- Sequential Cohesion: The sequence of data processing determines the grouping. Functions whose output simultaneously becomes the input for other functions are grouped together in a module.
- Functional Cohesion: The aim of this grouping is that a module can keep logic and data locally (information hiding).

To actually implement high cohesion, low coupling, the following can be done in code:

- **Independence of Design**: Each module can be designed independently of other modules and later changes take place only and exclusively in one module. The prerequisite is a constant definition of the interfaces and a "freezing" of the module specification.
- **Small Interfaces**: The number of messages exchanged between modules via interfaces is small.
- **Low Interface Traffic**: The frequency of information exchange between different modules is low.
- **Unity**: Similar problems and requirements are implemented similarly and classified and grouped accordingly.
- **Encapsulation**: Dependent modules are combined into larger units.

Example

A system is to be created that will allow easy checking of banknotes based on their security features and simplify the reporting of potential counterfeits.

- The system is to be made available to banks, bureaux de change and other cash acceptance offices.
- The system must be web-enabled and capable of simultaneously exchanging data with the central bank's laboratory system.
- The system must be multilingual and multi-client capable.

The following layers were designed:

These three layers contain the following seven logical layers:

These layers are then deployed on the following architecture:

Silver Bullet

This comes from the paper "No Silver Bullet—Essence and Accident in Software Engineering".

Building software is so difficult because of its complexity, conformity, changeability and invisibility.

Complexity can be split in inherit complexity and accidental complexity. Inherit complexity is complexity which comes from the fact that the project itself is complexity. Accidental complexity comes from spaghetti code.

Because a software needs to conform to interfaces of its environment, the interface is as complex as its environment. Put differently, the more software is interconnected, the more complicate it gets.

Because software is easily changed, many changes are made.

Software is invisible in itself. There is no geometric representation of software. To still be able to visualise it, one creates views into it, with diagrams and text.

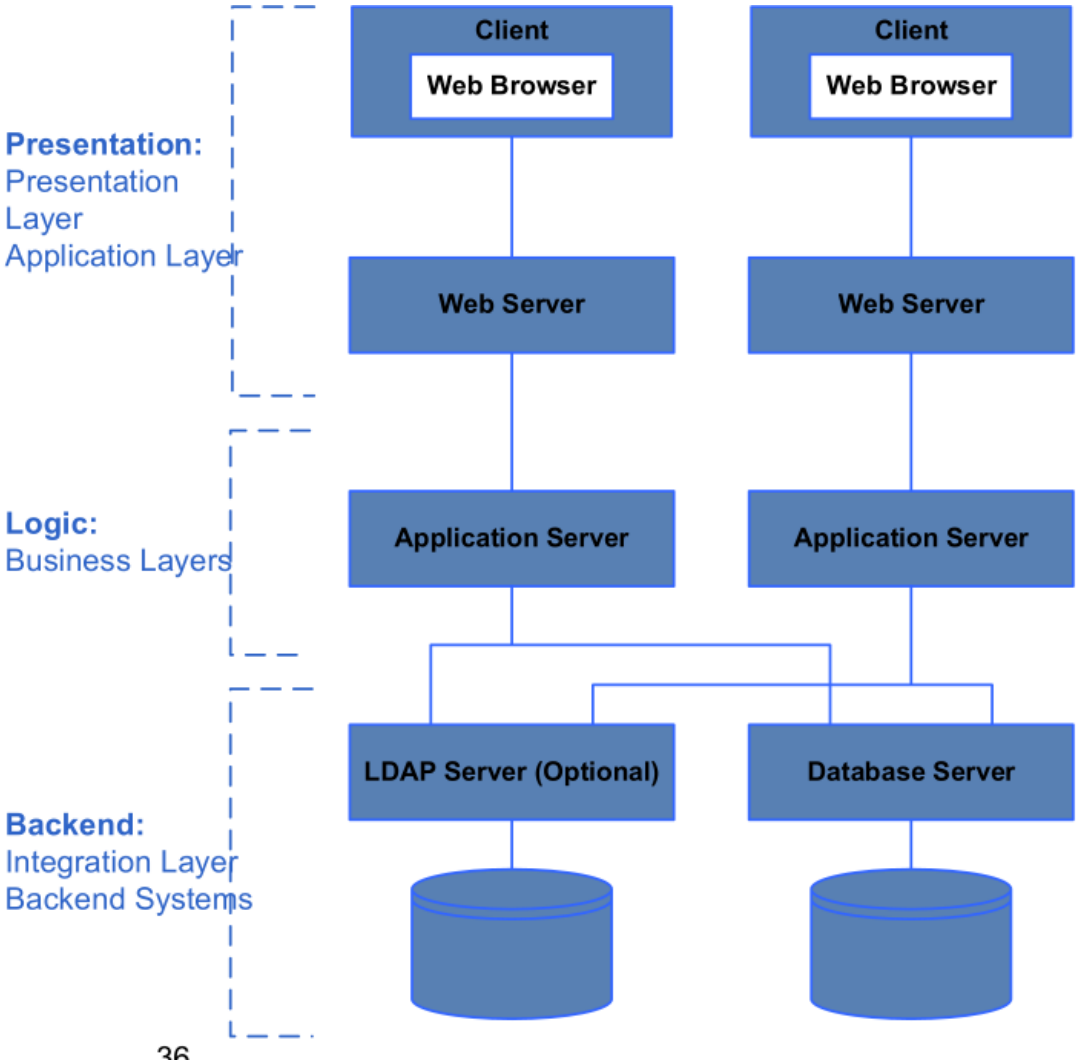


Figure 17: image-20230420091533612

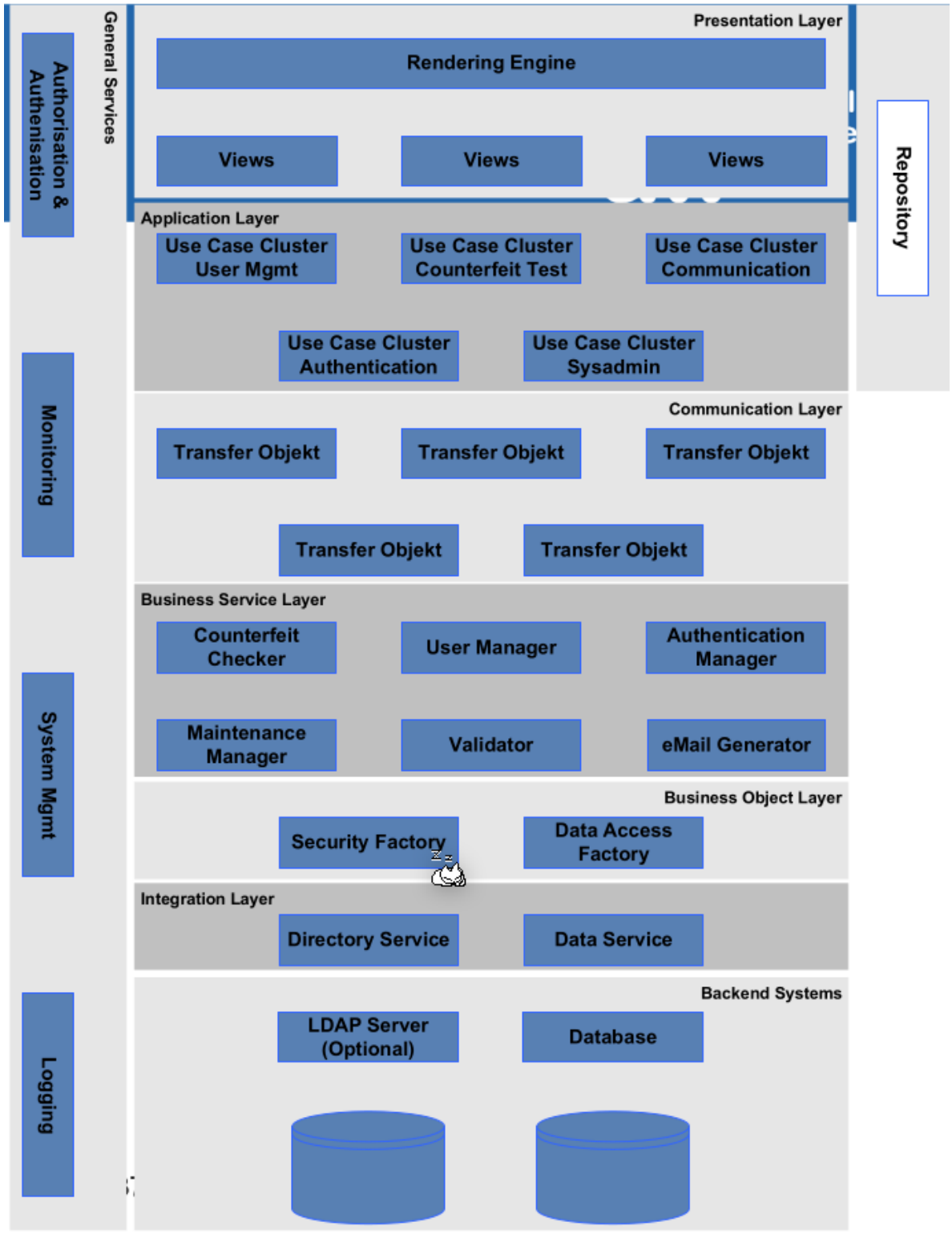


Figure 18: image-20230420093034031

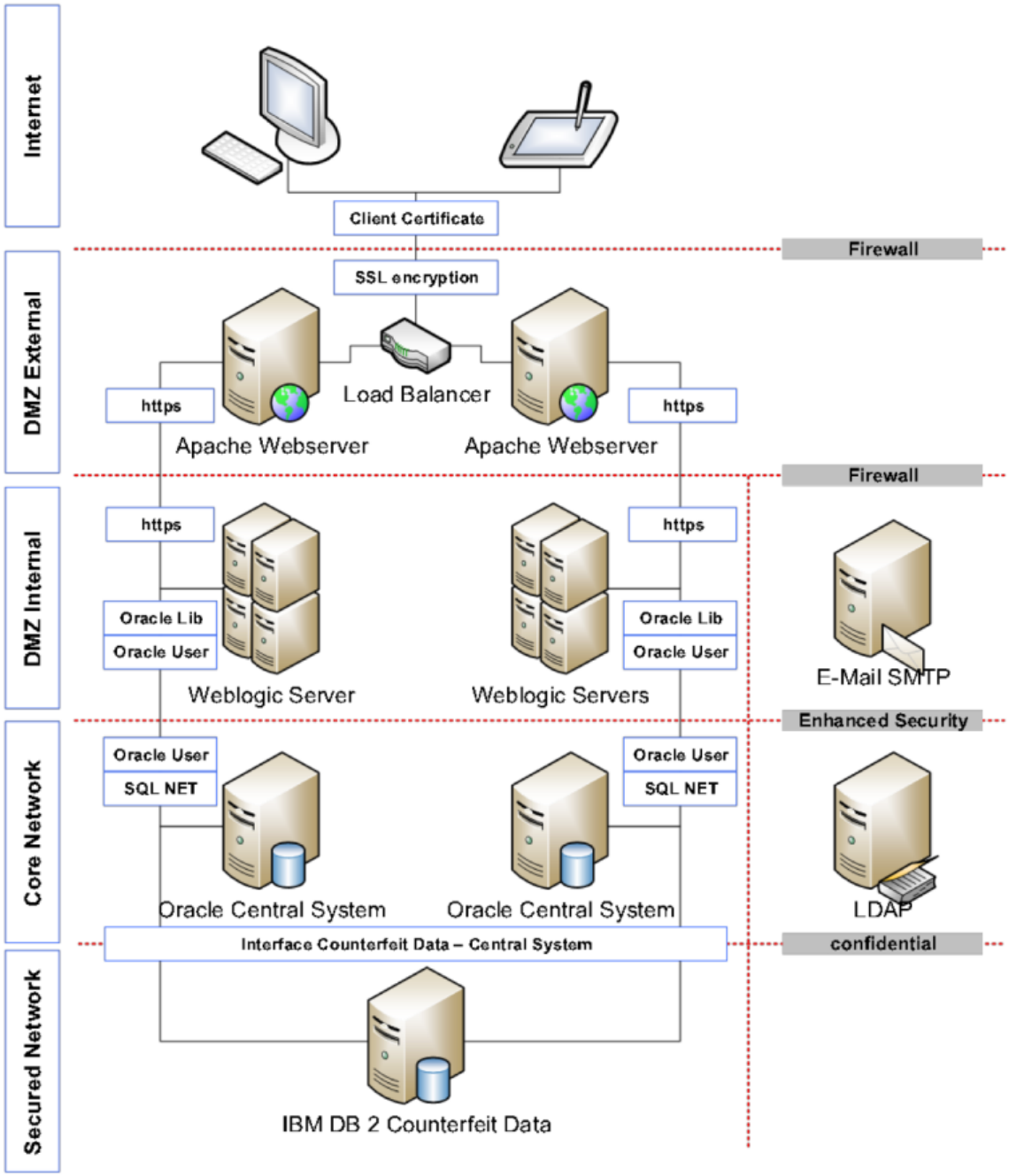


Figure 19: image-20230420093059847

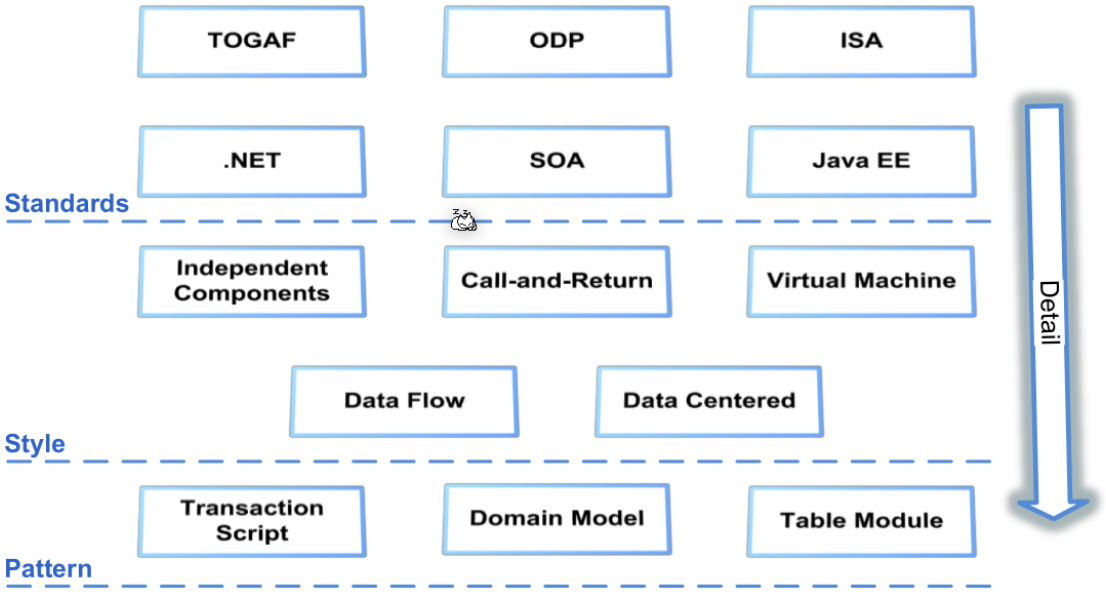


Figure 20: image-20230427092250637

Layering

Importantly, **tiers are not layers**. Tiers are where software is running (e.g. a DB-Server, application server, client, ...). These are not layers. Multiple layers can run on one tier and one layer can run on multiple tiers.

Distribution of Work

- Distributed Presentation: Distributed presentation hides the locality of each client, while the server presents all clients centrally (for example, on one screen)
- Remote Presentation: Remote presentation means that one client handles all presentation tasks, while the application and data are managed by the server
- Distributed Function: Distributed Function implements a division of labor at the functional level between client and server (Cooperative Processing)
- Remote Data: The server keeps all data in one place, no data is stored on the clients
- Distributed Data: The data is distributed on different servers, the client accesses different servers at the same time

Architectural Style

- Independent Components: Independently executing elements that interact with each other via messages
- Call-and-Return: Defined by a fixed communication mechanism between calling and called element
- Virtual Machine: Allows the realization of portable and interpretable systems
- Data Flow: How data flows
- Data Centered: Central task is access and update of data of a repository

Architektur Stil	Anwendung	Vorteil	Nachteil
Independent Components			
Communicating Processes	Parallel processing	Simple modeling, scalability	Complexity of the individual elements
Event Systems	GUI's, Real Time Systems	Independence of elements, change-friendliness	Non-deterministic behavior of the elements
Call-and-Return			
Main Program & Subroutine	Structured Programming, Client-Server (RPC)	Defined control flow	Scalability, expandability
Object Oriented	General design, client-server	Universal	Complexity, freedom of application
Layered	SOA, Multi-Tier Architectures	Conceptual integrity, locality of changes	Performance, complexity
Virtual Machine			
Interpreter	Processor and operating system simulation	Portability, flexibility	Performance
Rule-Based Systems	Expert systems	Flexibility through rules and regulations	Complexity, performance
Data Flow			
Batch Sequential	Host Systeme	Data control	Flexibility, interaction
Pipes and Filters	Software Converter, Compiler	Flexibility, distribution	Complexity
Data Centered			
Repository	Master Data Management	Simple	Single Point of Failure
Blackboard	Data-driven control systems	Scalability	Application limited

Figure 21: image-20230427094948029

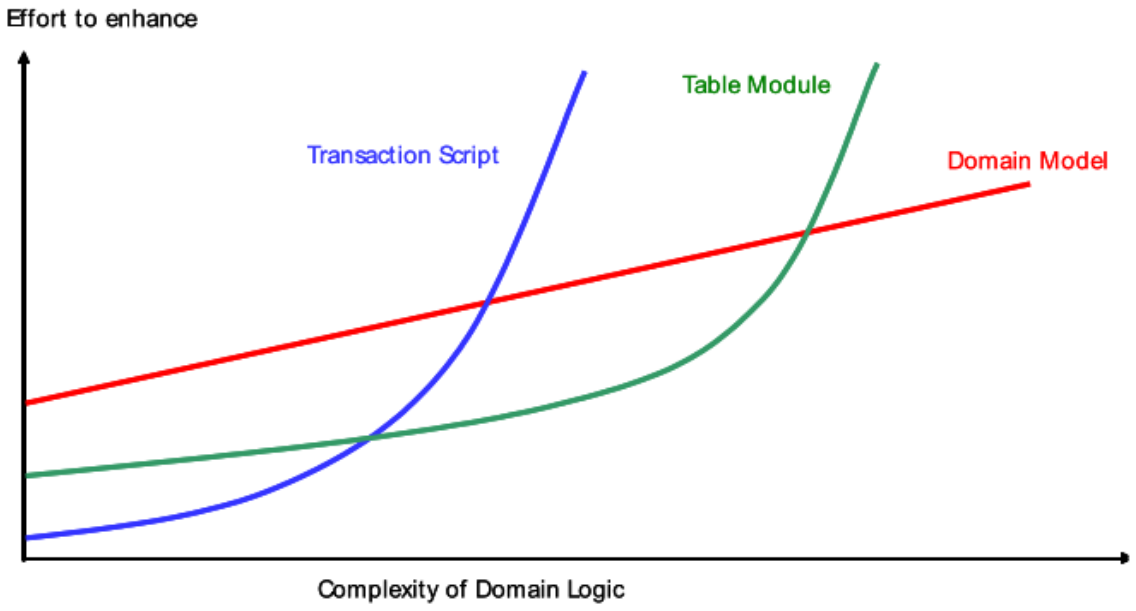


Figure 22: image-20230427100537631

The three Patterns

- The Transaction Script Pattern organizes and divides the business logic into individual procedures so that each procedure covers a single request from the Presentation Layer.
- The Domain Model Pattern describes an object model of the problem domain that includes behavior and data.
- The Table Module Pattern describes a single instance (singleton) that encapsulates the business logic for all rows in a database table or view.

Properties of Modules

Modules should be integrated that they compile, pass tests, run and deploy. Modularization in general enables team development and makes complex system manageable.

Software Craftsmanship

Manifesto

- We will not make messes in order to meet a schedule.
- We will not accept the stupid old lie about cleaning things up later.
- We will not believe the claim that quick means dirty.
- We will not accept the option to do it wrong.
- We will not allow anyone to force us to behave unprofessionally.

These points produced the following manifesto

Coding Kata and Dojos

Programmers solve small problems in a not-job-related code base, repeatedly.

In a dojo, these coding katas are solved in a group. One is programming, one is assisting and the rest is criticising. Who does what is cycling through the group.

SOLID

The SOLID patterns should help a mid-level software structure which can tolerate changes, is easy to understand and can be the basis for reusable components.

S - Single Responsibility Principle

> A module should be responsible to one, and only one, actor.

The single responsibility principle does **not** state that every function/module/... needs to have one responsibility. Rather, it states, that every module should have one actor, which can demand change.

This should be done to limit the impact different stakeholder's demands can have.

Symptom 1 - Accidental Duplication

One symptom of SRP being violated occures when different actors use the same functionality facilitated by the same module (e.g. overtime calculations used by the COO and CFO, but in different ways). If one user demands changes, the developer then implements those, it can be easy to miss the second actor. This will lead to features changing subtly enough (maybe some altered numbers) that nobody notices until it is too late.

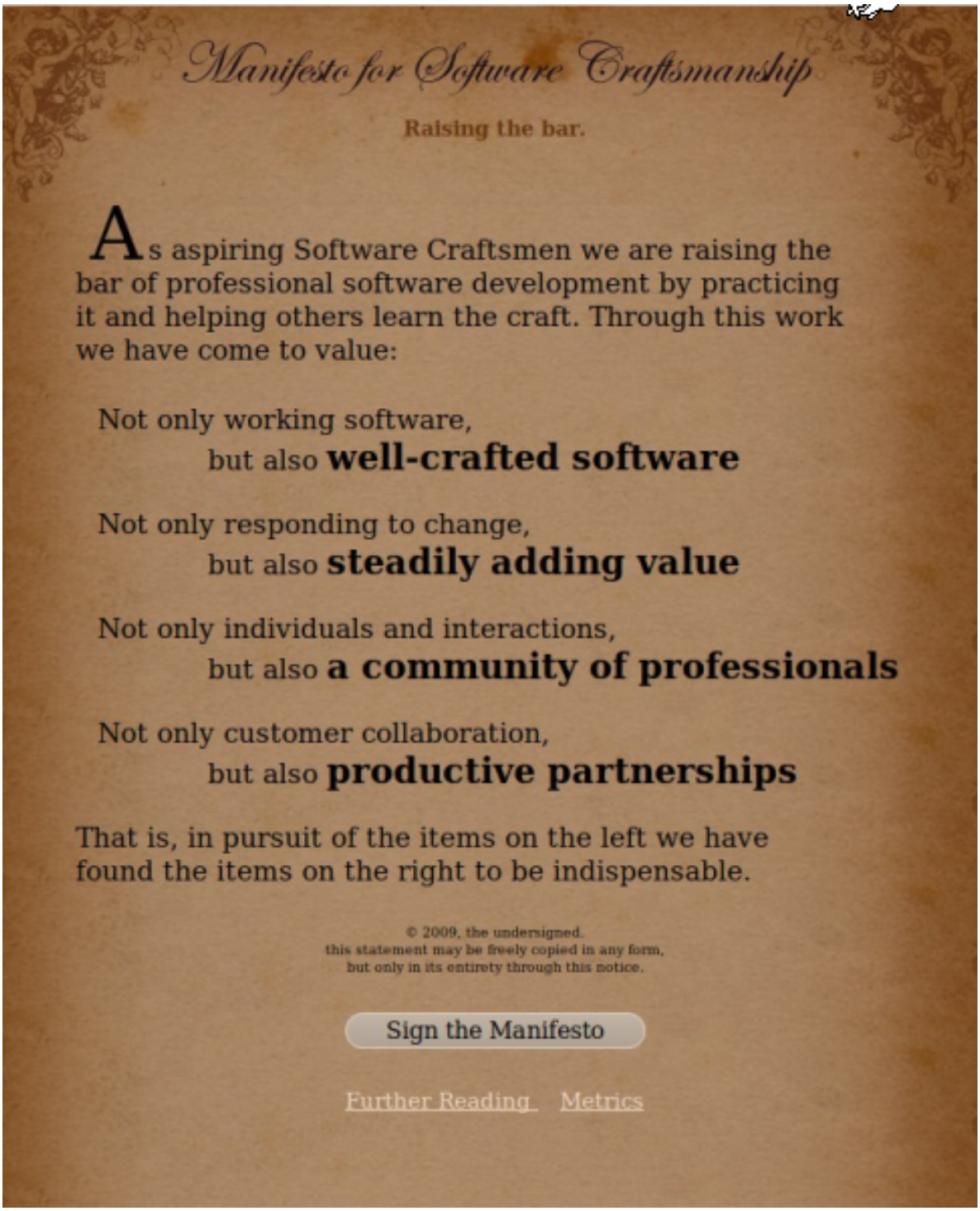


Figure 23: image-20230621104027697

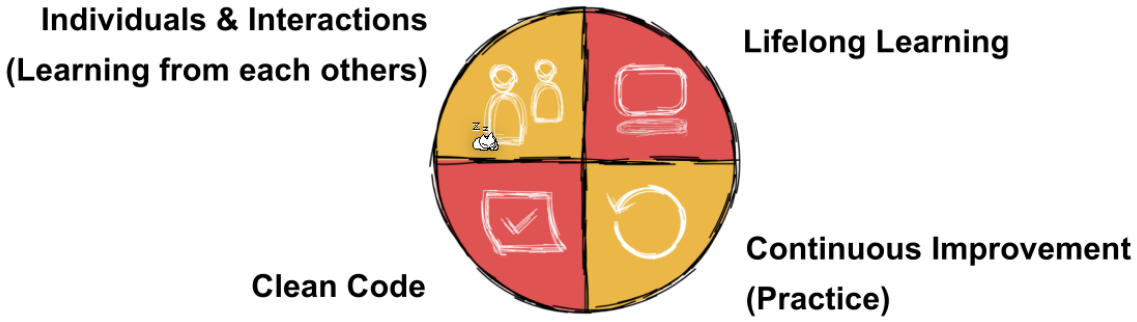


Figure 24: image-20230330093249776

Symptom 2 - Merges

A second problem occurs, when two actors want a change. The changes might be implemented by two different developers, who both will change the same module (as it is shared). When completing the request, a merge conflict is practically guaranteed.

Solutions

One possible solution is to split the data from the functions. To reduce the amount of classes a developer has to deal with, one can introduce a facade, which in turn uses the different logic classes. The classes, that actually implement the logic, mustn't know each other to avoid accidental duplication.

If one prefers to keep the logic closer to the data, one can implement the most important method in the original class and then use a facade for the other functions.

One possible downside of this "pattern" is that it promotes duplication to some part as some shared logic might be needed in multiple implementations.

- Cons:
  - If we have an application where 90% of logic is shared between two actors, should we duplicate those 90%? Probably not, but how much should be duplicated?
    - This even applies to the example: What if a field is added to `EmplyeeData`. Then we have the two symptoms again, so should we duplicate this class as well? Where do we stop? What if the facade needs two change? If this is thought to the end, should we duplicate even the login screen? Probably not...
    - This leads to this principal only being applicable if its already set in stone, what will change in the future (which is almost never the case)
  - One of the scrum principals is, that we should forget about the future. Adding unneeded complexity kills your project. This seems to be a primary case of adding complexity when its not clear if it is needed in the future.
- Mention that this pattern exists on the functions and class level, the component level (Common Closure Principle) and the architectural level (Axis of Change)

O - Open Close Principle

> Software entities should be open for extension but closed for modification

This means, functions should be easily be added to existing code, but existing code shouldn't be altered. To do this, while writing code, ask your self, can more functionality be added to this code.

Advantages:

- Maintainability Easier to add code
- Flexibility
- Scaleability

One way to archive the open close principle is by designing the architecture in a way that arrows between "modules" only go in one direction. This can be seen on the diagram below. All arrows point towards `Interactor`. This results in `Interactor` being protected from changes from `Controller` and `Database` since it doesn't even know about these modules. This also means, that not all modules are equally protected from changes and there is a hierarchy of protected-ness (namely `ScreenPresenter` = `PrintPresenter` < `Controller` < `Interactor`)

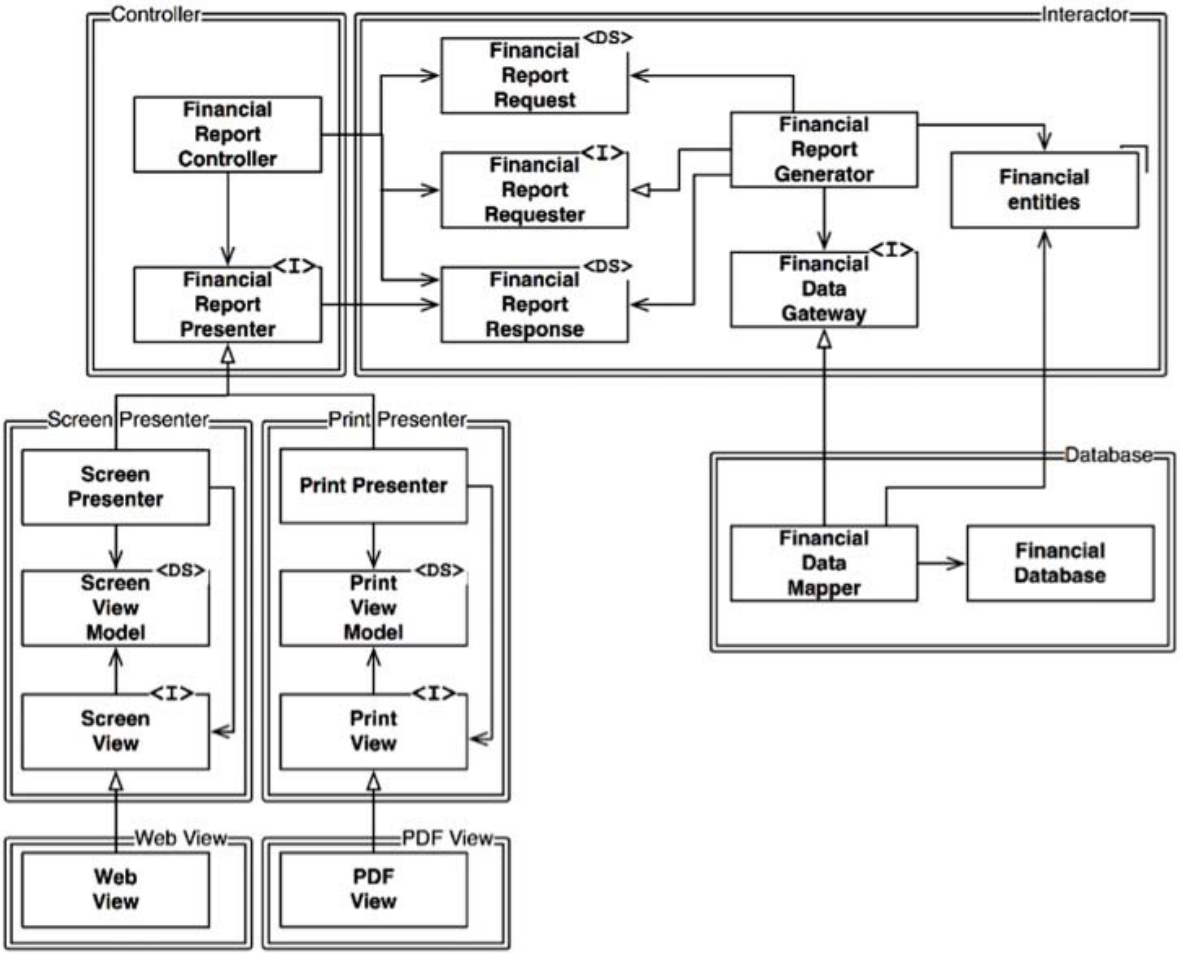


Figure 25: image-20230621201821788

L - Liskov Substitution Principle

> What is wanted here is something like the following substitution property: If for each object `o1` of type `S` there is an object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behavior of `P` is unchanged when `o1` is substituted for `o2` then `S` is a subtype of `T`.

The following square-rectangle example is the canonical violation of this principle. While `Square`

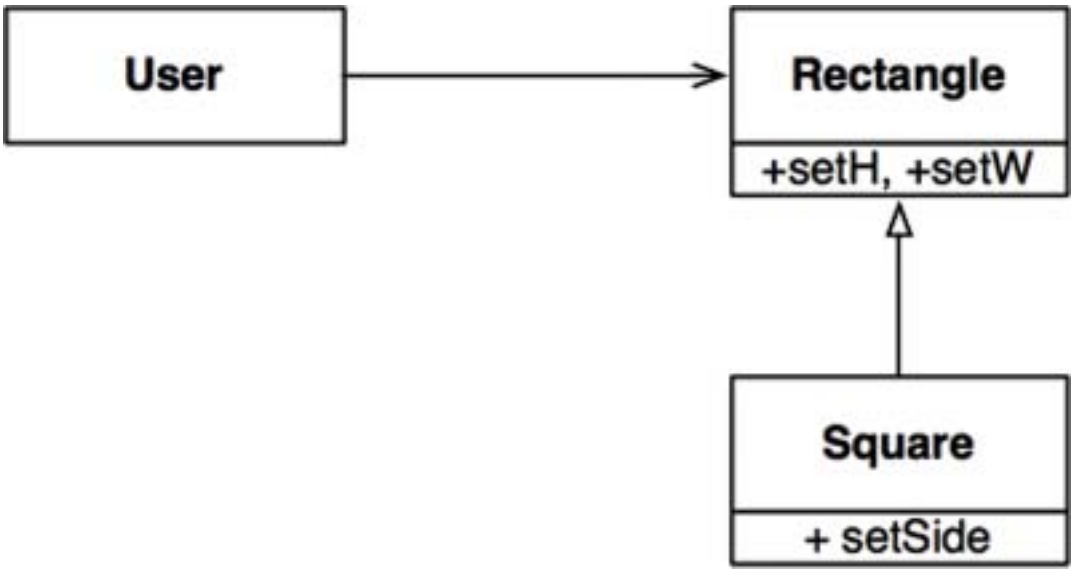


Figure 26: image-20230621203130187

inherits from `Rectangle`, it behaves different from a rectangle. This can be exemplified with the following code:

```
Rectangle r = ...
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

If `r` were to be a `Square` this method will likely fail. The only way for a calee to deal with this, is to add an if statement to differentiate between `Rectangle` and `Square`.

The following is an example, where the Liskov Subsitution Principle was up held. Both `BuisnessLicense` and `PersonalLicense` behave the same and `Billing` doesn't have to know which implementation it uses.

I - Interface Segregation Principle

> An object should only import what is uses

In the diagram above `User1` might only use `+op1`, yet they still have to import the whole `OPS` structure. If there were changes to `+op2` and `+op3`, `User1` probably still has to be redeployed.

This can be avoided by reworking the architecture in the following way:

D - Dependency Inversion Principle

> Avoid dependencies on volatile concrete classes

Depending on concrete implementation can be dangerous, since changes to those concrete implementation force changes to dependencies. This can be fixed by using abstract types, like abstract classes and interfaces, instead of their concrete implementation.

With that being said, there are always concrete dependencies (e.g. the `String` class in Java). Non-volatile concrete implementation, especially when defined by the platform, can be used without building

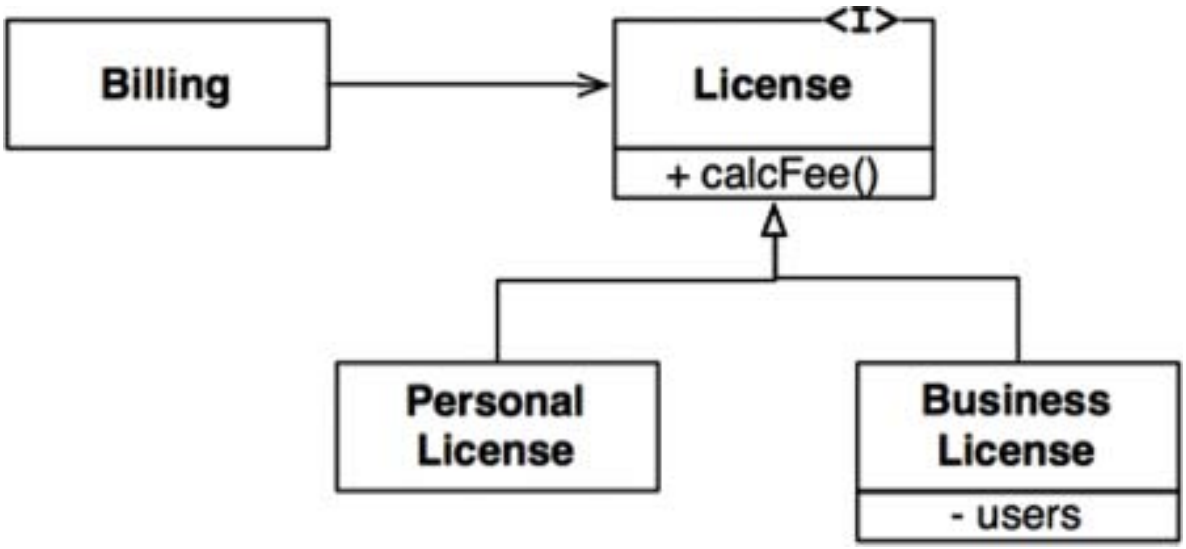


Figure 27: image-20230621203432538

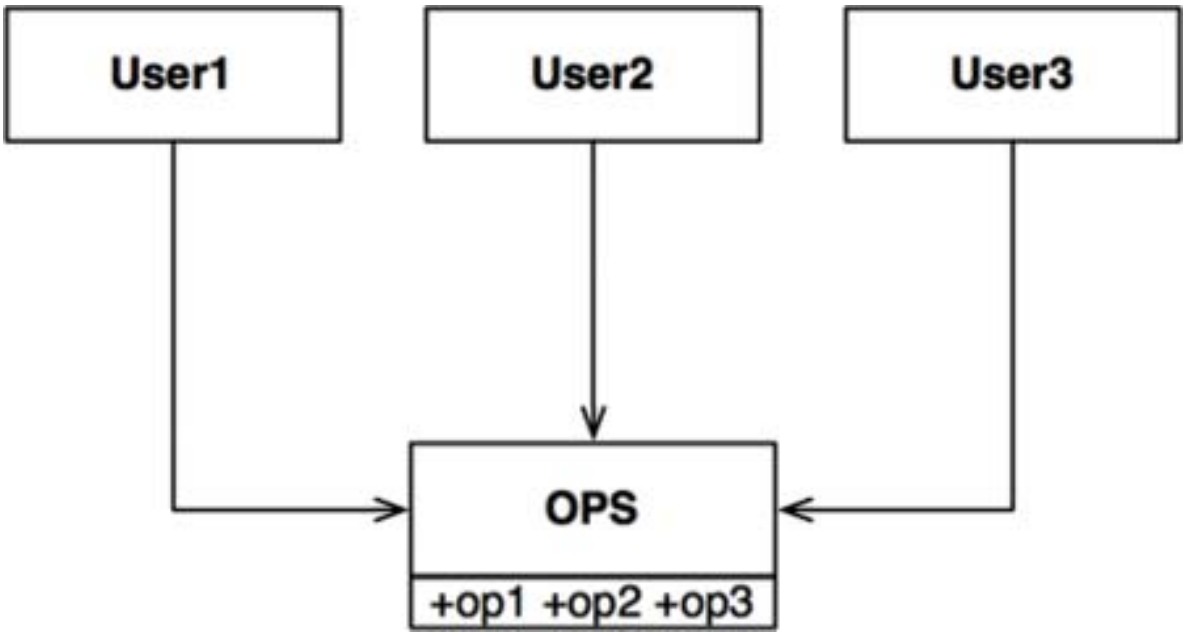


Figure 28: image-20230621203557253

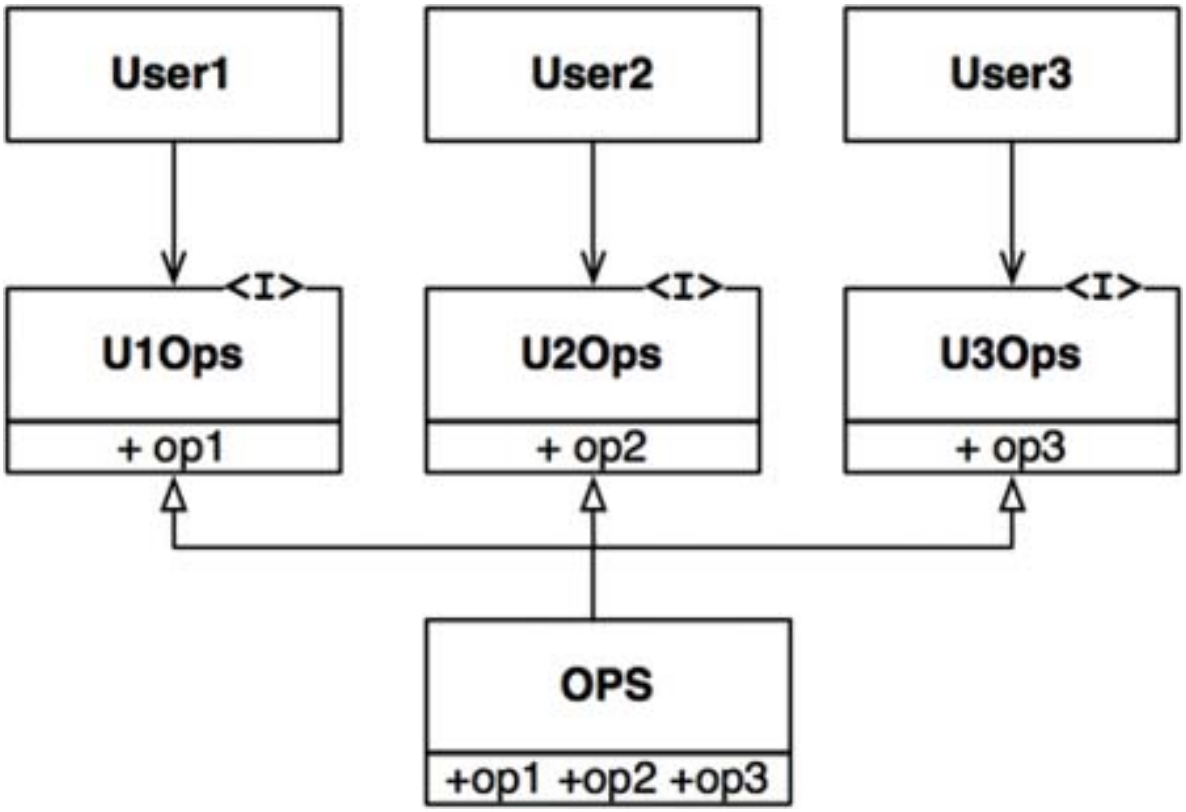


Figure 29: image-20230621204100452

needless abstractions.

However, references **volatile** concrete implementations should be avoided.

These following coding practices emerge from this:

- Don't refer to volatile concrete classes
- Don't derive from volatile concrete classes
- Don't override concrete functions
- Never mention the name of anything concrete and volatile

When creating a new object, in most programming languages, this requires a concrete implementation. This can be solved by employing abstract factories.

Most systems have dependencies to concrete implementation, which can't be architectured away. This is fine.