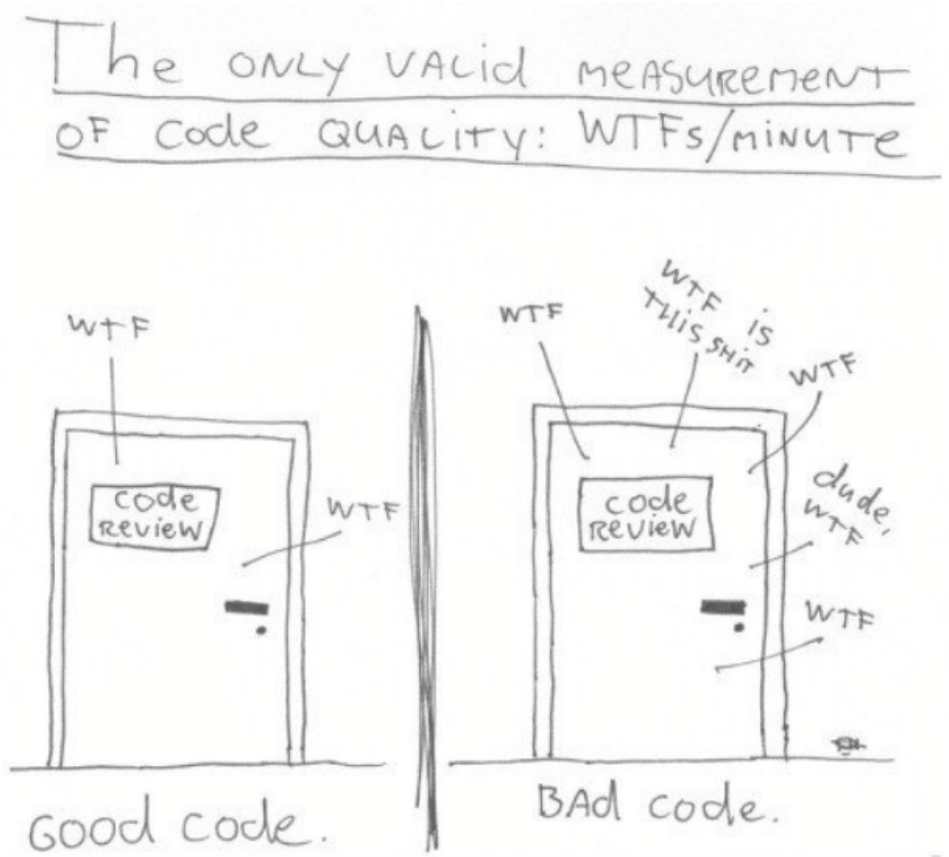
Software-Engineering 2

Prof. Dr.-Ing. Gerhard Wanner
Email: wanner@hft-stuttgart.de

ARCHITECTURAL PRINCIPLES

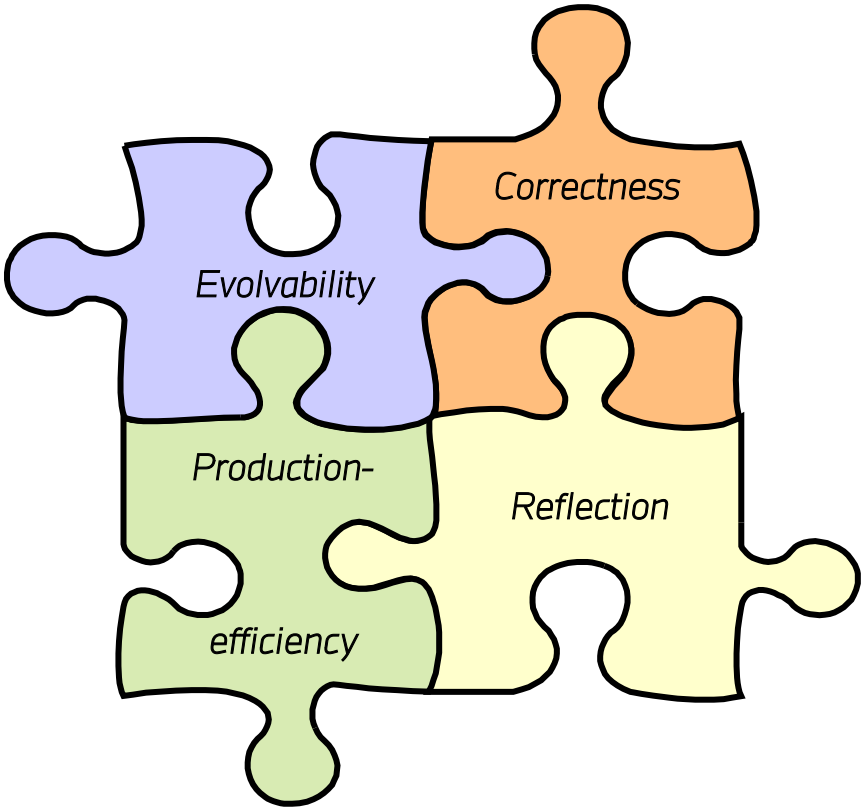


Source: Focus Shift / OSNews / Thom Holwerda, 2008

Agenda

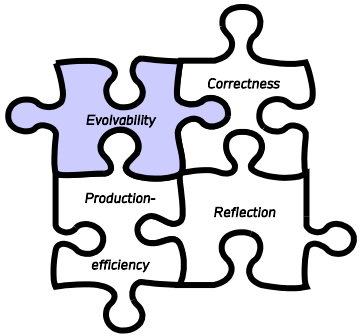
- Value System
- Problematic architectures - examples
- Architectural principles.

Value System - General conditions in software-engineering



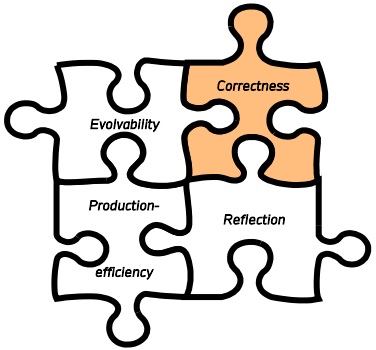
Value System - Evolvability

- Evolvability means, that software must have an internal structure that favors changes of that software
  - Software is usually operated over long periods
  - During this time, frameworks change, features have to be added, etc.
- Ideally, the implementation of a feature costs a fixed amount that is independent of when the feature is implemented
  - In practice, however, the price goes up for a feature, the later it is realized
    - At the beginning, features are reasonably priced
    - In the end it is no longer possible to add features, because no one understands the system → the software will be thrown out and reimplemented
- The simpler the software can be adapted to changed conditions, the higher its evolvability is
  - But evolvability is not obtained retrospectively. It must be considered from the beginning. The software must be designed for it.



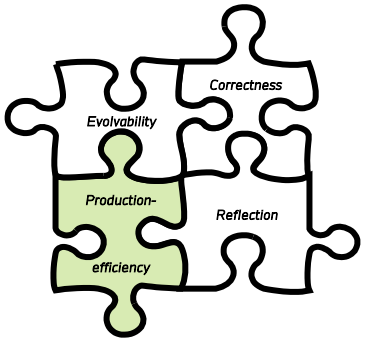
Value System - Correctness

- No one will deny that correctness is required
  - But the question is, how correctness is achieved
  - It's not sufficient that software is given to a test department, whose job it is to find errors
- Correctness must be considered during the development
- Developers have to deal with the issue of correctness!
  - Therefore, it must be clear what the requirements are. Often, they are not defined clearly
  - Developers are asked to implement a feature, without telling them precisely what the acceptance criteria for the feature.



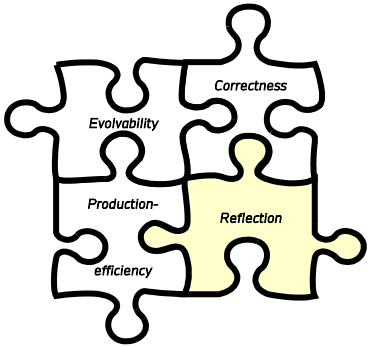
Value System - Production-efficiency

- Development time and the price of the software is an important aspect
  - This price is higher with an inefficient production of the software, e.g.
    - … manual operations, that are not automated
    - … high error-rates that demand repeatedly touch up of the same code-parts
- High production-efficiency means that the software can be developed further for years or even decades
  - Instead of starting again from scratch at a specific point in time
- The production efficiency as a value is also important to set the other values in a moderate ratio
  - To invest a lot of effort for the infinite correctness usually is the wrong approach.



Value System - Reflection

- Without reflection no further development is possible
  - Only those who reflect how a task has been solved may determine whether the chosen route was easy or difficult
  - Learning is based on reflection
- In a recent science like software-engineering it is important to always consider new findings
- This reflection is needed on all levels
  - Starting from pair programming or code reviews
  - Daily reflection of the team
  - Reflection after each iteration
  - Up to the reflection of the entire industry.

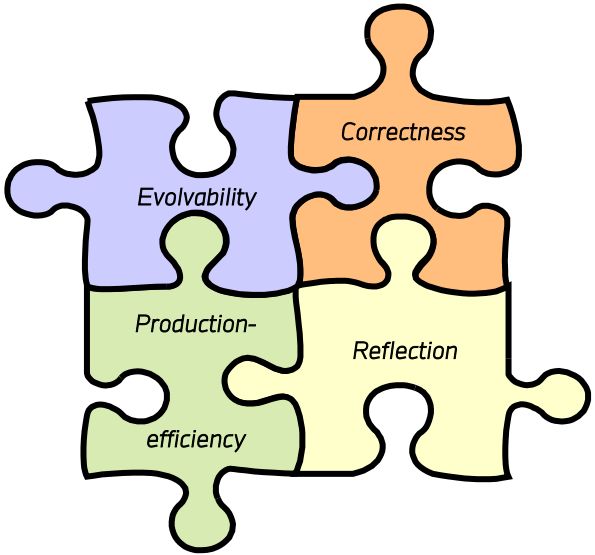


Value-System

- This values-system guides developers in their daily work
  - It does not contain problem solutions but defines general conditions for problem solutions

➔For the daily work those 4 values of the value-system are too abstract

➔Architectural principles help you to achieve the goals of the value-system.



Agenda

- Value System
- **Problematic architectures - examples**
- Architectural principles.

Examples of problematic architectures

- The examples are …
  - real life examples
  - were part of architectures that were created between 1998 and 2010
- They conform to the rule, not the exception
- The architectures were created by experienced programmers in upfront-design
- All products were going into production. They were further developed and maintained for years
- The adequacy of the architecture was judged controversial by the developers.

Examples of problematic architectures (2)

Project context: Overall solution for an energy provider

Special requirements

- Create GUIs using building blocks of domain-objects
- When the GUI of one domain objects changes, all occurrences should change automatically

Solution

- Objects can display theirselves in the GUI in different ways
- Creation of building blocks of more complex objects from simpler building blocks

Side effect

- Very high vertical dependency of the code (GUI → domain class → DB)
- Extremely complex programming model for the developers
- DB optimizations nearly impossible

Punchline

- Flexibility was never used as expected

Examples of problematic architectures (5)

Project context: solution for the management of public affairs

Special requirements

The processes of public affairs can change (e.g. changes of laws)

Solution

Development and integration of a workflow system that includes a spreadsheet based graphical business process language (BPMN was not invented yet)

Side effect

- Very complicated interpreter for the business process language
- Workflow component was one of the expensive parts of the overall system

Punchline

- Component was never used by the users
- Processes of the public affairs were stable

Examples of problematic architectures (6)

Project context: data management and planning for insurances

Special requirements

Unknown requirements for the menu-structure of the application

Solution

Dynamic menus and heavy usage of the command pattern

Side effect

- Complex code, difficult to understand
- Design not optimal (changes at different places necessary) → leads sometimes to inconsistencies

Punchline

At the end the application only had about 40 menu entries

Examples of problematic architectures (7)

Project context: Overall solution for an energy provider

Special requirements

- Object warehouse caches objects in memory for faster access
- Multiple servers for scaling used
- Necessity to synchronize object warehouses between multiple servers

Solution

- Design of a complex synchronization mechanism using a distributed network protocol

Side effect

- Very complex code, very complex network communication
- Application slow down

Punchline

- Caching was introduced in development environment where an MS Access database was used
- Cache was not necessary in production environment (Oracle caches itself)

Agenda

- Value System
- Problematic architectures - examples
- Architectural principles.

"Eselsbrücke" (Mnemonic)

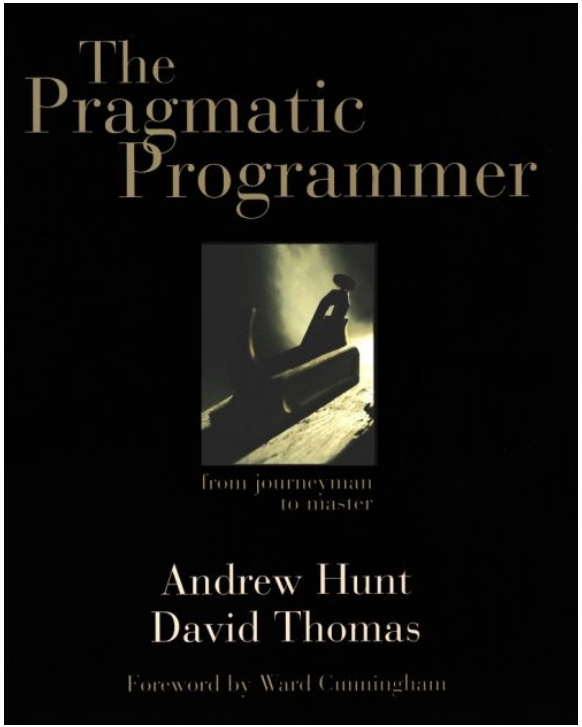
- A memory aid for dummies!
- Like patterns, architectural principles help creating a good architecture
- Patterns describe solutions for recurring problems
- → Architecture principles are behavioral rules in form of mnemonics to omit problems.

Architectural principles

- Don't repeat yourself (DRY)
- You aren't gonna need it (YAGNI)
- Keep it Short and Simple (KISS)
- Principle Of Least Astonishment
- The Pareto Principle (a.k.a 80/20 rule)
- Convention over configuration
- Cyclic dependencies (they are evil!)
- Favour Composition over Inheritance (FCOI)
- Boy Scout Rule
- Dependency Injection
- Global Util Helper – a bad idea….
- Single Responsibility Principle (SRP)
- Cargo Cult
- Conway's Law
- Law of Triviality
- Postel's Law.

Don't repeat yourself (DRY)

- Don't Repeat Yourself (DRY) or Duplication is Evil (DIE) is a principle of software development aimed at reducing repetition of information of all kinds
  - The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
  - The principle has been formulated by Andy Hunt and Dave Thomas in their book "The Pragmatic Programmer" (1999)
    - They apply it quite broadly to include "database schemas, test plans, the build system, even documentation."



Don't repeat yourself (DRY) (Cont.)

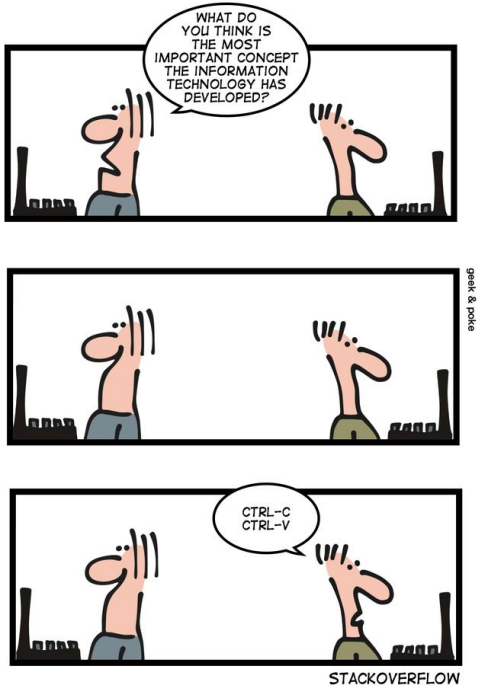
- Consequences
  - When the DRY principle is applied successfully, a modification of any single element of a system does not change other logically-unrelated elements
  - Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync
- Applying DRY
  - Besides using methods and subroutines in their code, Thomas and Hunt rely on code generators, automatic build systems, and scripting languages to observe the DRY principle across layers

    → Single source of truth
    → Implement and maintain functionality only once

  - Copy and paste is NO design pattern.

Don't repeat yourself (DRY) (Cont.)

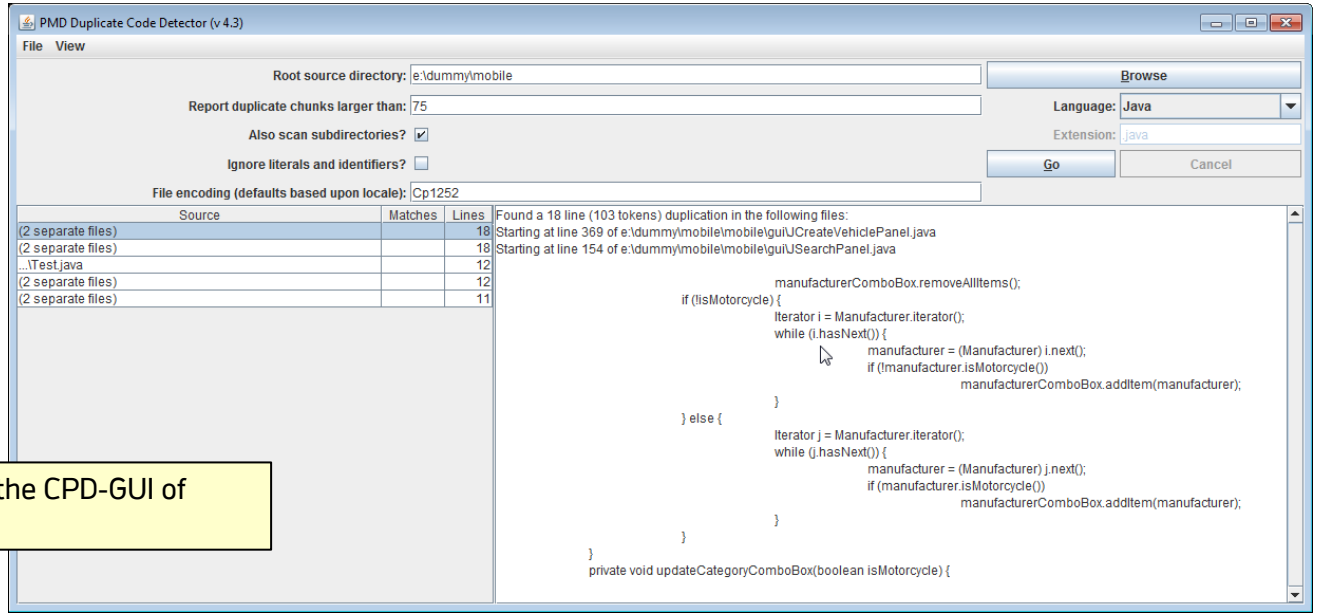
- Examples
  - Also known as Single Source of Truth, this philosophy is prevalent in model-driven architectures, in which software artifacts are derived from a central object model expressed in a form such as UML
    - DRY code is created by data transformation and code generators, which allows the software developer to avoid copy and paste operations
    - DRY code usually makes large software systems easier to maintain, as long as the data transformations are easy to create and maintain
  - Tools such as XDoclet and XSLT are examples of DRY coding techniques
    - Examples of systems that require duplicate information are EJBs v. 2, which requires duplication not just in Java code but also in configuration files
    - Examples of systems that attempt to reduce duplicate information include the Django web framework, Ruby on Rails and EJBs v. 3.



Source: Geek and Poke

Don't repeat yourself (DRY) (Cont.)

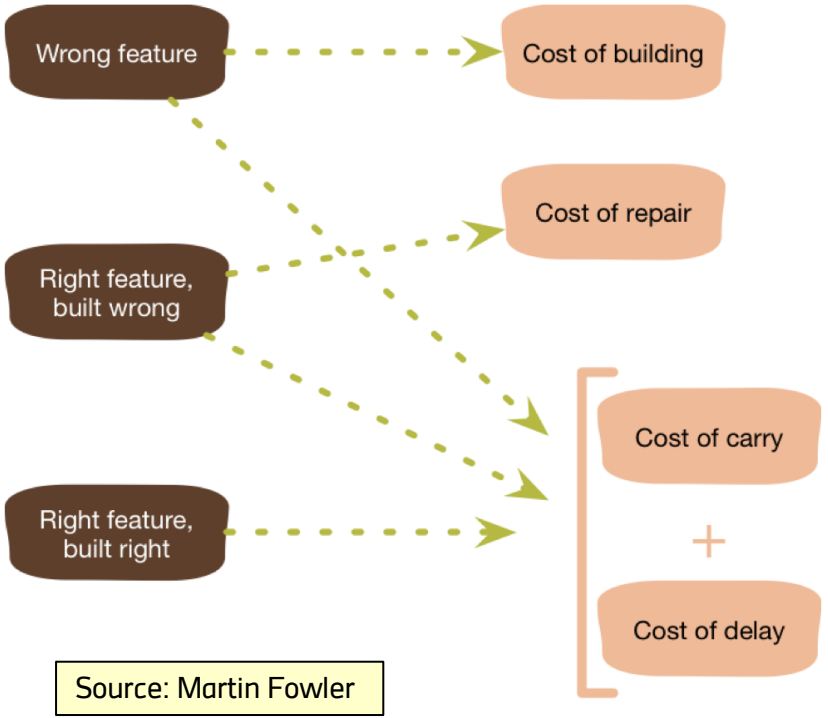
- Tool-support → PMD
  - PMD includes the component "Copy/Paste Detector" (CPD) that finds duplicate code.



Screenshot of the CPD-GUI of PMD

YAGNI (You ain't gonna need it)

- Functionality that is implemented in advance…
  - costs time
  - is not tested enough
  - limits the possibilities in the future
  - makes the code complicated
  - falls into oblivion
  - can lead to additional functionality that is implemented but not needed
  - will probably never be used.



Source: Martin Fowler

YAGNI (You ain't gonna need it) (Cont.)

Always implement things when you actually need them, never when you just foresee that you need them.

Source: XP co-founder Ron Jeffries

Even if you are totally sure that you will need a feature later on, don't implement it now. Usually, it will turn out either you don't need it after all, or what you actually need is quite different from what you foresaw needing earlier. There are two main reasons to practice yagni:

- You save time, because you avoid writing code that you turn out not to need.
- Your code is better, because you avoid polluting it with 'guesses' that turn out to be more or less wrong but stick around anyway.

Source: Cunningham & Cunningham

If you've ever worked on an agile team you've surely heard something like "Let's YAGNI that"
You've probably also heard the phrase "do the simplest thing that could possibly work"
Both of these sayings are an exhortation to developers to avoid overdesign and analysis / paralysis by focusing on the here and now instead of getting caught up in designing for a conjectural need.

Source: Jeremy Miller

Keep it Short and Simple (KISS)

- Perfect doesn't mean that you cannot add something
- It means that you cannot remove something

- Addresses the common problem among software engineers today, that they tend to over complicate problems.

The acronym was coined by Clarence Johnson, lead engineer at the Lockheed Martin Skunk Works

The principle is best exemplified by the story of Johnson handing a team of design engineers a handful of tools, with the challenge that the jet aircraft they were designing must be repairable by an average mechanic in the field under combat conditions with only these tools.

Source: Wikipedia

KISS (Cont.)

- Benefits when using the KISS principle
  - Ability to solve more problems, faster
  - Ability to produce code to solve complex problems in fewer lines of code
  - Ability to produce higher quality code
  - Ability to build larger systems, easier to maintain
  - The code base will be more flexible, easier to extend, modify or refactor when new requirements arrive
  - Ability to work in large development groups and large projects since all the code is stupid simple.

Principle of least astonishment

- The Principle of Least Astonishment (POLA/PLA) applies to user interface design, software design, and ergonomics
  - It is also known as the rule or law of least astonishment, or the rule or principle of least surprise (POLS)
- Example
  - Not optimal…

```
int multiply(int a, int b) {
    return a + b;
}
```

  - Better!

```
int multiply(int a, int b) {
    return a * b;
}
```

Principle of least astonishment

- All of the major Smalltalk language bugs are violations of the principle of least astonishment
  - Smalltalk spends a lot of time building up expectations in users that the system will behave in a certain way
  - The major language bugs are where it violates its own expectations
- Examples:

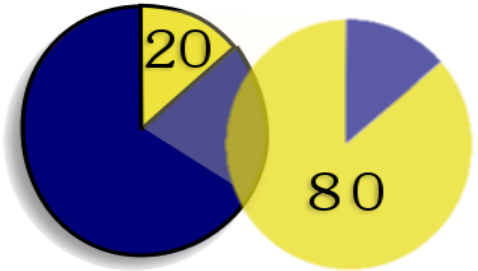
Collections `#add:` and `#at:put:` messages return the operand. The expectation is that a non-querying operation returns `self`

If you add instance variables to a subclass of Collection, these variables will periodically be clobbered because `#grow` only copies predefined instance variables instead of intelligently copying all of them. This violates the expectation that behaviour is local; you shouldn't have to override `#grow` just because you add an instance variable

There should be a method to recursively copy an object and all of its subparts, and this method should be called `deepCopy`. The actual `deepCopy` returns a copy of the receiver with shallow copies of each instance variable.

80/20 rule - Pareto principle

- The Pareto principle (also known as the 80-20 rule, the law of the vital few, and the principle of factor sparsity) states that, for many events, roughly 80% of the effects come from 20% of the causes
  - In IT that means that …
    - 80% of the requirements can be realized using 20% of the effort
    - The remaining 20% of the requirements are 80% of the effort.



Microsoft has learned that 80 percent of the errors and crashes in Windows and Office are caused by 20 percent of the entire pool of bugs detected

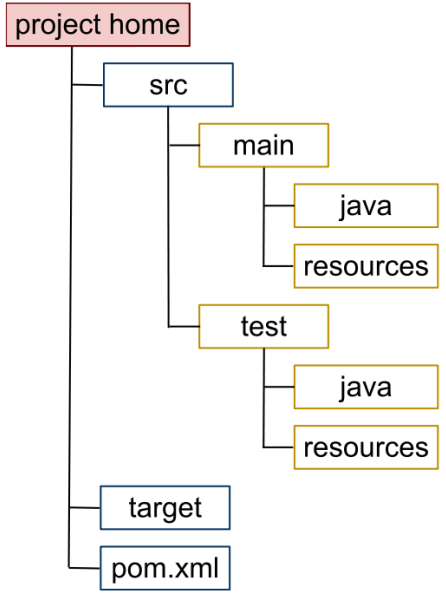
Source: crn.com, Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features

Convention over configuration

- Convention over configuration (also known as coding by convention) is a software design paradigm which seeks to decrease the number of decisions that developers need to make, gaining simplicity, but not necessarily losing flexibility
  - The phrase essentially means a developer only needs to specify unconventional aspects of the application
    - For example, if there's a class `Sale` in the model, the corresponding table in the database is called `sales` by default
    - It is only if one deviates from this convention, such as calling the table "`products_sold`", that one needs to write code regarding these names
- Problems of configurations
  - Some frameworks need multiple configuration files, each with many settings
    - These provide information specific to each project, ranging from URLs to mappings between classes and database tables
  - A large number of configuration files with lots of parameters is often an indicator of an unnecessarily complex application design.

Convention over configuration (Cont.)

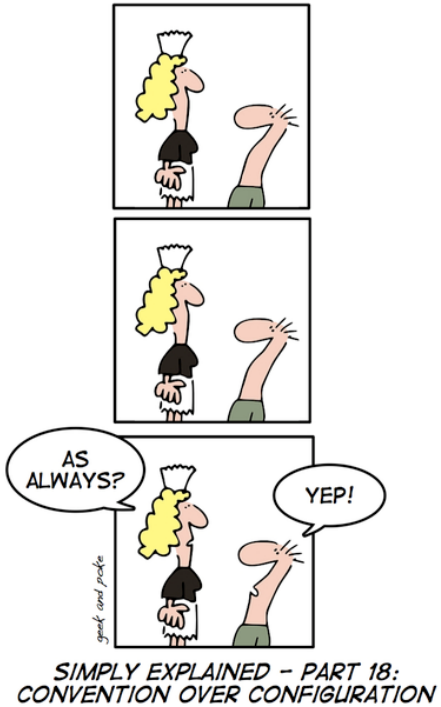
- Many modern frameworks use a convention over configuration approach
  - A few such frameworks include: Spring, Ruby on Rails, Kohana PHP, Grails, Grok, Zend Framework, Pylons, CakePHP, symfony, Maven, ASP.NET MVC, ColdFusion on Wheels, Web2py (MVC), and Apache Wicket.



Wikipedia: Auto-generated directory structure for a Java project using Maven

Convention over configuration (Cont.)

- The concept is older, however, and can be spotted even in the roots of Java libraries
- For example, the JavaBean specification relies on it heavily (JavaBeans specification 1.01):
  - "As a general rule we don't want to invent an enormous **`java.beans.everything`** class that people have to inherit from. Instead, we'd like the JavaBeans runtimes to provide default behaviour for 'normal' objects, but to allow objects to override a given piece of default behaviour by inheriting from some specific **`java.beans.something`** interface."



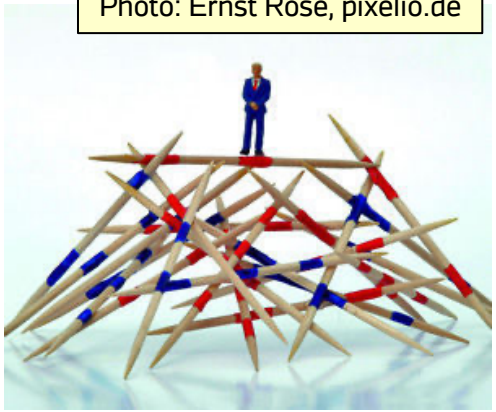
SIMPLY EXPLAINED – PART 18: CONVENTION OVER CONFIGURATION

Source: Geek and Poke

Cyclic dependencies are evil

- Problems of circular dependencies
  - Circular dependencies can cause many unwanted effects in software programs. Most problematic from a software design point of view is the tight coupling of the mutually dependent modules which reduces or makes impossible the separate re-use of a single module
  - Circular dependencies can cause a domino effect when a small local change in one module spreads into other modules and has unwanted global effects (program errors, compile errors)
  - Circular dependencies can result in infinite recursions or other unexpected failures
  - Programs with circular dependencies are difficult to test
  - Circular dependencies may also cause memory leaks by preventing certain very primitive automatic garbage collectors (those that use reference counting) from deallocating unused objects.

Photo: Ernst Rose, pixelio.de



Cyclic dependencies are evil (Cont.)

- Causes and solutions
  - In very large software designs, software engineers may lose the context and inadvertently introduce circular dependencies
    → There are tools to analyze software and find unwanted circular dependencies (e.g. Sonargraph)
  - Circular dependencies are often introduced by inexperienced programmers who need to implement some kind of callback functionality
    → Experienced programmers avoid such unnecessary circular dependencies by applying design patterns like the observer pattern
- Modifications in an upper-level layer should never lead to errors in deeper layers.

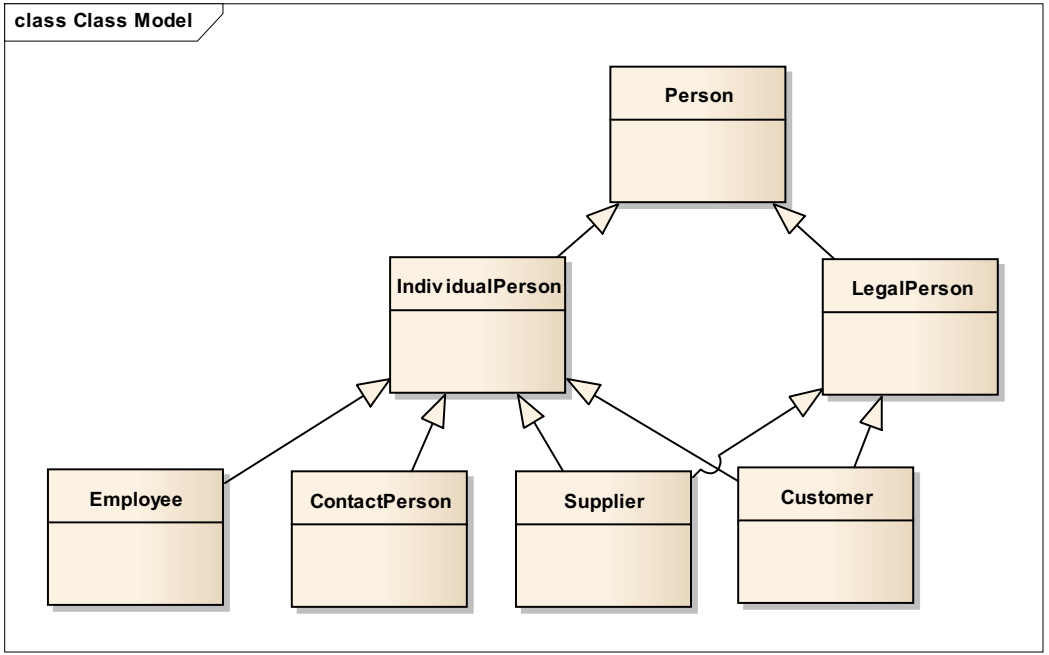
Favour Composition over Inheritance

- Strategy Pattern instead of Template Method
- Composition is dynamic, inheritance static.

> One advantage of composition is that it is more flexible because behavior can be swapped at runtime
>
> One disadvantage of composition is that the behavior of the system may be harder to understand just by looking at the source
>
> Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation"
>
> Gang of Four, 1995

Favour Composition over Inheritance (Cont.)

- Example



- What's the problem?

Favour Composition over Inheritance (Cont.)

- Solution

Boy Scout Rule

- Leave every code you read in better shape than you found it
  - You shouldn't wait for the code to get really ugly
  - Every time you have to read and understand some code, ask yourself: Can I make it easier? Can I make it more readable? Can I make it better?
    - Take at least a small step, every time you answer is yes. Good is better than good-enough
    - Make small, uncritical changes that increase the readability immediately
    - This helps turn the tide, change the (bio)degradation into positive evolution
    - No one writes good code right from the start. Good code evolves from normal code, by using the Boy Scout Rule
  - If you cannot find any way to make the code better, at least eliminate blocks of code with methods. Stick to "7 lines per method", and "Single Responsibility of a method" rules.

Boy Scout Rule - Ideas on how to be a Boy Scout

- Adding to a long function
  - Extract something; there must be plenty of opportunities to pull out some idea and name it. You want to add 3 lines, take out 5. Net improvement leaves a function 2 lines shorter. But I bet you can easily do better. Fix another complex conditional nearby to start a pattern of improvement
- Adding to a complex conditional
  - Extract the conditional into a helper function. Name it and its parameters well. Write a few tests for it. That's picking up the obvious beer cans
- Copy/Paste/Tweak temptation
  - You see a copy/paste/tweak opportunity that meets the functional requirements. Don't do it, except maybe to test your hypothesis. Before, or after the change, do what the code is telling you: extract the common code into a helper function. Generalize and parameterize it so it handles both cases. Write tests around the code to be extracted to guard against breaking existing functionality.

Boy Scout Rule - Ideas on how to be a Boy Scout (cont.)

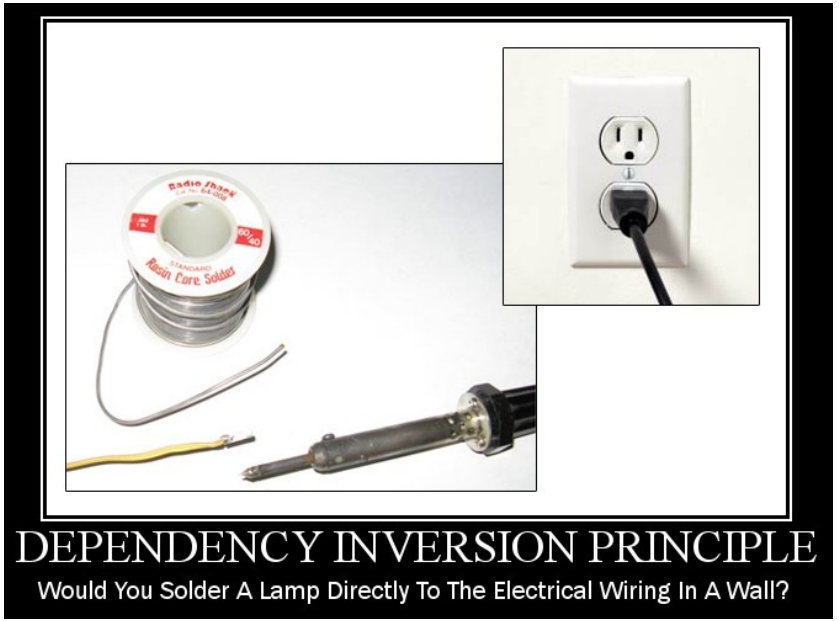
- Cryptic Local Variable Name
  - Once you figure out what the variable is for, rename it to help you, and your teammates, on the next code visit
- Deep Nesting
  - Pull out a nesting level or two into a helper function
- Broken Glass and Beer Cans in the Fire Pit
  - Put on some gloves. Clean out the big pieces of glass and cans. Bag the trash. A heavy-duty bag is best. Be careful not to put any embers in the bag. Take the trash back to the boat.

http://blog.solidcraft.eu/2010/05/boy-scout-rule-in-practice.html

Source (partly): James Grenning's Blog, Boy Scout Rule Applied to Every Day Coding

Dependency Injection

- Target: Loose coupling
- Other names:
  - Inversion of control
  - Hollywood principle
- Configuration and wiring with other components is taken out of the component.



Source: Los Techies, Derick Bailey, SOLID Development Principles – In Motivational Pictures, 2009

Dependency Injection (Cont.)

- Examples
  - ORM → covered in Middleware Technology
  - Spring → covered in Middleware Technology
  - OSGi (→ covered in Software Engineering 2)
  - CDI (→ covered in Middleware Technology).

Global Util Helper – a bad idea….

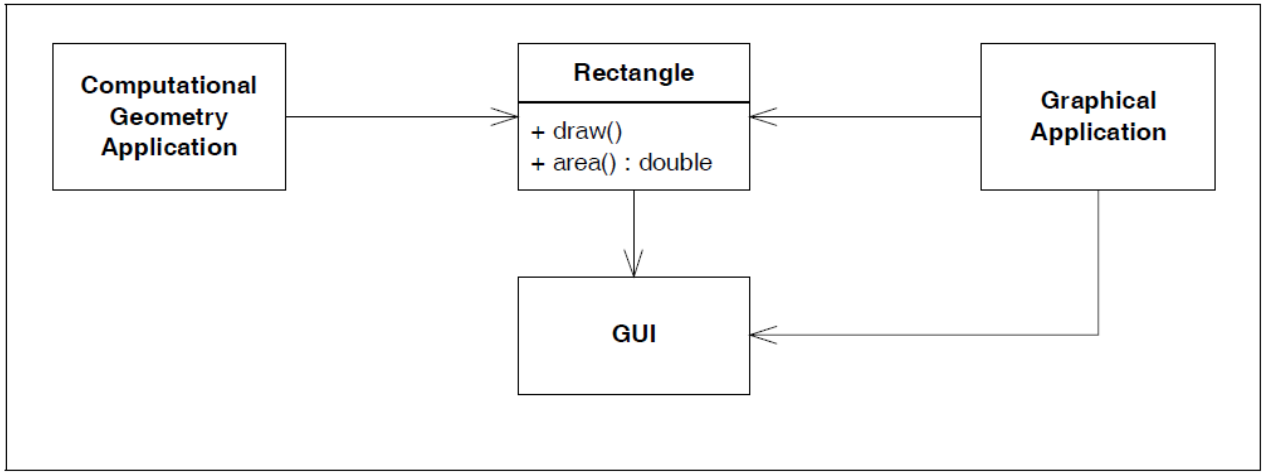
Source: Los Techies, Derick Bailey, SOLID Development Principles – In Motivational Pictures, 2009

Single Responsibility Principle (SRP)

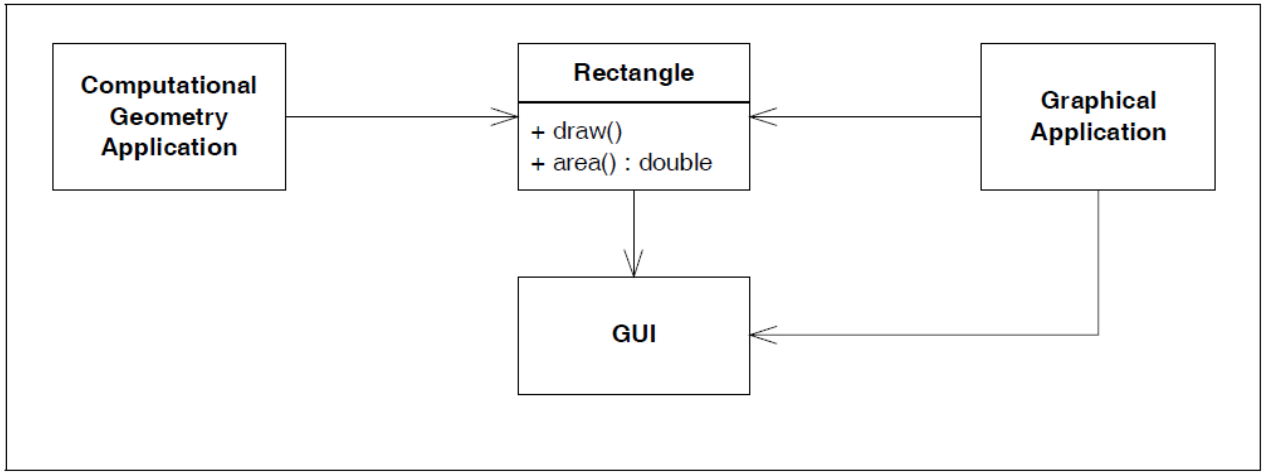
- This principle was described in the work of Tom DeMarco and Meilir Page-Jones. They called it cohesion.

> **There should never be more than one reason for a class to change**

SRP - Example


Computational Geometry Application
Rectangle
+ draw()
+ area() : double
Graphical Application
GUI

SRP - Example: Problem



Rectangle::draw() → Draws the rectangle on the screen
Rectangle::area() → Computes the area of the rectangle.

SRP - Example: Solution

- Separated responsibilities.

Open Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

- When a single change to a program results in a cascade of changes to dependent modules, the program becomes fragile, rigid, unpredictable and unreusable

- The open closed principle addresses this problem
  - It says that you should design modules that never change
  - When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

Open Closed Principle (Cont.)

- Modules that conform to the open-closed principle have two primary attributes
- Open for extension
  - This means that the behavior of the module can be extended
  - That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications
- Closed for modification
  - The source code of such a module is inviolate
  - No one is allowed to make source code changes to it.

Source: Los Techies, Derick Bailey, SOLID Development Principles – In Motivational Pictures, 2009

Cargo Cult

- In the South Seas there is a cargo cult of people
  - During the war they saw airplanes with lots of good materials, and they want the same thing to happen now
  - So they've arranged to make things like runways, to put fires along the sides of the runways, to make a wooden hut for a man to sit in, with two wooden pieces on his head to headphones and bars of bamboo sticking out like antennas - he's the controller - and they wait for the airplanes to land

  - They're doing everything right. The form is perfect. It looks exactly the way it looked before. But it doesn't work. No airplanes land.

Cargo Cult Programming

- Cargo cult programming is a style of computer programming characterized by the ritual inclusion of code or program structures that serve no real purpose
  - Cargo cult programming is typically symptomatic of a programmer not understanding either a bug he or she was attempting to solve or the apparent solution (compare shotgun debugging, deep magic)
    - The term cargo cult programmer may apply when an unskilled or novice computer programmer (or one inexperienced with the problem at hand) copies some program code from one place and pastes it into another place, with little or no understanding of how the code works, or whether it is required in its new position
  - Another sign of Cargo cult programming is the usage of old patterns or programming structures that are not needed any more
    - Example: Usage of old J2EE patterns in modern Java EE applications
- Cargo cult software engineering: same phenomenon for the application of Software Development Processes.

Conway's Law

> **Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations**
>
> M. Conway

- Although sometimes construed as humorous, Conway's law was intended as a valid sociological observation
  - It is based on the reasoning that in order for two separate software modules to interface correctly, the designers and implementers of each module must communicate with each other
  - Therefore, the interface structure of a software system will reflect the social structure of the organization(s) that produced it.

Conway's Law (Cont.)

- Eric S Raymond, restated Conway's law in The New Hacker's Dictionary:
  - The organization of the software and the organization of the software team will be congruent
  - He originally stated that "If you have four groups working on a compiler, you'll get a 4-pass compiler

- How to deal with Conways Law?
  - To get the right communication structure for a product that has to be developed Conway proposes the following "clean slate approach"
    - Define the corporate mission
    - Understand the business processes
    - Adapt the business processes so that they match the corporate mission
    - Structure the it-organization so that it matches the adapted business processes.

Law of Triviality

- Parkinson's law of triviality, also known as bikeshedding or the bicycle-shed example, is C. Northcote Parkinson's 1957 argument that organizations give disproportionate weight to trivial issues
  - Parkinson demonstrated this by contrasting the triviality of the cost of building a bike shed to an atomic reactor:
    - Parkinson shows how you can go in to the board of directors and get approval for building a multi-million or even billion dollar atomic power plant, but if you want to build a bike shed you will be tangled up in endless discussions
    - Parkinson explains that this is because an atomic plant is so vast, so expensive and so complicated that people cannot grasp it, and rather than try, they fall back on the assumption that somebody else checked all the details before it got this far
    - A bike shed on the other hand can be build by anyone. So no matter how reasonable you are with your proposal, somebody will seize the chance to show that he is doing his job, that he is paying attention, that he is *here*
- The law has been applied to software development and other activities.

Law of Triviality (Cont.)

- Related principles and formulations
  - In the context of programming language design, one encounters Wadler's law, named for computer scientist Philip Wadler
    - This principle asserts that the bulk of discussion on programming language design centers around syntax (which, for purposes of the argument is considered a solved problem), as opposed to semantics
  - The duck technique in corporate programming is an applied example of Parkinson's law of triviality:
    - A feature added for no other reason than to draw management attention and be removed, thus avoiding unnecessary changes in other aspects of the product

- Often you find that also in code reviews that are all about "that method doesn't have a doc comment", "why are you using var", "why do you set the brackets in that way" or "alphabetize your includes".

Postel's Law

> **Be conservative in what you send.**
> **Be liberal in what you accept.**
>
> **J. Postel**

- In 1981, Jon Postel formulated the Robustness Principle as a basic implementation guideline for the then-new TCP
  - The intent of the Robustness Principle was to maximize interoperability between network service implementations, particularly in the face of ambiguous or incomplete specifications
  - It was quickly accepted as a good proposition for implementing network protocols in general
  - Later it was applied to the design of APIs (e.g. accept calls to the old version of an interface) and even programming language design
- → Better interoperability.

Literature

- Clean Code: A Handbook of Agile Software Craftsmanship
  - Robert C. Martin
  - Prentice Hall, 2008
  - ISBN-13: 978-0132350884
- Clean Code Developer
  - https://clean-code-developer.com/
- How To Write Unmaintainable Code - Ensure a job for life ;-)
  - Roedy Green
  - https://github.com/Droogans/unmaintainable-code