PTS

1. **Metódy vývoja softvéru, softvérové kontrakty** (vodopádový model; I-I vývoj; agile; softvérové kontrakty)

- (Business analysis)
- Requirements
- Design
- Implementation (Construction)
- Verification and validation
- Deployment
- Maintenance

- Waterfall approach: Every phase starts after the previous one finishes.
- Iterative-incremental approach: You select a small part of the system, build it in a waterfall style.

Maybe the right question is: how many iterations should you do, long should an iteration take?

Waterfall approach

- Verification and validation at the end of the process.
  - Errors within any phase are found only at the end of the project (especially bad for errors in requirements).
  - We cannot evaluate whether the objectives of the project are attained during the process.
- For long projects, it is very natural that the requirements change
  - We may not have enough knowledge on how to attain the goals of the project.
  - Outside interference (Change of strategy within the company, legislative change, etc.).
  - Change avoidance may impact the quality of the product.
- Unknowns may exist when the requirements are fixed.
  - Can this even be implemented, how fast, under which design?
- You cannot deploy partial product.

Waterfall approach:

+ Easier to manage (in most aspects).
+ Easier to determine scope, time and price.
+ Allows to focus on each phase separately.
+ Guarantees that a reasonable documentation exists.

Iterative approach:

+ Allows to deploy partial product
  - Best way to clarify the requirements and the scope of the project.
+ Deals easier with changing requirements.
+ Faster feedback (not only with respect to requirements)
- Requires flexible design and higher quality of code
- It is hard to determine scope, time, and price
  - This is a common requirement in the industry.

Iterative approach is typically superior. When is waterfall approach appropriate (or longer iteration / larger increments are appropriate)

- Requires stable and well defined requirements, not many unknowns in the solution.
- It is not easy to test or deploy the solution.
- High requirements on safety and reliability.
- You need to determine price, scope, and time fast (You need to waterfall at least until requirements / analysis).

Fast feedback is valuable.

- It is easier to correct an error if you detect it fast.
- You can avoid similar errors in the future.
- You can evaluate the degree the project can fulfill its objectives faster and more accurately (fail fast).
- Some feedback is more valuable than other.
  - It is important in which order we build our application.

Sometime you plan not to use what you have done:

- Prototyping
- Evolutionary

Prototyping - why to build something that will be thrown away?

- Faster and cheaper feedback

Other two main aspects of running a project

- Predictive planning
  - The focus is on predicting time, and money necessary to produce given scope
  - Works well if there are not too many unknowns.
- Adaptive planning
  - The focus is on what should be build.
  - Constantly adapting the plan according to the feedback.

Agile

Not a software development process (this is quite common misconception). But it implies

- Iterative development.
- Adaptive planning.

It is defined by the set of principles proposed in Manifesto for Agile Software Development

Software contracts

Is there a difference between buying software and tailor-made furniture?

- Software is an incredibly complex product.
- Even if you know what to build it is hard to specify everything.
- A big issue is the lack of vocabulary (customer is familiar with different domain).
- Very often the effect of the produced software on the customer is unknown.

Fixed price, time, and scope.

- You need to know the scope of the project
- Implies waterfall at least until reasonably detailed requirements are set.
- It is hard to change requirements.
  - Contracts typically contain measures how to change the scope (change boards), but the software builder has mostly an upper hand in the negotiations.
  - Changes can significantly increase the price of such project (Is the price then really fixed?)
- Limited feedback.
- „Perhaps the reason these type of contracts (Fixed price, time, and scope) survive is because they can be defended in court."
  - Allan Kelly
- Risk on the software development side is on the developer
- The customer takes the risk that the requirements are correct

Everything is OK it this suits you case.

In the case there are unknowns you cannot fix all of these

- price
- time
- scope

At best, you can pink two of these.

Iterative development

- Collaboration is encouraged.
- Thus it make sense to share the risks instead of sharply dividing who is responsible for what
  - While collaborating, it is often hard to say who is at fault.
  - Lawyers are expensive, the resources could be better used elsewhere.

Agile contracts

Key features

- Risk sharing (both parties are interested in project success)
  - Business goals not attained, late delivery, higher cost, smaller scope, ...
- Ability to terminate the contract at several points without major penalty.
- Contract defines processes, not requirements
  - Fail fast if one side of the contract does not believe the project.
- No detailed requirements just high level requirements and project goals.
- Measure of success is related to project goals, not requirements.

Cooperation is instrumental in such contracts.

Agile contract principles / choices

- Manhour payment.
- Recouring contracts.
- Payment for product (and product) only if satisfactory.
- Capped time / materials.
- Money for Nothing, Change for Free.
- Contract based on business goals.

Software tenders

- Fixed price and time.
  - Project size estimation by estimating representative epics.
  - Parallel development with various contractors at the beginning.
- Contracts based on measurable business outcomes.
  - Such a contract has various parameters which could be the evaluation criteria
- Agile contracts.
  - How to find a contractor in a meaningfull way while preserving required transparency?

2. **Manažment konfigurácií** (ciele manažmentu konfigurácií; typy VCS; git; commit; branch; merge; rebase)

- Software configuration management is the task of tracking and controlling changes in the software. It includes tracking changes in source code, documentation, and other artefacts.
- Primarily done using Version control systems (VCS).
- Some other systems can be useful in this context (e.g. Issue tracking systems)
  - For all artefacts it is known where they are.
  - More versions of the same artefacts.
  - Multiple people working on the same artefacts concurrently.
  - Storing historical versions of the artefacts.

We should track exactly what is necessary to build, run, and work on the project.

- Manually written source files. Yes
- Generated source files. No - but we need to save the artefacts needed to generate the source.
- Final binary. No.
- Images. Yes.
- Requirements. Yes.
- Deployment scripts. Definitely.
- Compiler. Well, maybe ...

Commit:

- Creates a new version of the system
- Unit of change in VCS
- Each commit should make sense on its own.
- A single commit should not be easy to divide to more commits..
- After a commit the project should remain in a sound state (what sound means varies, e.g. development branch vs mainline branch).

Branch:

- Separate copies of the system.
- A commit affects only one branch.
- Branches can be created and merged with other branches.
- There are various reasons to have slightly different copies of the system (development, major releases, experimental).

What is GIT?

- Distributed version control
- Created for the development of Linux kernel
  L. Torvalds: I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'.
- GITHub - web based version control repository and Internet hosting service – **do not confuse it with git**. Alternatives include GitLab, BitBucket, SourceForge, . . . ,
- GIT is just one particular VCS, there are alternatives too, e.g. CVS, SVN, . . . Some of the above services support other VCS than git.
- Version control services have many other features to manage projects unrelated to git.

There are three main levels of configuration:

- computer level (--system)
- user level (--global)
- project level

You need to set

- Name
- E-mail address

You want to set

- Your favorite text editor to write commit messages and other stuff

Create repo: git init/ git clone

File states: untracked, unmodified, modified, staged

Basic commands:

- `git status`: shows the state of the files
- `git add`: changes the state to staged
- `git rm/mv`: if you delete/move files, let git know
- `git commit` (`git commit -a`)

You may want to do other stuff:

- `git diff` (or use `gitk`)
- `git reset HEAD <file>`: unstage
- `git checkout (--) <file>`: throw uncommited changes
- `git commit --amend`: change last commit

View commit history: git log/ git blame/ gitk

- `git branch`: shows branches
- `git branch <name>`: creates a branch
- `git checkout <branchname>`: change branch
- `git branch -d <name>`: delete branch

- `git merge` - merges some other branch into current branch, the merged branch still exists.
  - git tries to merge stuff automatically
  - if he does not know what to do, it lets you resolve the conflicts
  - the new commit has links to two commits (top commits of both branches)
- `git rebase` - alternative to merging
  - gits try to apply the commits in other branch one by one
  - it tries to resolve conflicts
  - if he does not know what to do, it lets you resolve the conflicts (this may happen multiple times during a rebase)
  - the commit history is linear (good for bisecting)

Remote repository [11]

Basic commands

- `git clone`
- `git pull`: Incorporates changes from a remote repository into the current branch
- `git push`

Other stuff

- `git push origin -delete "branchname"`
- `git push --force`:
  - changes commit history
  - do not do this if more people are working on the branch
  - e.g. before merging to master you create a better history, then you need to force push it.
- `git fetch`: just downloads from remote repository, without merging to current branch
- `git remote`: manage repositories.

How to pull while you have uncommitted changes and you do not want to lose them?

- git stash
- git pull
- git stash pop - may create a conflict that needs to be resolved

Stash works like stack, and has many other uses

- Each commit is identified by a part of its hash.
- `HEAD`: What we see in the working directory, normally top of the branch, however we can move wherever we want by `git checkout`.
- `HEAD~i`: points *i* commits back.
- `git revert <commit>`: This does not change the history, just adds a new commit.
- `git reset --hard <commit>`: This changes the history.
- `git rebase -i HEAD~k`: interactive rebase is a good tool to adjust history
- `git tag`: tag important commits (version bumps, etc.)

There are various possible workflows. Example

- master branch
- development branch - merges to master only on important milestones
- feature branches - merges to development branch only when the feature (or an important part of the feature) is finished

3. **Modelovanie domény** (model; želané vlastnosti modelu; ciele modelovania domény; typy vzťahov medzi triedami v UML diagrame tried; abstrakcia atribútov, abstrakcia typov, abstrakcia závislostí)

System, model view[1]

- System: A set of elements organized to achieve certain objectives form a system. Systems are often divided into subsystems and described by a set of models.
- Model: Model is a simplified, complete, and consistent abstraction of a system, created for better understanding of the system.
- View: A view is a projection of a system's model from a specific perspective.

There are two distinct things to model while creating software

- Domain model - models the domain and the concepts from the domain.
- Model of the SW system we build.

While modeling SW system, modeling with various level of detail is useful

- Classes, most important attributes, relations between classes.
- + most of the methods, however, some aspects are omitted for model simplicity
- Full implementation model, can be used as a template for implementation

What makes a good model? [2]

- Clearly specified **object of modeling**, that is, it is clear what thing the model describes.
  - an existing artifact
  - physical system
  - a collection of ideas about a system being constructed
- **Specified purpose** and contributes to the realization of that purpose
  - communication between stakeholders
  - verification of specific properties (safety, liveliness, timing,..)
  - analysis and design space exploration
  - code generation
  - test generation

A model can be descriptive or prescriptive. If a model has to serve several distinct purposes then often it is better to construct multiple models rather than one.

- **Traceable**: each structural element of a model either
  - corresponds to an aspect of the object of modeling,
  - encodes some implicit domain knowledge,
  - encodes some additional assumption. A distinction must always be made between properties of (a component of) a model and assumptions about the behavior of its environment.
- **Truthful**: relevant properties of the model should also carry over to (hold for) the object of modeling.
  - In the construction of models often idealizations or simplifications are necessary.
  - The modeler should always be explicit about such idealizations/simplifications, and have an argument why the properties of the idealized model still say something about the artifact.

- **Simple**
- Extensible and reusable.
  - Ideally, it should be possible to model a whole class of similar systems.
- Interoperability and sharing of semantics.

Some popular approaches to analysis and design

Object-Oriented Analysis And Design

- It's a structured method for analyzing, designing a system by applying the object-orientated concepts, and develop a set of graphical system models during the development life cycle of the software.
- Prevalent style of analysis and design (the amount of visualization varies), including if the system is not built using OO programming.

Domain driven design:

- placing the project's primary focus on the core domain and domain logic;
- basing complex designs on a model of the domain;

UML

Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. [4]

- Developed in 1994 by Booch, Jacobson, Rumbaugh at Rational Software
- Managed by Object Management Group, published as a ISO standard
- Most recent revision is 2.5.1 from december 2017.

UML does not have a compiler $\rightarrow$ older versions are frequently used, adaptation of new features is slow.

Types:

- Use case diagrams
- Activity diagrams
- **Class diagrams**
- Sequence diagrams

UML Relationships

- Association, N-ary association, association class,
- Aggregation,
- Composition
- Generalization
- Dependency (e,g,. use, call, create, required interface, interface realization).

UML Relationships

Examples:

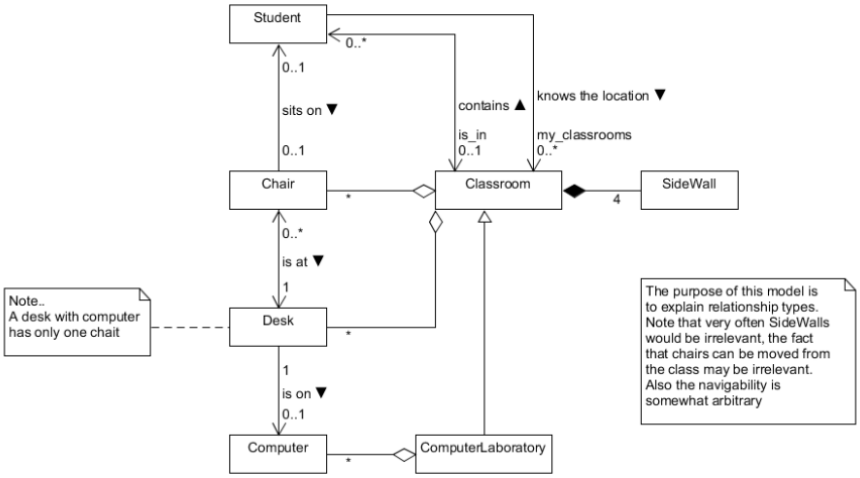
- Association - class has an attribute that refers to the other class
- Aggregation - has a list of references to other class. Many believe, that one should use association instead of aggregation [6] unless you add a specific meaning to it.
- Composition - the class owns instances of the other class, it is responsible for their life cycle.
- Generalization - subclass superclass
- Dependency
  - use, call - gets a reference as a method parameter and does something with it (calls a method)
  - create - e.g. factory pattern
  - required interface - e.g. interface required in constructor or in method call,
  - interface realization - implements interface

UML - Relationship properties

- naming the relationship (especially association), naming can have a direction to make it more meaningful
- naming the endpoints
- multiplicity
- visibility (not to be confused with the direction of relationship name)
- constraints

UML - how to draw a class diagram

- Minimize crossings (if you need many crossings, your model is too complex and probably lacks some abstraction)
- Don't repeat yourself
  - If a class A is associated with class B and e.g. the end of the relationship at B has the name c, you **should not** class A should not have attribute c of type B.
  - If the name of the attribute end is b (or bs i in case to to many multiplicity) the naming is unnecessary.
- The diagram should have a subject of modeling. This includes time scope. Some relationships are static within the scope, some are created and destroyed. Your model should have means to do this.
- The model should be able to perform use cases within its scope.



How to make a model describing complex world simple and truthful? Abstraction

- Abstract from things that are not relevant to the problem.
- Question relevancy of apparently obvious classes and relationships.
- Should a concept be a subclass or is it just an attribute?
- Develop concepts that capture common characteristic of various elements in the model.

Abstraction of types:

- Instead of separate classes, one can just have type properties
- We can control which values are allowed for which property using a relation between properties and values

Abstraction of attributes

- Instead of attributes one can have just a list of attributes
- We can control which attributes are allowed for a given type using a relation between type and properties

Abstraction of relationships

- Instead of many relationships we can make a relationship class that generalizes all of them.
- This can be used not only to generalize relations between two classes
- We can associate relationship types with its allowed operands.

4. **Dizajn** (ciele dizajnu; dizajnové princípy; SOLID)

Modeling and design

Common levels of abstractions:

- Classes, most important attributes, relations between classes.
- \+ **most of the methods, however, some aspects are omitted for model simplicity**
- Full implementation model, can be used as a template for implementation

Aspects that may be omitted early in the design process typically include concurrency and persistence.

- However, you should have your general approach settled as these are typically crosscutting concerns.
- You just do not need to decide exactly where each lock is, etc.

Given the right decision was made at the architecture level your system is modularized enough that a subsystem you are dealing with

- Is well defined (defined scope and interfaces)
- Is small enough to be comprehensible by a single person.

Goals for the design at this level:

- Modularization
- Abstraction
- Information hiding
- Separating interface and implementation
- Low Coupling (few dependencies between parts)
- High Cohesion (parts do just one thing)
- Sufficiency, Completeness, Simplicity, Flexibility ...

You want most of thes on other levels too, but there are subtle differences.

Modularization

- focused mostly to modularize implementation and verification while at architectural level we are more focused on modularizing requirements, documentation, etc.
- this impacts the result to some degree.

Abstraction

- focused on stuff like simplicity, avoiding repeated code

Information hiding, Separating interface and implementation, Low Coupling, High Cohesion

- to preserve modularization over time
- not necessarily that strict for tightly related parts

- Abstraction (techniques from domain modeling are applicable to the modeling of the software system)
  - This includes stuff like abstraction of types, relationships. etc.
  - it is good idea to perform these simplifications before refining the model
- Don't repeat yourself, Rule of 3
- You ain't gonna need it
- ...

For object-oriented design:

- SOLID
- Dependency injection

You can verify design:

- Can you really perform all the function prescribed by the interfaces you have to implement?

It is much harder to validate the design, example:

- We expect aspect A of the system to vary, thus our design makes it easy to change aspect A
- Validation means confirming that this assumption is correct.
- This can be done e.g. by evaluating past changes (even before the development of our system started).
- Often it is hard to predict future changes - the flexibility of design is important (however one needs to balance the flexibility and the added complexity).

YAGNI

You ain't gonna need it.

- Relevant also on the architecture level
- Unless the evidence says otherwise, you should prefer simplicity over
  - additional features
  - excessive flexibility
  - performance
  - etc.

This does not mean that you should not apply basic principles while creating the design. However, you should avoid doing complex non-standard stuff unless you have really good evidence you need to.

DRY

Don't repeat yourself

- Repeated code is
  - longer
  - harder to maintain
  - error-prone
- You should avoid repeating the code.
- It is always possible using the right technique.
- Tradeoff: sometimes avoiding the code repetition creates some quite hard to grasp abstraction (e.g. template method pattern with many parameters) - goes against simplicity

Rule of 3

If in doubt, apply (rule of thumb) Rule of 3:

- If similar piece of code appear three times, change the design so that the code is not repeated.
- The idea is, if the code appears for the third time, it is likely that it will happen even more, in which case the change is utterly necessary.
- If you have code that is repeated twice document this very carefully (e.g. comments in the code referring to the other part and vice versa).
- Similar reasoning can be applied elsewhere (e.g. if some aspect of the system changes three times, you should change your design so that the change is easy to do)

Low Coupling

Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules [1].

- A change in one module usually forces a ripple effect of changes in other modules.
- Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
- A particular module might be harder to reuse and/or test because dependent modules must be included.

- Information expert principle (rule of thumb for low coupling)
  - A Responsibility belongs to a class which has most data to perform the task
- Coupling strength, Coupling distance
  - It is more acceptable to have high coupling between classes with small distance

Cohesion - cohesion refers to the degree to which the elements inside a module belong together [2].

- Example metrics LCOM4:
  - Create a graph, vertices - attributes and methods, edges method calls a method or uses an attribute
  - LCOM4 = number of components
  - If LCOM4 $\neq 1$, the class is not very cohesive.

Encapsulate what varies

If some functionality (e.g. method of a class) changes a lot we should encapsulate this functionality

- Create a new class whose concern is exactly this behavior
- Hide behind an interface (strategy pattern)
- This protects the rest of the class from the frequent changes.
  - protects the original class against newly introduced errors.
  - allows to narrow the scope of the change making it easier.

Alternative approach: abstraction - design things so, that the changes just affect input data (e.g. a change in a configuration file)

Object oriented design

Information hiding, Separation of interface and implementation

- Encapsulation does some information hiding, however, interfaces are a more robust tool to do this.
- Implementation depends on the interface and not vice versa
  - This is general principle valid also outside OOP.
  - OOP separates interface from implementation using polymorphism.
  - The caller (who requires interface) needs not to be aware of who exactly the callee (implements interface) is - polymorphic dispatch.
  - The caller code should not need to recompile upon a change in the callee code.
  - Private/public implicitly define an interface, however, this tool may not be sufficient.

SOLID

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle (Liskovej substitučný princíp)
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

This implies:

- high cohesion, LCOM4$= 1$
- class has a responsibility

How to separate other responsibilities? E.g.. strategy pattern.

Open-closed principle

Software entities should be open for extension, but closed for modification.

How to change a class without modifying it?:

- Inheritance
- Composition (preferred, *"Composition over inheritance"*)

How to compose objects without creating an explicit dependency?

- Dependency injection (via constructor/ method, described in the next lecture)

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- Inheritance is not as IS-A relationship
- e.g. Square is a rectangle, but square should not be a subclass of a rectangle.

Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface.

- Segregates interface from implementation
- Makes the interface easier to use in the client code
- Limits the impact of the interface change
- You can use Adapter pattern to adjust the class to the required interface.

Dependency inversion principle [4]

One should "depend upon abstractions, not concretions

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.
- By dictating that both high-level and low-level objects must depend on the same abstraction, this design principle inverts the way some people may think about object-oriented programming
- If you apply this principle 100% you should have an interface between each cooperating classes. This may not always be most practical, however, more distant parts of the system should be separated by interfaces and adhere this principle.

Dependency injection and dependency inversion

- A class should not create instances of other classes (if we do sociable testing it might make some exceptions of this rule for closely related classes) - dependency injection / dependency injection of a factory.
- A class should not depend on the implementation of the collaborator, just on the interface - dependency inversion

Dependency injection is a way to implement **dependency inversion**.

5. **Návrhové vzory a UML diagramy** (návrhové vzory; UML diagram tried; UML sekvenčný diagram; code smells; refaktorizácia)

UML Sequence diagrams

- UML class diagram with several additional comments in text form is often sufficient to explain design.
- In some more complex cases one needs to provide detailed dynamic description of how the classes accomplish a goal.
- UML Sequence diagrams provides dynamic view of the described system.

- UML class diagram with several additional comments in text form is often sufficient to explain design.
- In some more complex cases one needs to provide detailed dynamic description of how the classes accomplish a goal.
- UML Sequence diagrams provides dynamic view of the described system.

- Have a quick look at the examples and the reference.
- Note that synchronous and asynchronous calls are distinguished.

Design patterns

- Are reusable solutions for design.
- Give terminology to ease speaking about design.
- Known solution is easier to understand.
- Provide inspiration even if the case is not covered by a design pattern.
- They may indicate missing features in the programming languages.
- Some of the patterns are included into languages (e.g. Decorator in Python).
- Duck-typing languages need fare less elements to attain the same goal. Even if some elements (e.g. interfaces) are not in the code explicitly, they are still there implicitly.

Decorator pattern - There are other viable solutions to this problem, but

- many of the other solution are less flexible,
- other solutions are much harder to explain - if you use the pattern a single word “Decorator” is enough to describe several classes.

Design pattern Gang of Four types:

- Creational
- Structural
- Behavioral

But we have also

- Concurrency patterns
- Domain-specific patterns
- ...

Creational design patterns

- **Factory method** - a method (which may be static but is not a constructor) of a class, which returns new instances of some other class.
  - The method may return instances of more than one class depending upon the parameters received.
  - The caller does not need to know the exact class of the instance (just an explicit or implicit interface it has to satisfy).
  - A class containing a factory method can be injected into a class calling it e.g. in constructor. We can replace the factory if we need to produce instances of a different class.
- **Abstract factory** - An interface containing several related factory methods.
  - Generally, there exists more implementations of the interface.

Factory method - Example

- A graph may be represented with adjacency matrix or adjacency list
- We want to create graphs in our class A, but which representation is desirable is outside the scope of the class.
- We need the following
  - GraphFactory - interface defining the factory method
  - SparseGraphFactory and DenseGraphFactory implement GraphFactory
  - Graph - interface that contain some graph methods (so we can actually do something with the graph even if we do not know the implementation).
  - SparseGraph and DenseGraph implement Graph.
  - Our class A that creates graphs takes GraphFactory in its constructor.
  - According to the implementation of Graphfactory we provide class A either creates SparseGraph or DenseGraph instances.

- **Builder** - A class that is used to incrementally create or modify instances of other class..
  - If creating or modifying an object is complex we could end up with two sets of methods, one for the building / modifying phase and second for the actual usage phase. This violates single responsibility principle.
  - Distinct data structures may be needed in each phase.
- Example: Java StringBuilder

Builder - example

- Graph has methods addEdge(...), removeEdge(...), addVertex(...) (represented by adjacency lists)
- We want to replace the edge by two consecutive edges incident to a new vertex.
- We can: remove edge, add vertex, add two edges.
  - We have to manipulate the lists a lot.
  - This is especially bad if we do many such operations
- GraphBuilder - we add the new vertex and write its new adjacency list. Note that now we do not have a graph now, thus it is good that we have a distinct clas in this situation. We correct the entries in the adjacency lists of incident vertices, We have a proper graph again and we can convert the result to Graph.

- **Object pool** - Instead of creating / destroying a class we just take an instance from / return an instance to a pool of objects.
  - Useful if it is hard to create an instance (threads, connections, etc.)
  - We can control the resources by providing limits on the number of instances available.
  - typical use: ThreadPool, ConnectionPool
- **Prototype** - New instances are being created by copying a prototype.
- **Singleton** - A class that is guaranteed to have only one instance.

Structural patterns

- **Decorator**
- **Composite** - Treelike structure to compose objects.
- **Facade** - Class giving an easy access to whole subsystem.
- **Adapter** - Class that modifies the methods of another class to satisfy an interface.
- **Proxy** - An object representing another object.
  - Access proxy, remote proxy, virtual proxy, ...
- **Flyweight** - We divide a class into a part that is common for many instances and a part that is specific for each instance.

Behavioral patterns

- **Iterator**
- **Observer** - Notify objects of about changes.
- **Strategy** - Encapsulation of an algorithm.
- **Template method** - Method in an abstract class using other abstract methods.
- **Null object** - Sometimes reasonable "default" exists which can be returned in case of failure.
- **Memento** - Keep and reconstructs a state of an object
- **Visitor** - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Code smells

Code smell is

"a code smell is a surface indication that usually corresponds to a deeper problem in the system" - - M.Fowler - -

- Code smell is by definition not a bug.
- May indicate technical debt.

Refactoring is a process of restructuring the source code without changing its external behavior.

Incorporating refactoring into software development process

- Refactoring should be a incorporated into our software development process

Example: Test-driven development

1. Write a test.
2. Check if the test fails (this is, according to my experience, unexpectedly useful)
3. Write code
4. Check if the test passes
5. Refactor
6. Check if the test passes

Note the division between adding functionality and refactoring, this is important.

Code smells - examples

Code smells - Sourcemaking

- Long class
- Too many arguments in a method
- Switch statement
- Parallel inheritance hierarchies
- Repeating code
- Too many comments (even in the case they seem useful)

Refactoring - examples

Code smells - Sourcemaking

- Decompose complex conditional
- Extract Method
- Extract Variable
- Replace Nested Conditional with Guard Clauses
- Introduce Parameter Object
- Form Template Method

6. **Testovanie a kvalita softvéru** (zabezpečovanie, plánovanie a kontrola kvality; techniky na zabezpečenie kvality softvéru; testovacia pyramída; testovateľný kód; dependency injection)

What is quality of software

- Software functional quality (External)
  - how well it complies with or conforms to design and functional requirements
  - fitness for purpose, how it compares to competitors in the marketplace
  - **degree to which the correct software was produced**
- Software structural quality (Internal quality)
  - how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability.
  - **what is under the hat**, lack of technical debt

Software quality management:

- Quality assurance - Setting up processes and standards within the organization.
- Quality planning - Define the quality attributes associated with the output of the project and how those attributes should be assessed.
- Quality control - Reviews software at its various stages to ensure quality assurance processes and standards are being followed

To create quality software you need to:

- Measure attributes associated with quality
- Control and enforce them

This has to be an integral part of the development process.

Structural quality:

How to enforce code readability?

- code reviews
- linter (especially important in languages with a lot of backward compatibility heritage)
- pair programming
- ...

Automated tools can be integrated into the tools used (e.g. git hooks).

Functional quality:

- Verification - The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition.
- Validation - The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders.

You should verify and validate not only the whole product but all artefacts.

- Example: How to validate requirements?
  - Customer may read the requirements document (however, this is often insufficient)
  - Prototyping, Test case generation, ...
- Verification - The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition.
- Validation - The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders.

Verification

- Manual - preferred only in certain circumstances
- Automated - seems like a time wasting sometimes but actually saves a lot of time in projects that take longer than a few days.

Testing:

- Defect detection techniques (the numbers are probably not too precise, but you should check it out to see what the techniques are).

Testing pyramid

- Small-scale automated tests (**unit tests**) are fast - we can and should perform a lot of them.
- Medium-scale test (**integration tests**) - they check, if the smaller parts work well together. You do not need to catch each exceptional case as this should be done on the lowest level. They take a bit longer, but you need less of them. You should automate them too.
- High level tests (**system tests**) - a few test that check if the parts of the application are correctly put together. Even if you need just few of them, it is still a good idea to automate them.
- GUI tests - you want just to assure that each element is associated with the correct action. You should consider to automate these test (there are many tools to do this), but testing by humans has some merit here.

Testing automation

- Automating the tests saves a lot of resources in long term projects.
- To benefit fully from the automated testing the tests should be created and improved consistently during the software development process.
  - You want to have confidence in your automated tests. That is, if the test passes, there is a good chance that you did not break something while performing changes.
  - Good tests should give you confidence to fight against software entropy - refactor when necessary.
  - If you do not do this, you might end with very fragile code "If it ain't broke, don't fix it". This approach guarantees that the software entropy takes over and the source may become unmaintainable over time.

Perhaps the most important concern is to guarantee that your tests change as little as possible when you do refactoring.

- This requires not only good design of tests but also good design.

Class under test - class we want to test - what about its collaborations.

- Solitary - We strictly test separate classes. All other classes should be excluded from the testing.
- Sociable - We can use some classes closely related to the class under test (especially when the tests are still fast and there are no significant side effects present).

Even if we prefer sociable unit tests, we want to have the control on what to separate and what not.

Unit tests - dependencies between classes

How the following dependencies affect our ability to perform unit test (of course, in languages like Python we can do almost anything, but this may introduce gaps into the testing)?

- creates, destroys
- calls method
- modifies
- ...

All these dependencies present a huge problem. But dependencies are important, aren't they?

Dependency injection and dependency inversion

- A class should not create instances of other classes (if we do sociable testing it might make some exceptions of this rule for closely related classes) - dependency injection / dependency injection of a factory.
- A class should not depend on the implementation of the collaborator, just on the interface - dependency inversion

Dependency injection is a way to implement **dependency inversion**.

What about the collaborators during the tests? They are behind an interface, thus we can create a new implementation of these classes - **test doubles**.

- If DI is used a lot, the instance creation may become very complicated.
- However, testing is very often the main reason to do DI (other reasons is to gain flexibility, e.g. strategy pattern).
- Thus very often it is possible to indicate defaults for the dependencies injected.
  - There are subtle problems with this approach - this creates dependencies between the sources and affect compilation time negatively
- Other approach is to use DI (micro)frameworks - tools supporting DI.

What if we do functional programming?

- DI is a notion of OO design - OO design is applicable for functional programming.
- If a function f calls function g which we want to separate from f, we do not use it directly. We should have g injected - that is f takes a function with the same signature as g as a parameter (default may be set to g). This allows us to replace g with a test double.

Sociable versus solitary unit tests

I prefer sociable approach

- Advantage of solitary test:
  - they may be faster
  - failure precisely indicates what failed
- but:
  - may require a lot of detailed mocking
  - may make tests too coupled with implementation.
    This make cause that in the case of refactoring we need to rework the tests. This goes against one of the main reasons to white detailed unit tests. Another reason why this coupling is bad is that it is good to have two separate approaches to implementation and testing. If it is not the case it is possible that we do the same mistake twice.
- Solitary unit tests are an accepted approach - there are ways to mitigate these issues.

Test driven development

- Three rules of TDD (I consider this a more hardcore version of TDD).
  1. You are not allowed to write any production code unless it is to make a failing unit test pass.
  2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
  3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.
- This guarantees that the tests will cover boundary and exceptional cases that were implemented

7. **Implementácia a integrácia** (programovacie konvencie; continious integration; test driven development)

Why to have coding conventions [2]:

- 40%–80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Coding conventions, levels:

- Programming language level
- Organization level
- Project level
- Package level
- Single source level

The lower the level, the greater the importance of consistency.

- Comment conventions
- Indent style conventions
- Line length conventions
- Naming conventions
- Programming practices
- Programming principles
- Programming style conventions

Linters

Automated tools to check if you follow the coding conventions and the coding style (to some degree).

- pylint
- Cpplint

Use them, it is very cheap and efficient way to improve code quality.

- If an exception is required, you can mark it in the code.
- This forces you to think more whether the exception is worth it.

How to deal with erroneous inputs?

- Defensive programming - we expect each input may be incorrect.
- Design by contract - a part of the function definition we have what the inputs should satisfy and what happens in that case (e.g. undefined behavior).

Response to detected failure: Fail fast

- Fail fast and visibly.
- Makes it easier to check and correct errors

Note that fault tolerance is not opposite to fail fast. One can fail visibly and still leave large part or the whole system intact.

Test driven development

TDD workflow:

1. Add a test
2. Run all tests and see if the new test fails
3. Write the code
4. Run tests
5. Refactor code
6. Run tests

If you do TDD, you should add your code in very small increments. These rules require you to make even smaller increments.

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Advantages

- great unit test coverage
- efficiency
- little need for debugging
- encourages you to minimize the code that is hard to test (user interfaces, working with databases, etc.).

Disadvantages:

- code and tests are written simultaneously, they may share blind spots
- risk of greater coupling between tests and implementation

How often should you integrate.

- Frequently - integrating frequently allows to detect disagreement between teams working on various parts faster.
- To be able to obtain customer feedback, you should even be able do deliver/deploy frequently.

To illustrate necessary practices we will focus on one specific approach and quite popular approach: **continuous integration**

One can extend continuous integration further.

- Continuous integration - You integrate your system with each commit.
- Continuous delivery - Current version of system is always ready for deployment (not necessarily with each commit but e.g. daily).
  - More testing needed.
- Continuous deployment - You deploy your product continuously.
  - Requires architecture supporting deployment without affecting the production too much.
  - This is architecturally significant requirement.

- Maintain a Single Source Repository.
  - VCS has everything necessary to build and deploy the project.
  - Minimal branches, stable mainline (Reasonable branches are bug fixes of prior production releases and temporary experiments.)
- Automate the Build
  - Automate each aspect of the build
  - IDE independent
- Make Your Build Self-Testing
  - All levels of testing automated
- Everyone Commits To the Mainline Every Day
- Every Commit Should Build the Mainline on an Integration Machine
- Fix Broken Builds Immediately
  - Mainline is in sound state all the time.
  - If a build breaks very often you just return to previous version immediately