# Engineering Practices for Building Quality Software

## Week 1

**What is Quality Software?**

quality:
+ The standard of something as measured against other things of a similar kind; the degree of excellence of something
+ Of good quality; excellent

### Quality throughout the engineering process

Design

Architecture & Security

Implementation

Testing and Deployment

### Design

Defining good design

Coupling, Cohesion, and more

Quality metrics

Software design patterns

### Architecture & Security

Security

Usability

Performance

Availability

### Implementation

Code style

Debugging

Commenting

Build process

### Testing and Deployment

Test selection

Test adequacy

Continuous integration

Continuous deployment

### Engineering practices for building quality software

Striving for quality throughout the lifecycle

Variety of approach across 6 content areas

Practical measures of quality

Identifying problems before they start

## What is good design?

### What is software?

## Software quality attributes

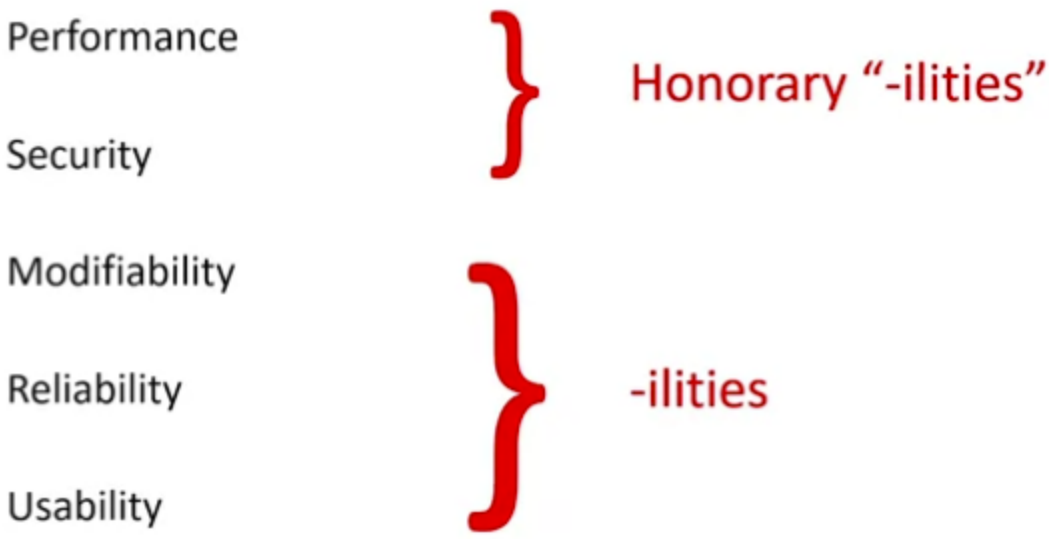
Performance

Security

Modifiability

Reliability

Usability




## Software quality criteria

Coupling & cohesion

Liskov’s substitution principle and the other SOLID design principles

Law of Demeter

## Coupling and cohesion

Mantra: Low coupling, high cohesion

Coupling: the level of dependency between two entities

Cohesion: how well an entity’s components relate to one another

## Liskov’s substitution principle

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness)

## SOLID

S - Single responsibility principle

O - Open/closed principle

L - Liskov substitution principle

I - Interface segregation principle

D - Dependency inversion principle

## Law of Demeter

Principle of least knowledge

Formally: a method m of an object O may only invoke the methods of the following kinds of objects:

- O itself
- m’s parameter
- Any objects created / instantiated within m
- O’s direct component objects
- A global variable, accessible by O, in the scope of m

## Good design is considering trade-offs

Consider the effects of your work in a variety of perspectives

Quality attributes help answer: When is it good enough?

Quality criteria help identify locations which need extra case

Improving one often degrades another: choose wisely

# Quality Attributes according to Microsoft

Chapter 16: Quality Attributes

For more details of the topics covered in this guide, see Contents of the Guide. Quality attributes are the overall factors that affect run-time behavior, system design, and user experience. They represent areas of concern that have the potential for application wide impact across layers and tiers.

https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658094(v=pandp.10)

# CMU SEI Technical Report on Quality Attributes

Quality Attributes

December 1995 * Technical Report This report describes efforts to develop a unifying approach for reasoning about multiple software quality attributes. Computer systems are used in many critical applications where a failure can have serious consequences (loss of lives or property).

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=12433

# Section: Quality metrics

# Measuring coupling

Coupling: degree to which entities on each other

## Instability

A metric to help identify problematic relationship between entities

Measures the potential for change to cascade throughout a system

Instability, despite the connotation, is not necessarily good or bad

Instability is determined by two forms of coupling: afferent and efferent

For module X:

- Afferent coupling (AC) counts other modules that depends on X
- Efferent coupling (EC) counts other modules that X depends on

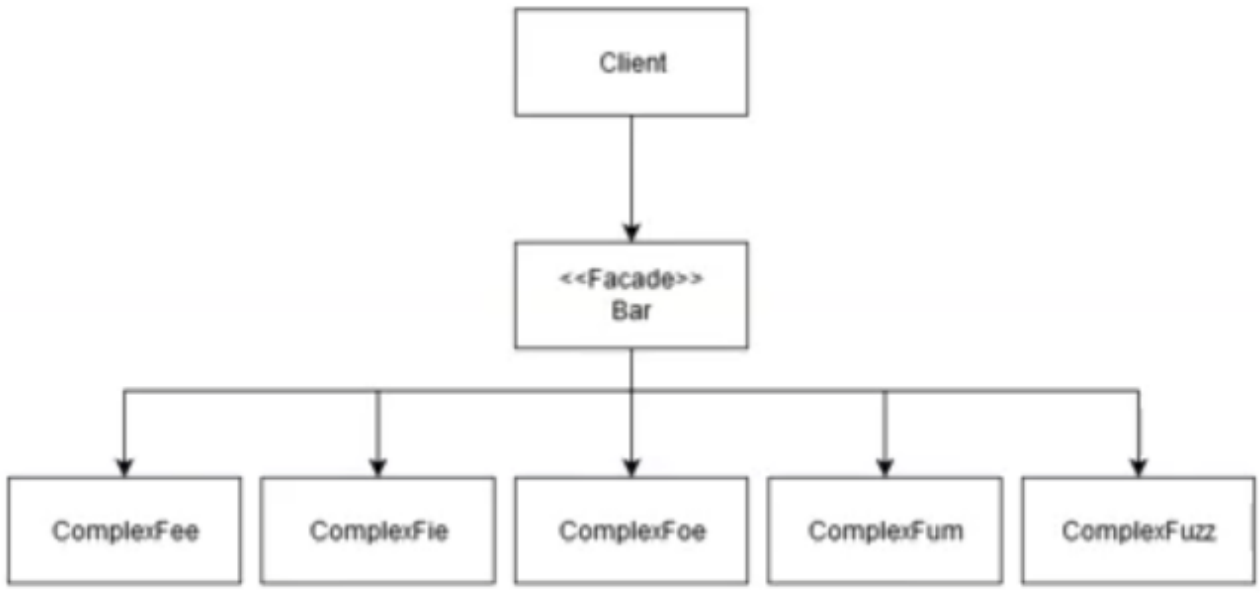
Instability is the proportion of EC in the total sum of coupling

- $I = \frac{EC}{AC + EC}$

# Instability Example (1/3)

## High Instability

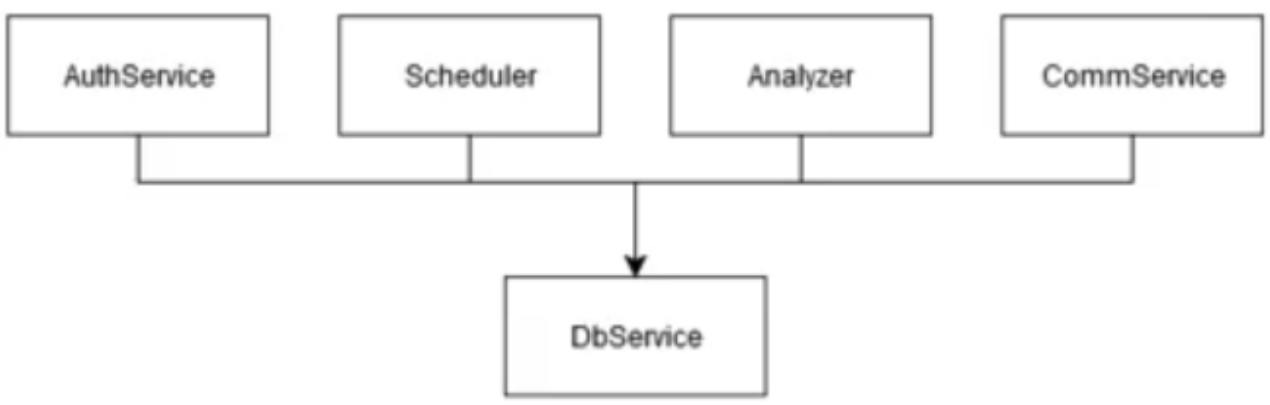
$$\frac{5}{1+5} = \frac{5}{6}$$




# Instability Example (2/3)

## Low Instability (High abstractness)

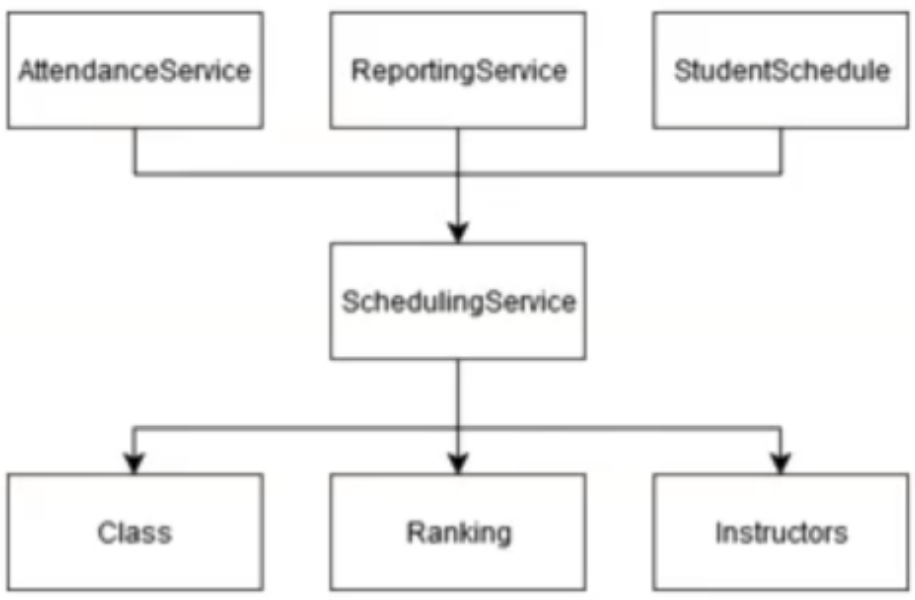
$$\frac{0}{4+0}$$




# Instability Example (3/3)

## Similar amounts of AC and EC

$$\frac{3}{3+3} = \frac{3}{6}$$



Coupling can be measure quantitatively

- High coupling can lead to a domino effect when change occurs
- Measuring coupling can help identify problem classes and design
- Instability is not necessarily good or bad
- Perfect instability or abstractness is neither possible nor advisable

## Coupling factors

Coupling Factor ( )

Next: Data Abstraction Coupling ( Up: Coupling Previous: Efferent Coupling ( ) Works with all instances of a common meta-model, regardless if they where produced with the Java or the UML front-end. The respective call, create, field access, and type reference relations (Java) or association, message and

http://www.arisa.se/compendium/node109.html#metric:CF

$$\in \text{package}^{C}$$

## Measuring cohesion

Cohesion: Measure of the interdependence of data and responsibilities within a class

### Lack of Cohesion of Methods (LCOM)

A metric to help identify problem classes

Several variants exist (LCOM1, LCOM2, LCOM96a, etc.)

LCOM4 (Hitz & Montazeri) is relatively simple yet useful

### LCOM4 measurement

This is simple measure of the number of connected components

Having one connected components of one is considered high cohesion

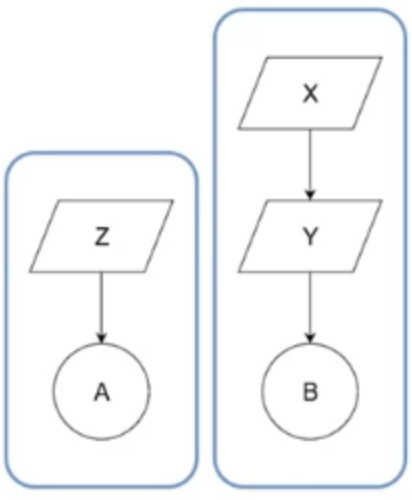
Methods are “connected” if

a) they both access the same class-level variable, or

b) one method call the other

LCOM4 Example

Consider the above example. There are two sets of connected components, and LCOM4 = 2.



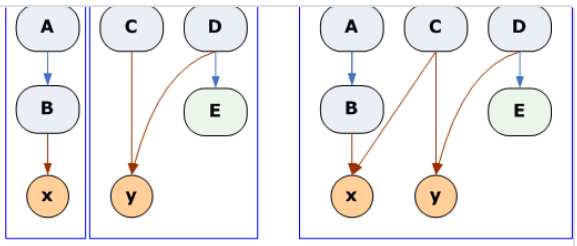

Cohesion can be measure quantitatively

- Low cohesion can indicate poor quality and design
- LCOM4 is a simple way of measuring the cohesion of classes
- LCOM4 is not the only measurement approach
- Cohesion is only one measure, to be considered in context

Cohesion metrics

Cohesion metrics measure how well the methods of a class are related to each other. A cohesive class performs one function. A non-cohesive class performs two or more unrelated functions. A non-cohesive class may need to be restructured into two or more smaller classes.

https://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM4




# Additional Measures of Quality

Defect density

Cyclomatic complexity

Cognitive complexity

Maintainability rating

Coupling factor

Lack of documentation

## Defect density

One of many post-mortem quality metrics

Defect count found divided by number of lines of code

Quick check of quality between classes

## Cyclomatic complexity

Complexity determined by the numbers of paths in a method

All functions start with 1

Add one for each control flow split (if/while/etc.)

## Cognitive complexity

An extension of Cyclomatic complexity

Attempts to incorporate an intuitive sense of complexity

Favors measuring complex structure over method count

## Maintainability rating

Measure of the technical debt against the development time to date

Example ratings for differing ratios (used by SonarQube)

Outstanding technical debt estimate is:

- A - Less than 5% of time already spent on development
- B - 6% - 10%
- C - 11% - 20%
- D - 21% - 50%
- E - 50% and above

## Coupling factor

The ratio of classes of coupled class and the total umber of possible coupled classes

Two classes are coupled if one has a denpendency on the other

The coupling factor (CF) or package p is defined as:

$$CF(p) = \frac{\sum_{c \in classes(p)} Coupled(p, c)}{0.5 * Classes(p)^2 - Classes(p)}$$

## Lack of documentation

Simplistic ratio of code which is documented and all code

One comment per class and one comment per method is expected

$$LOD = 1 - \frac{\#documented}{\#needing\ documentation}$$

## Many additional metrics exist

Use additional metrics of quality as appropriate

Consider the cost and benefit of developing and tracking metrics

Some can be used during development; others only in retrospect

Historical trends can be used to inform future process and projects

Lack Of Documentation ( )

Next: Code Conventions Up: Documentation Previous: Documentation Works with all instances of a common meta-model, regardless of whether they were produced with the Java or the UML front-end. Handle Description How many comments are lacking in a class, considering one class comment and a

http://www.arisa.se/compendium/node121.html#metric:LOD

$$x^{LOD}(\mathrm{Class})$$

Metric Definitions

JavaScript, PHP Complexity is incremented by one for each: function (i.e non-abstract and non-anonymous constructors, functions, procedures or methods), if, short-circuit (AKA lazy) logical conjunction (&&), short-circuit (AKA lazy) logical disjunction (||), ternary conditional expressions, loop, case clause of a switch statement, throw and catch statement, go to statement (only

https://docs.sonarqube.org/latest/user-guide/metric-definitions/

# Section: Software design pattern

# Introduction to Patterns and the Observer Pattern

## Design patterns

Elements of reusable design

Vocabulary to talk about design

Just another abstraction, among many, to reason about design

Each design patterns identifies 2 things:
+ Problems that exists
+ Solution/process to solve problem

## Observer

Common barrier to better modularity: Need to maintain consistent state

Subject - has changing state

Observe - needs to know about changes to that state

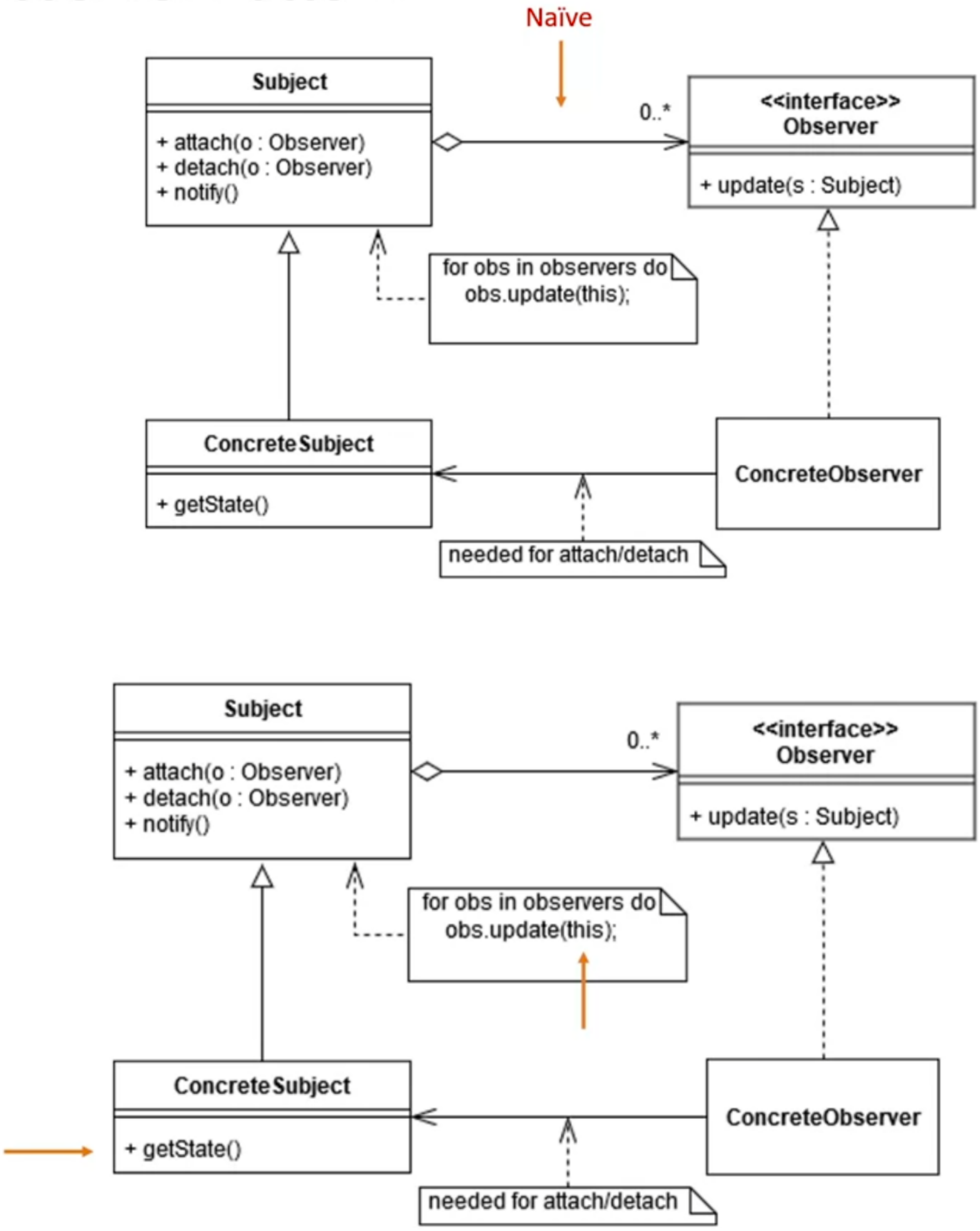
Obvious approach: Subject informs Observers on change

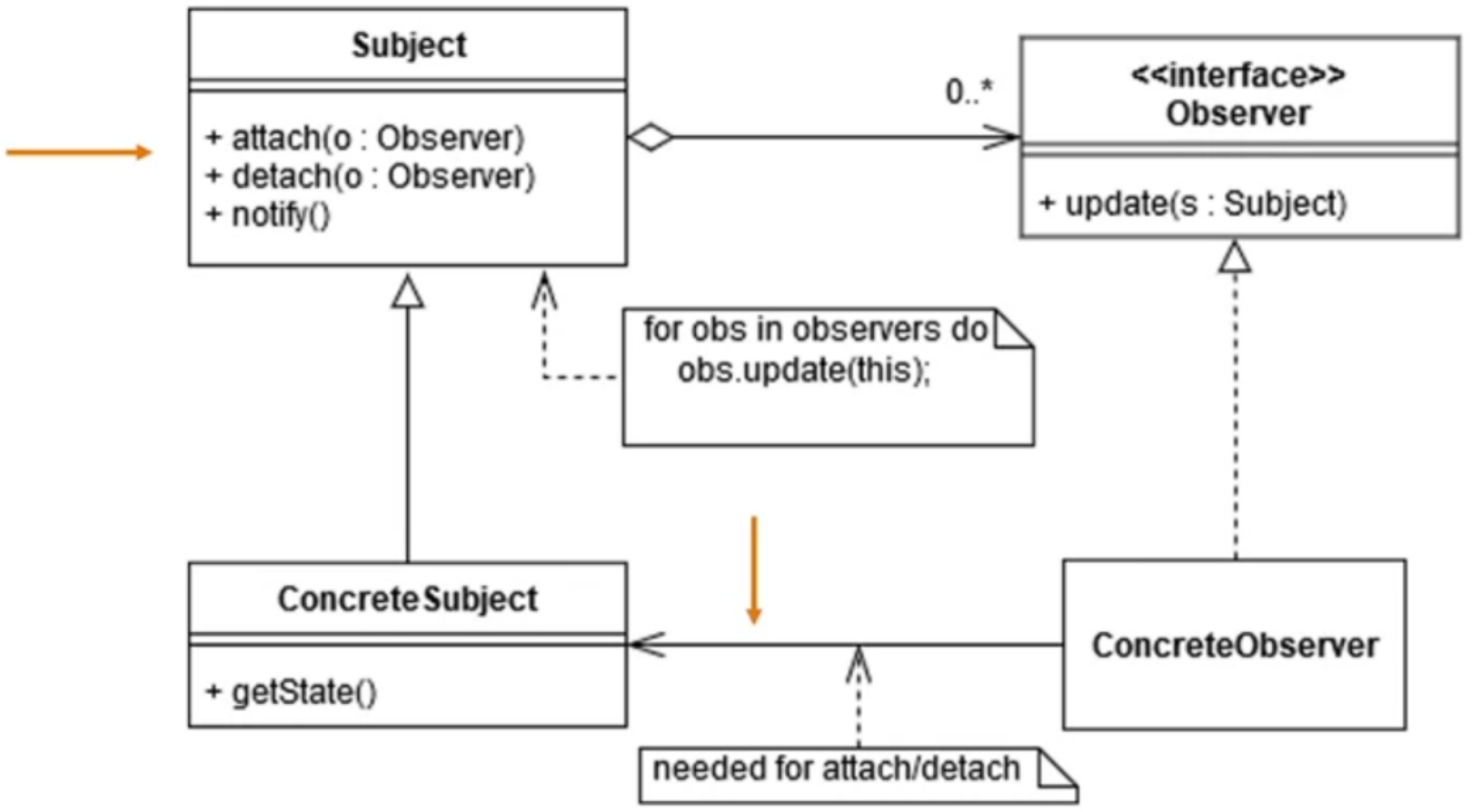
## Problems with the Obvious approach

Conceptually, Subjects had no knowledge of Observers.

Better approach: Create a “naive” dependency

# Observer Pattern






# Observer in the Wild!

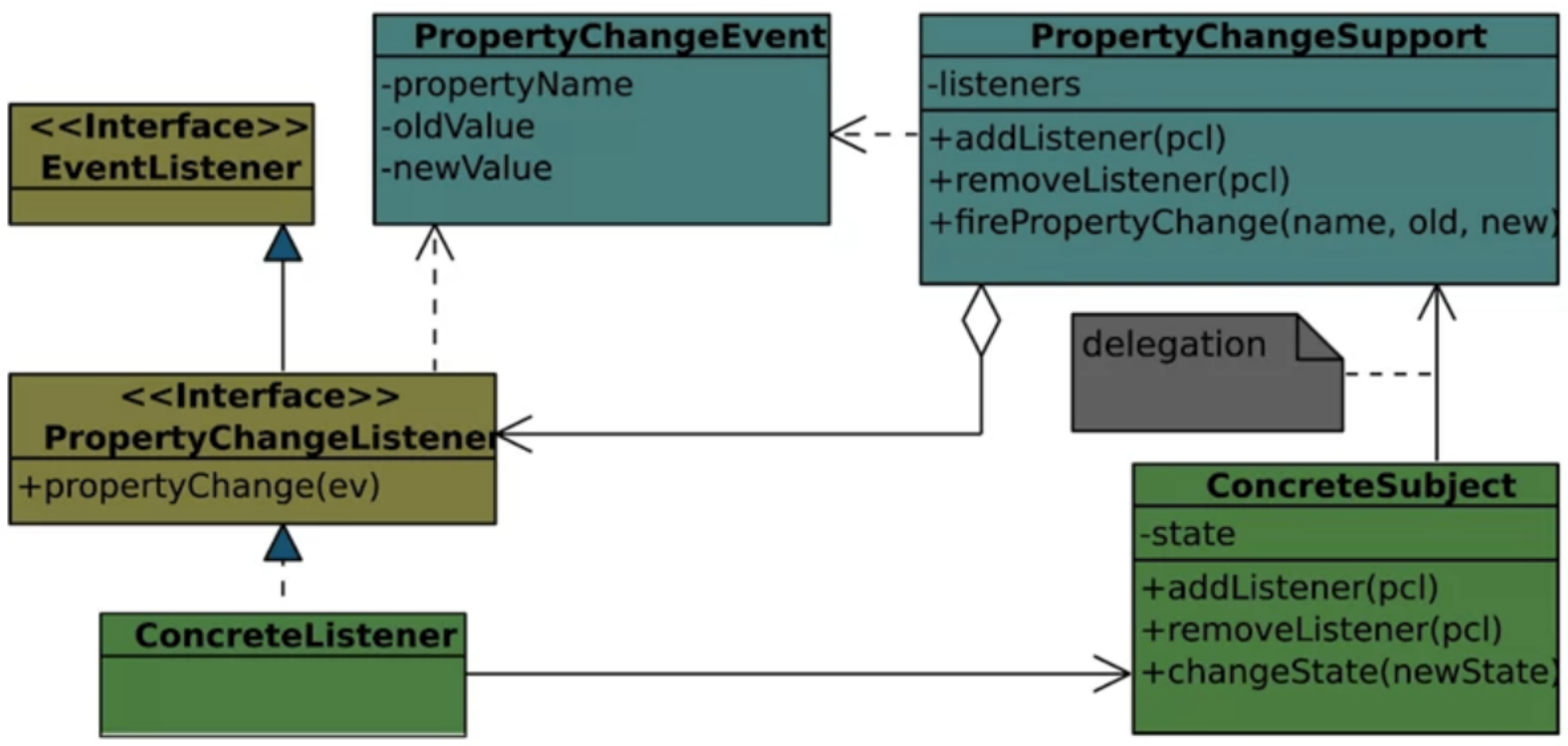



Figure - © John Collins

## Patterns: common solutions to common problem

Each pattern adds a tool to your toolbox when you run into issues.

The Observer patter is a common solution to the pub/sub (publish/subcribe) problem

Patterns are not meant to be taken literally or as set in stone

Vary the solution's suggested shape to meet the situation's needs

Design Patterns and Refactoring

Design Patterns and Refactoring articles and guides. Design Patterns video tutorials for newbies. Simple descriptions and full source code examples in Java, C++, C#, PHP and Delphi.

https://sourcemaking.com/design_patterns/observer/cpp/1

## Strategy Pattern

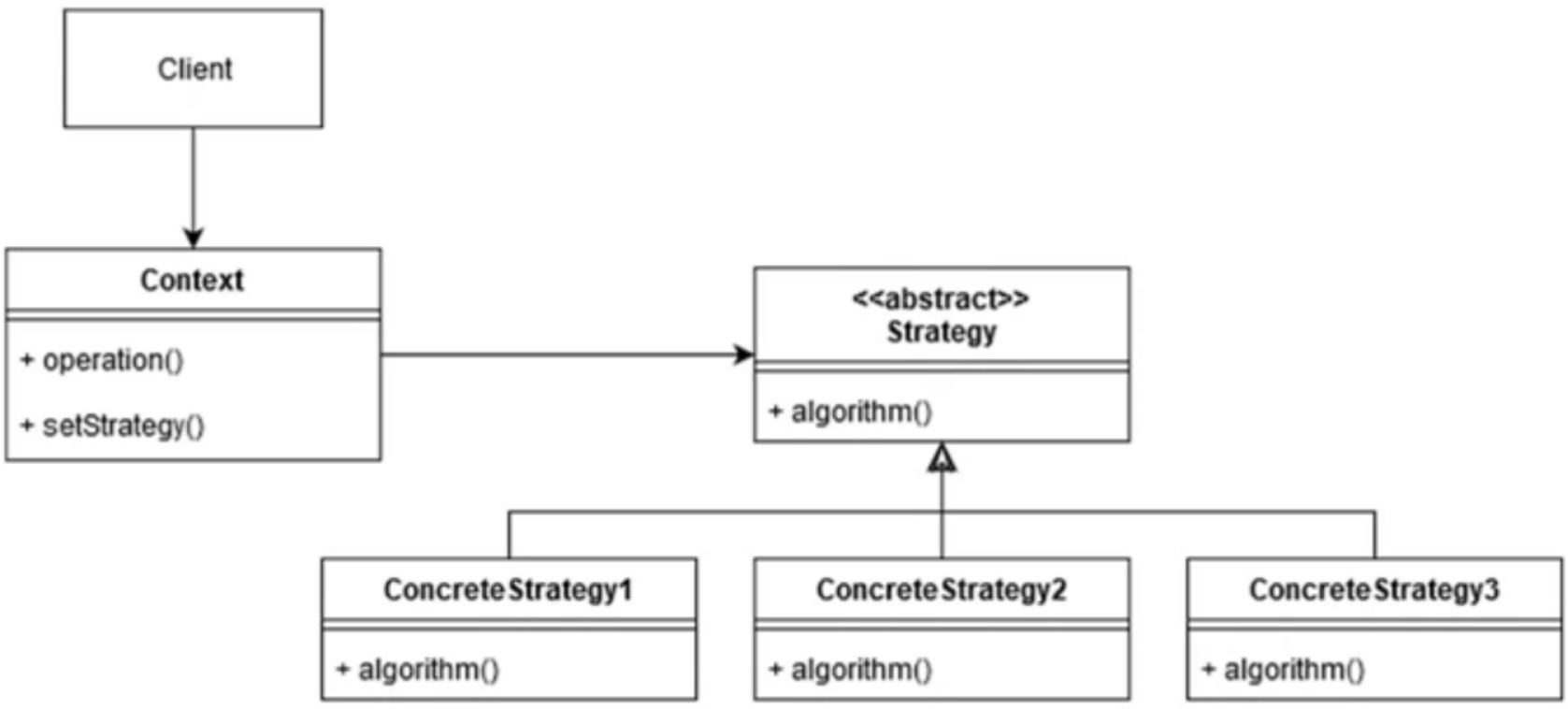
Looks at trying to encapsulate complexity and delegate it to an external hierarchy

## Strategy

Define a family of algorithms encapsulate each one, and make interchangeable

**Definition**

- Client
- Context
- Strategy
- ConcreteStrategy




## Example #1

Logger can be configured for several types of output

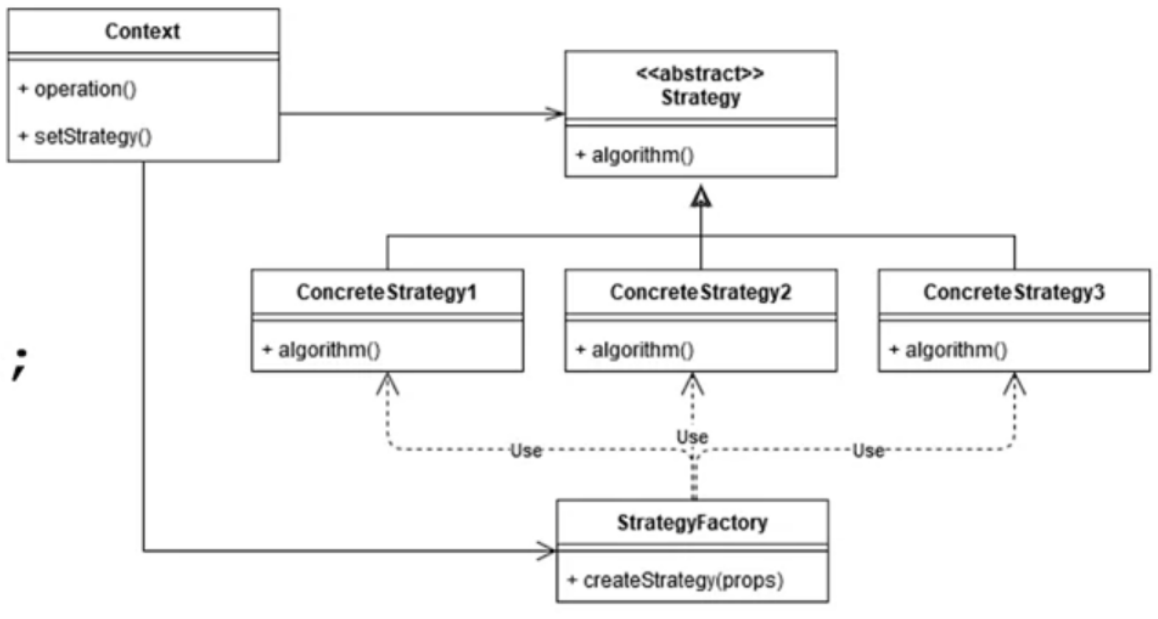
**Selecting the Strategy**

Whose responsibility is it to select the strategy to use?

When should the Context select? When should it not?

Client or Context requests Strategy instances from a StrategyFactory, passing properties uninterpreted

# Example




```
File my_file;
.
.
.
.
Cipher my_cipher =
CipherFactory.getCipher(my_file.length()0;
my_cipher.decrypt(my_file);
```

## Strategy allows expansion in a family of algorithms

Abstract operation out of a client into its own object hierarchy

Strategy seprates operation based on circumstances

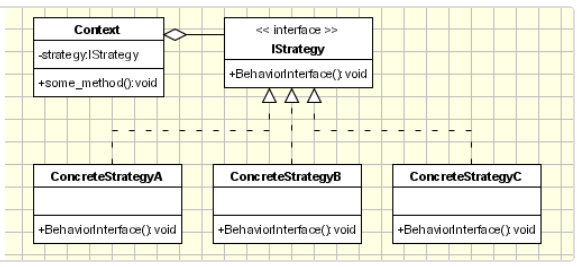
Factory encapsulates complexity of selecting sepcific ConcreteStrategy

Client can be purely ignorant of types and selection criteria

Strategy

There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime. Define a family of algorithms, encapsulate each one, and make them interchangeable.

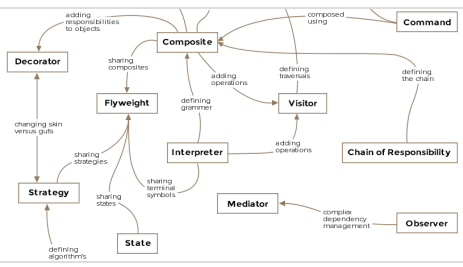
https://www.oodesign.com/strategy-pattern.html




How to use Strategy Design Pattern in Java? Example Tutorial

Hello guys, you might have heard about it, Can you tell me any design pattern which you have used recently in your project, except Singleton? This is one of the popular questions from various Java interviews in recent years.

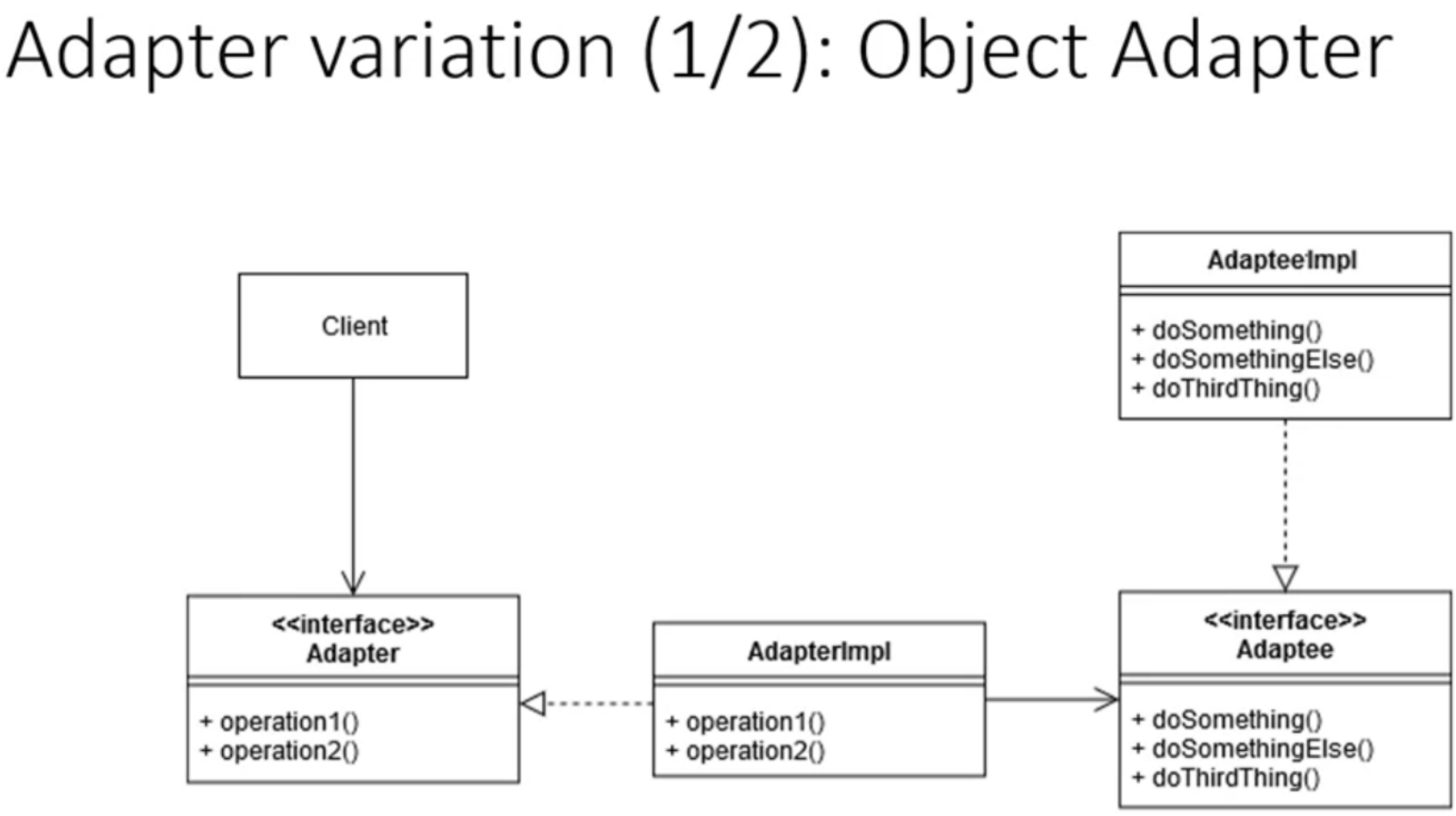
https://www.java67.com/2014/12/strategy-pattern-in-java-with-example.html



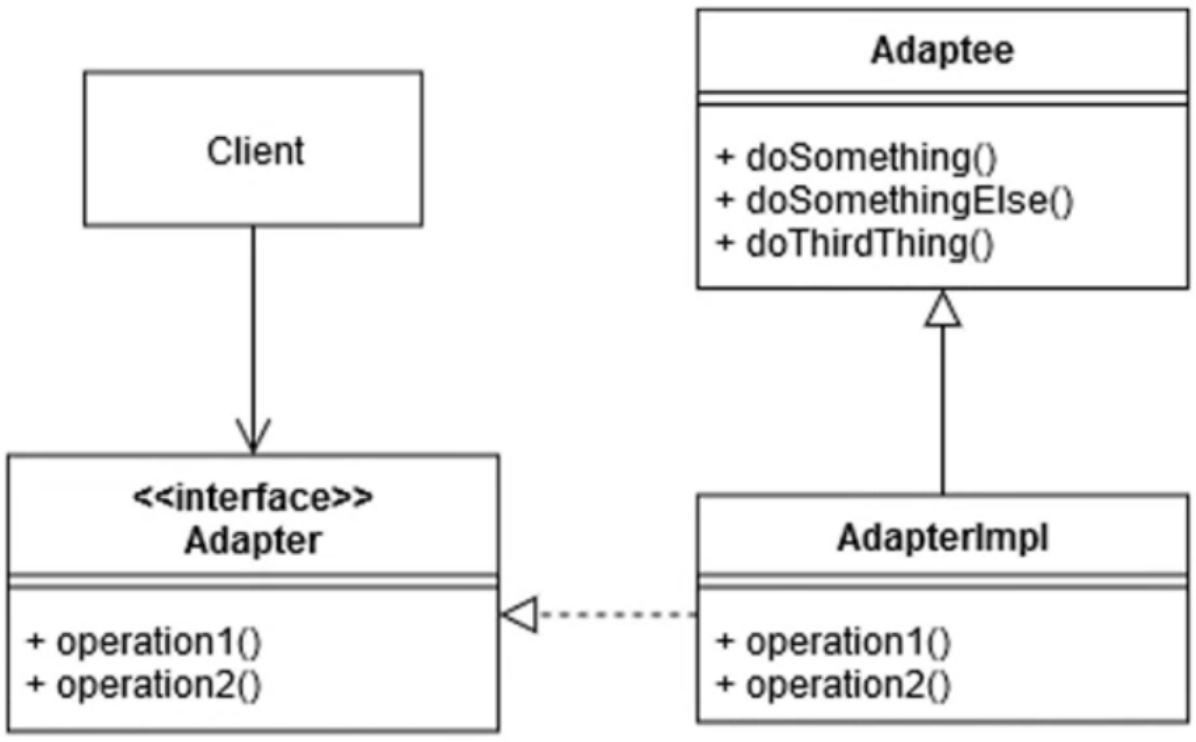

## Adapter pattern

open/closed principle: closed to modification/open to extension

Def: Convert the interface of a class into another interface that clients expect



# Adapter variation (2/2): Class Adapter




Mantra: “Prefer composition over inheritance”
~ “Prefer delegation over inheritance”

### Adapter limits change when replacing classes

Legacy code relies on the same interface as before

Adapter maps the legacy class to methods of new class

Two forms of the adapter: Object Adapter and Class Adapter

One of several forms of refactoring pattern

Adapter Pattern - GeeksforGeeks

This pattern is easy to understand as the real world is full of adapters. For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other.

https://www.geeksforgeeks.org/adapter-pattern/

Adapter

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate. Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

https://refactoring.guru/design-patterns/adapter

# Week 2

## What is Software Architecture?

### What it Architecture?

The architecture of a system is the set of fundamental concepts or properties of the system in its environment, embodied in its elements relationships, and the principles of its design and evolution.

The software architecture of a system is the set of structure needed to reason about the system, which comprise software elements, relations among them, and properties of both.

### Why should we care?

Good things are well architected

Good architecture is hard

Poor architectures collapse quickly

## Architectural qualities

Performance

Ease of maintainance

Security

Testability

Usability

## Software architecture is bigger than software

Views constraint, control, and possibility from an enterprise lens

Good architecture helps project success, bad architecture ensures it fails

Architectural qualities drive much of the work of an architecture

Established architectural styles should be recognizable and repeatble

ISO/IEC/IEEE 42010 - Getting Started

This page provides some starting points for using the Standard (previously IEEE 1471:2000). The best place to start is with the Standard itself. The published version of ISO/IEC/IEEE 42010 can be obtained from ISO or from IEEE. The Standard has several parts. The first part is a conceptual model of architecture descriptions (ADs).

http://www.iso-architecture.org/ieee-1471/getting-started.html

# Architecture style

Building a software is the same as building a house, same process but different methodologies

**Styles:**

- Pipe-and-Filter
- Event-based architectures
- Layered architectures
- Blackboard
- Others

## Pipe-and-Filter

Developing components such that the input and the output are the same

Similar the “pipe” operator in *nix system

Allows for expansive reusability and parallelization

About transformational system, not appropriate for interactive system or statefulsystem (like a word processor / website)

## Event-based architecture

Breaks the common “reactive” model

Design entire system in Observer/Observable pattern

Also called the publish-subcribe model

Very asynchronous and run in parallel

Request-response vs. publish-subscribe, phần 1: Sự khác biệt là gì?

Tất cả chúng ta đều biết rằng máy tính và các thiết bị điện tử khác - máy in, bộ định tuyến, máy tính xách tay, điện thoại thông minh, v.v. - được nối mạng để chúng có thể trao đổi thông tin.

https://iotgateway.vn/request-response-vs-publish-subscribe-phan-1-su-khac-biet-la-gi

## Layered architectures

Only communication is between adjacent layers

Layers form abstraction hierarchy

Dependencies are top-down

Client-Server is an example of a layered architecture

### Architectural styles convey essential aspects

Architectural style constrains design to benefit certain attributes

Ensure that all components within the system work together well

Incorporates ideas at the enterprise level, rather than class design

Established architectural styles should be recognizable and repeateable

> 💡 avoid hammer and nail problem

## View, Viewpoint, and Perspective

### How we think about and document architecture

Focus on essential aspects of the system being built

Focus on separation of concerns

Focus on addressing common issues found in large systems

### Views

An View captures a model meant to address some concern

An architecture is the combination of all Views

Each View is defined by a single concern called Viewpoint

### Viewpoint

View is the vehicle for portraying architecture

Viewpoint is the best practice for portaying some specific concern

A set of viewpoints should address artifacts, development, and execution

6 viewpoints:

- functional
- informational
- concurrency

→ describe the artifacts created before the development

- functional viewpoint: capture the system’s components along with the responsibilities, interfaces and primary interactions between them
- deployment focus on dependencies of the system being deployed: network, data storage needs, runtime environment
- operation: focus on the long-term running of the app

### Perspective

For cross-cutting concerns that don’t fit into a single view

Quality attributes come into play here

Apply to all the viewpoints in the system

### Viewpoint and Perspective guide the world

Addressing each concern by documenting them separately

Views are the final product at this stage and are the architecture

Viewpoints allow system organization systematically

Perspectives influence each viewpoint and the views they represent

# Writing Scenarios

## Architectural quality

A system must exhibit values of quality attributes to be successful

Quality concerns tend to cut across views

Documenting (and testing) these qualities is difficult

## Quality attribute scenarios

Source

Stimulus: some events that occurs that require the response caused by the source.

Environment

Artifact

Response

Response measure

## Scenario example - context

Parts Depot, Inc.
Large retailer exploring web services
Large number of services

`Source` Customer using standard web browser

`Stimulus` User adds an item to their online shopping cart

`Environment` System is operating normally. System has fewer than 50 concurrent users. Internet latency is less than 100ms from customer browser to site

`Artifact` WAGAU Tomcat website

`Response` Round trip time from customer clicking “add to cart” button to customer browser update showing updated cart

`Response Measure`
95% of the time, under 2,5sec
99.9% of the time under 10sec

## Scenarios help use define good systems

Heuristics are used to allow definitive measurement

Documenting quality scenarios is key to validation and acceptance

Identifying the assumptions in the envionment is critical

User sign-off on scenarios gives you definitive goals of quality

# Assignment: Assessing Quality through Scenarios

**Select a public website that you use enough to be familiar with what a typical user may want to do. This website should not require the peer reviewer to sign up for an account or pay to use the site in any way. The website should also not, to the best of your knowledge, serve malware, use JavaScript to complete cryptocurrency mining, or any other negative practice that might harm the peer reviewer.**

**Select three quality attributes that are likely to be important when deciding a website architecture for the website you chose. You can use usability, security, performance, reliability, or any other reasonable quality attribute as the basis of your selection. Briefly explain the importance of each quality attribute as it relates to the software/service you selected.**

**Then, for each selected quality attribute, write one scenario that would help quantitatively assess whether the software solution meets its goal. You shall write your scenarios using a format that was presented in the lectures.**

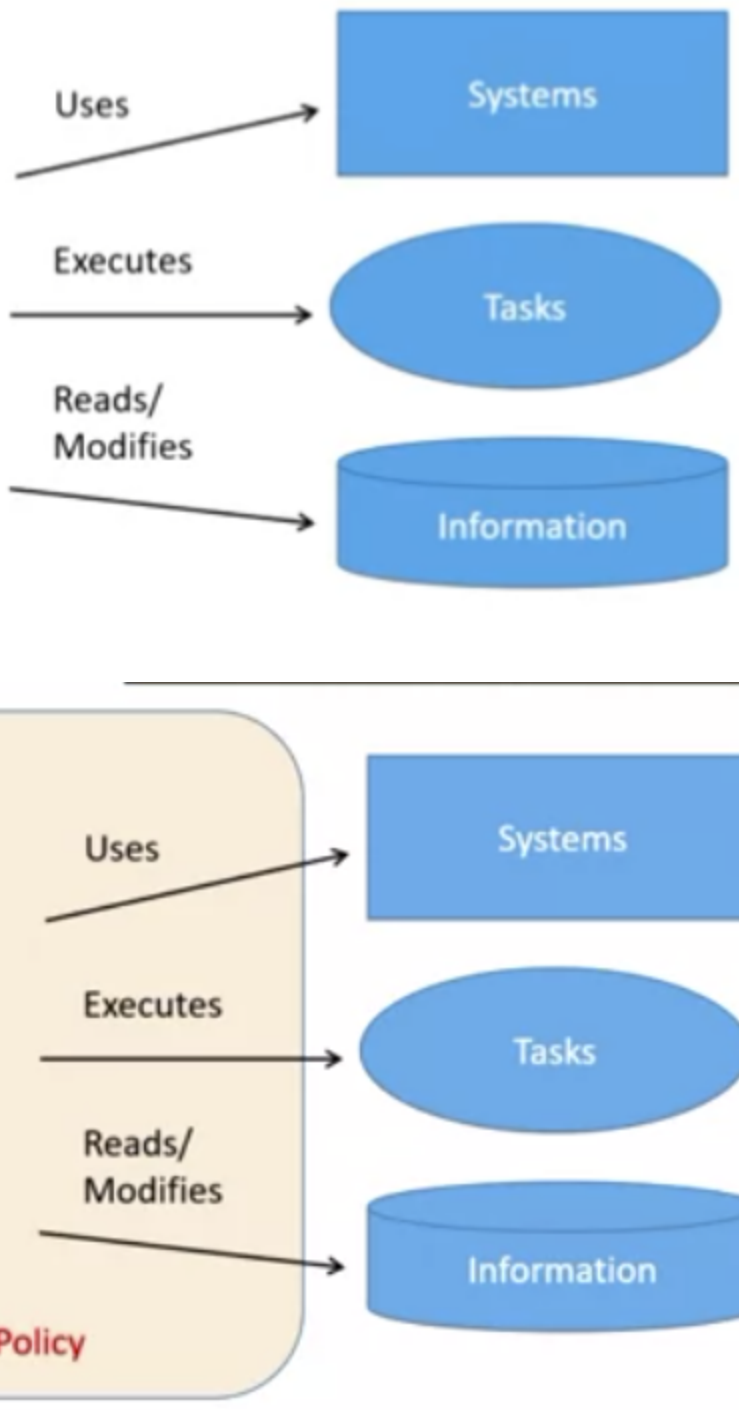
**Write your first selection, discussion of importance and scenario here.**

Write your second selection, discussion of importance and scenario here.

Write your third selection, discussion of importance and scenario here.

## Security Perspective

Security is the set of processes and technologies that allow the owners of resources in the system to reliably control who can perform actions on particular resources.




### Security is Risk management

Policy: identify risks, categozie, and prepare mitigation

Evaluating the cost and benefit of risks and their prevention

Identify the principals, resouces, and sensitive operations

### Access control matrix

| | Principle 1 | Principle 2 | ... | Principle *n* |
|---|---|---|---|---|
| Resource 1 | | | | |
| Resource 2 | | | | |
| Resource 3 | | | | |
| ... | | | | |
| Resource *m* | | | | |

Read, Write, Execute, Update, Audit, Migrate, Resize, Delete, etc.

### Security threats

Ientify sensitive resources

For each resource:

- What are the characteristic of attackers?
- Who could try to infringe security?
- How could they attempt to bypass countermeasures?
- What would be the consequences?

### Security requires persistent evaluation

Resources, attacks, and best practices must evolve over time

Managing risk can only happen with analysis

Analysis includes actor chracteristics

Attack trees can be used to organize security risks

## Attack trees models

Is an organizational tactic to list attack on a resource across threat

### Identify attack vectors

Think like an attacker

Document possible attacks in a tree

Use the tree to guide further security analysis

### Attack tree models

Format to document results of security analysis

Allows for visual categorization to aid understanding

Gaps and selective elaboration

# Example: Opening a safe

Goal: Open Safe

OR 1. Pick Lock
2. Learn Combination
OR 1. Find Written Combination
2. Get Combination From Target
OR 1. Threaten
2. Blackmail
3. Eavesdrop
AND 1. Listen to conversation
2. Get Target To State Combination
4. Bribe
3. Cut Safe Open
4. Install Improperly

Adapted from Bruce Schneier's "Attack Trees"

# Example: Opening a safe

Goal: Open Safe

OR 1. *Pick Lock*
2. Learn Combination
OR 1. *Find Written Combination*
2. Get Combination From Target
OR 1. *Threaten*
2. *Blackmail*
3. Eavesdrop
AND 1. **Listen to conversation**
2. *Get Target To State Combination*
4. **Bribe**
3. **Cut Safe Open**
4. *Install Improperly*

Adapted from Bruce Schneier's "Attack Trees"

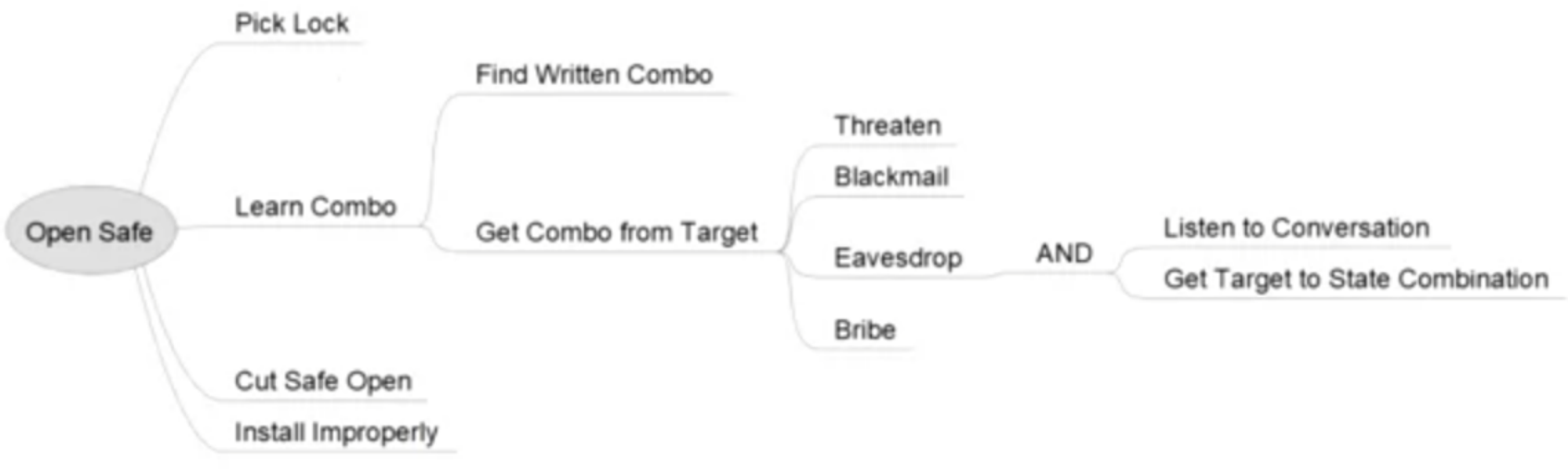
Goal: Open Safe - $10k

# Example: Opening a safe

OR 1. Pick Lock - $30k
2. Learn Combination - $20k
OR 1. Find Written Combination - $75k
2. Get Combination From Target - $20k
OR 1. Threaten - $60k
2. Blackmail - $100k
3. Eavesdrop - $60
AND 1. Listen to conversation - $20k
2. Get Target To State Combination - $40k
4. Bribe - $20k
3. Cut Safe Open - $10k
4. Install Improperly - $100k

Adapted from Bruce Schneier's "Attack Trees"

## Using Mind maps




## Attack trees allow for visualizing threats

Can be used to identify possibility, cost, and other attributes

Seeing the larger pictures is beneficia; in analysis

There is still a great deal of estimation here; use caution

Updating the tree on a regular basis as envionment and attacks change

# Security Tactics

Follow best practice to ease the difficulity of meeting and assesing security

Domain experts have identified and organized defense and analysis

Technical and social attacks and defenses need to be considerd

## Recoginzed security principles

Principle of least privilege

Secure the weakest link

Defind in depth

Separate and compartmentalize

KISS

Avoid obscurity

Use secure defaults

Fail secure

Assume external entities are untrusted

Audit

## Security audit

Most organizations are not mature anough to organize active deterrence

Auditting of access and behavior is essential to discovering breach

Logging of all secure, authorization, and authentication is crucial

## Authenticate and Authorize

### Verify the role and the authority

Ensure that the access control matrix:

A) Is implemented properly by access mechanisms

B) Properly implements intent of security policy

Critical: Ensure reliable identification of each principal (role) during use

## Information Protection

Secrecy (or Privacy) is an essential necessity for data

Transmission of data (necessary for business) exposes it to attack

Encryption can protect information outside authorization control

Security requires persistent evaluation

Resources, attacks, and best practices must evolve over time

Managing risk can only happen with analysis

Analysis includes actor characteristics

Attack trees can be used to organize security risks

## Week 3: Quality in implementation

## Coding style

## Section: Debugging and Static Analysis

## Debugging

The art of removing defects

Two major forms: print statements and debugging tools

Difficulty in visualizing state of program

### Error messages

Error messages are notoriously cryptic in many languages

Even simple mistakes can result in massive walls of error text

Customizing error messages can also help

### Conclusion

Using a debugger grants powerful execution statement views

Debuggers have a sharp learning curve

Print statements are quick but limited

In both cases, visualization is not error-proof

## Static analysis

Variety of ways to check code quality by examining source

Attempt to check beyond syntax error

Common error-prone areas of development can be analyzed

### Types of analysis

Unused variables

Empty catch blocks

Unneccessary object creation

Duplicated code

Incorrect boolean operator

Some typos

Use of null pointers

Division by zero

Overflows

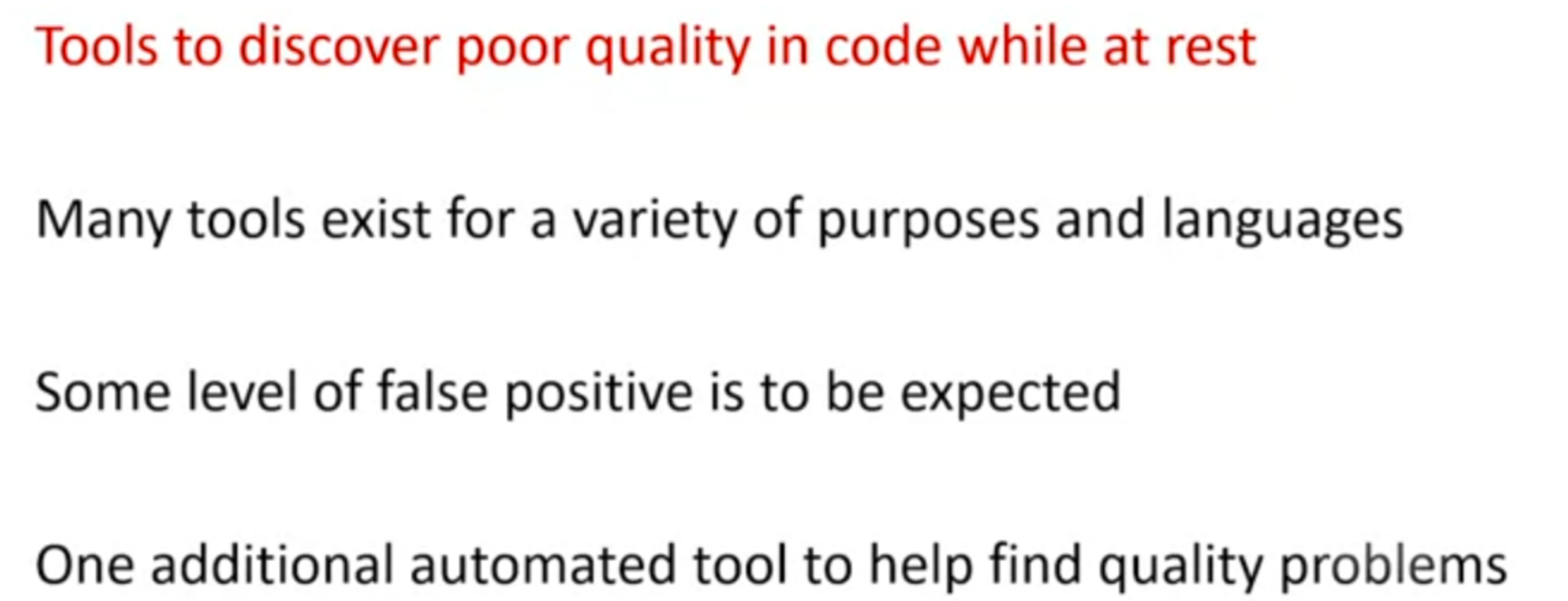
Out of bound checks

Information leak

Use of unsanitized input

Much more...

### Static analysis control




PMD

See also Getting Started Download pmd-bin-6.44.0.zip Extract the zip-archive, e.g. to C:\pmd-bin-6.44.0 Add folder C:\pmd-bin-6.44.0\bin to PATH, either Permanently: Using System Properties dialog > Environment variables > Append to PATH variable Temporarily, at command line: SET PATH=C:\pmd-

https://pmd.github.io/#about

## Comment and Self-documentation

### Documenting in code

Documenting is key to understanding, now and later

Self-documenting code get updated dynamically

Comments provide context that code cannot

Both forms are likely necessary

## Version Control Systems

Version control allows developers to maintain incremental backups

Allows for separation of code for development, testing, production

Even more powerful when used to integrate team contribution

## Commits

Store the newest version of the code-base

Allows inclusion of descriptive notes (”commit message”)

Can be used for traceability

## Branching

Master, Hotfixes,  Release branches, Develop, Feature branches

## GitHub and GitLab

Two primary providers of remote git repository

Generally, very similar in capability

Allows for push/pull and webhook functionality

## Conclusion

Effectively a must on modern software development

Store backups with contextual history

Connect changes to bug tracking

Allows multiple developers to work on the same project at once

# Build Process

Simple approach: using IDE or CL to compile and execute

Simple, effective, no prior preparation needed

Repeatability, consistency, errors are all concerns at the command line

Understanding and modifiability are issues in a IDE

## Automated build

Develop files which control the build process

Invoke tool commands to accomplish common tasks

Some primary options: make, Gradle, Maven, Ant

## make and Ant

Traditional tools with rich history and extensive documentation

Ant primarily used for Java builds and various file system tasks

make - the original-portable and language-independent

### Gradle and Maven

Both are expandable and maintainable build tools

Maven improved on Ant; Gradle incorporates benefits from both

Gradle has simpler, shorter, non-XML configuration files

### Conclusion

Modern build tools provide reproducible tasks on demand

make was the beginning and is still used widely, especially for C/C++

Ant provided the first “modern” build tool

Maven and Gradle are the prevailing build tools for Java

# Week 4: Quality in Testing and Deployment

## Test Selection

Overall goal: Select the minimum number of tests which identify defects better than random testing

### Definitions

Test case

Test data

Test suite: set of test cases

SUT: software under test

Actual output

Expected output

Oracle: determine whether or not the test actually passes
JUnit, PyUnit, PHPUnit

Code coverage: measure how well the tests work by measure the extent to which they cover the code structure (percentage)
+ Benchmark of coverage (e.g.: at least 80% statement coverage)

Manual testing

Automated testing

Test selection: how we select input to run on the SUT

Test adequacy: how well our test has done in finding the bugs

### Who should select test?

Developer

- Understand the system
- Will test it gently
- Driven by deadline

Tester

- Must learn the system
- Will attempt to break it
- Driven by “quality”

### Test selection

Numerous approaches to selecting tests to run

Manual Testing
Exception Testing
Boundary Testing
Randomized Testing
Coverage Testing

Requirements Testing
Specification Testing
Safety Testing
Security Testing
Performance Testing

### Randomized testing

Using randomized input value to identify defects in isolated code

Works quite well in finding defects

Automation of test generation allows many random tests to be run

### Code coverage

Using coverage metrics to identify code not covered

Helps discover use cases and execution paths missed in test planning

Generate tests cases to meet coverage criteria

**Higher level**

- Condition coverage
- Decision coverage
- Modified condition/decision coverage

### Requirements testing

End-to-End tests which highlight common and important users actions

Written without knowledge of code structure

Helps determine when you are “done”

### Much more info

Just a few examples of test selection approaches

Many of those listed have more in-depth lectures

## Test adequacy

Determine whether a test suite is adequate to ensure the correctness or desired level of dependability that we want for our software

... but this is very difficult

... in fact, for correctness, it is generally impossible

### Approximating adequacy

Instead of measuring adequacy directly, we measure how well we have covered some aspect of the
+ program structure
+ program inputs
+ requirements
+ etc.

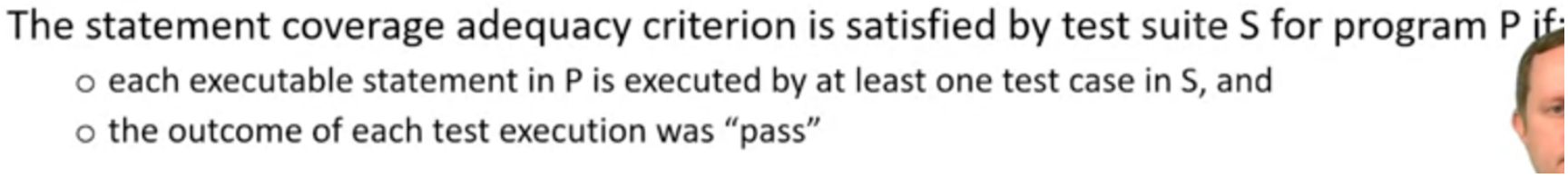
This measurement provides a way to determine (lack of) thoroughness of a test suite

### Adequacy criteria

Adequacy criterion = set of test obligations

A test suite satisfies an adequacy criterion if

- All the test succeed (pass)
- Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
- Ex:




## What do we know when a criterion is satisfied?

If a test suite satisfies all the obligations in the criterion, we have some evidence of its thoroughness

- We do not know definitively that it is an effective test suite
- Different criteria can be more or less effective

You would not buy a house just because it’s “up to code” but you might avoid a house that is not

## Where do test obligations come from?

**Functional**: From software specifications

Ex: if spec requires robust recovery from power failure, test obligations should include simulated power failure

**Fault-based**: from hypothesized faults (common bugs)

Ex: check for buffer overflow handling (common vulnerability) by testing on very large inputs

**Model-based**: from model of system
Models used in specification or design, or derived from code

Ex: Exercise all transitions in communication protocol model

**Structural** (white or glass box): From

Ex: Traverse each program loop one or more times

## Coverage: Is it Useful or Harmful?

Measuring coverage

- % of satisfied test obligations can be useful
  - Progress toward a thorough test suite
  - Trouble spots requiring more attention
- ... or a dangerous seduction
  - Coverage is only a proxy for thoroughness or adequacy
  - It’s easy to improve coverage without improving a test suite (much easier than designing good test cases)
  - The only measure that really matter is (cost) effectiveness

## Comparing adequacy criteria

Can we distinguish stronger from weaker adequacy criteria?

**Empirical approach**: study the effectiveness of different approaches to testing in practice:

- Issue: depends on the setting
- Cannot generalize from one organization or project to another

**Analytical approach**: describe conditions under which one adequacy criterion is provably stronger than another

- Stronger = gives stronger guarantees
- One piece of the overall “effectiveness” question

## Goals for Adequacy criterion

**Effective** at finding faults:
+ Better than random testing for suites of the same size
+ Better than other metrics with suites of similar size

**Robust** to simple changes in program/input structure

**Reasonable** in terms of the number of required test and coverage analysis

### Vast area of software engineering research

Primer not sufficient to gain full understanding

Several other resources exist in the space to develop further understanding

## Test-driven development

Flips the standard approach of “write code, write tests for the code”

By writing the tests first, you avoid some developer bias

Gives you a check to know when you’re done

**Process**

1. Add some test(s) to the test suite
2. Execute all tests in the suite and confirm new test(s) fail
3. Develop code to introduce new functionality
4. Execute all tests in the suite and confirm new test(s) pass
5. Refactor

### Tests up front

Focus on what the code should do, not how it will be implemented

Tests are not thrown together at the last minute

“Test-first students on average wrote more tests and... tended to be more productive”

### Conclusion

Develop tests up front, then code until tests pass

Building tests before implementation allows focus on behavior

Allows for incremental status updates

Key feature of many Agile approaches

## Section: Deployment

## Continuous integration

Automating quality controls work through tools

Each process runs and reports whenever a developer completes an action

Live reports give managers and developers visualization of quality

**Continuous integration platforms**

Services which provide extensible ability to perform actions

All began with automated testing of code after build

Many tools are now available for nearly all steps post-development

**Jenkins**

The big player in Continuous Integration and Automation servers

Plug-ins determine the capabilities of the server

One of the readings is a link to more information on the Jenkins Pipeline

**Continuous integration**

Automated processes designed to aid post-development tasks

Automated testing allows for rejection of commits

Tasks could include code style checking, static analysis, and more

Led to further improvements of the post-dev process

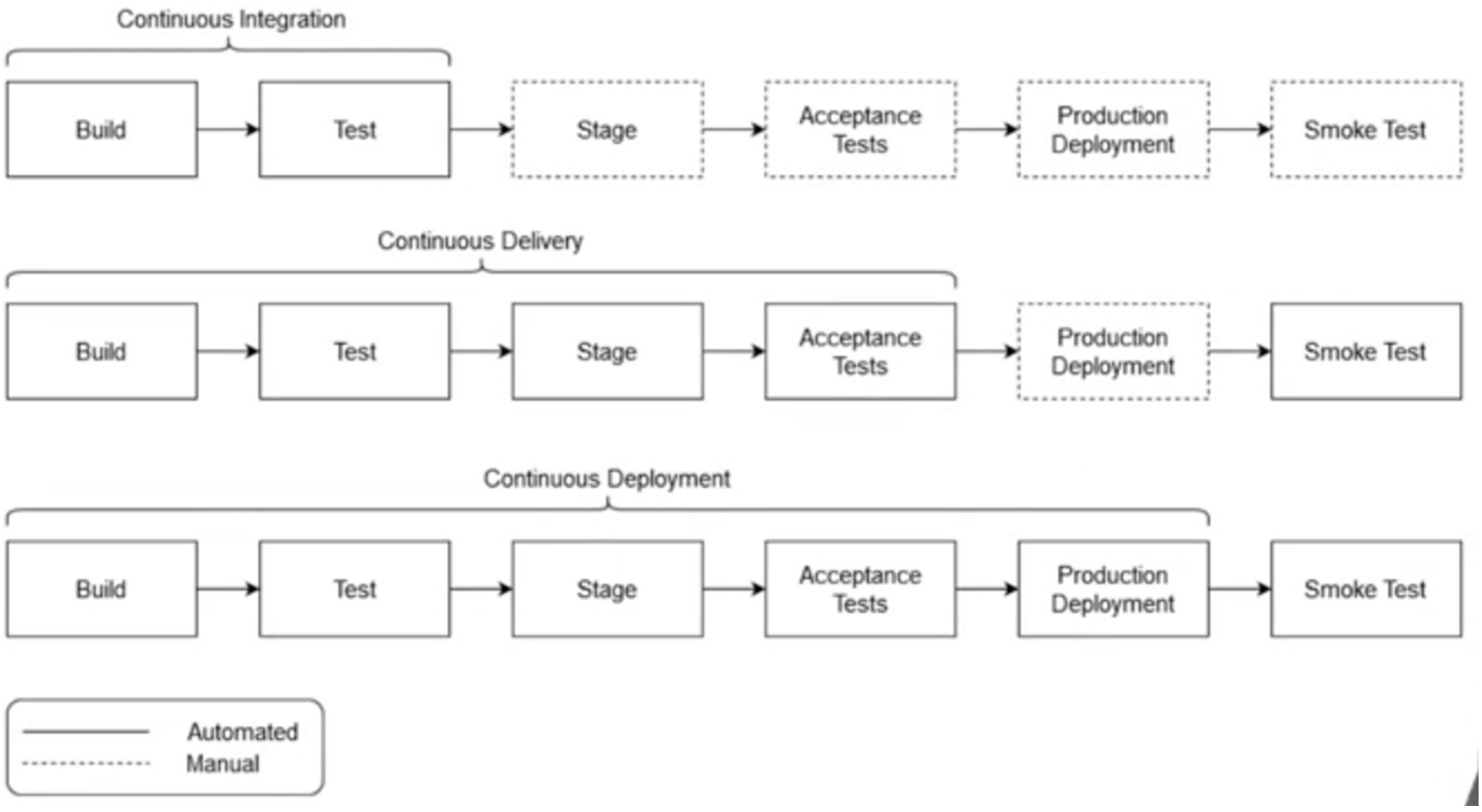
**Continuous Delivery / Continuous Deployment**

Continuous Delivery provides automated builds that are ready to deploy

Continuous Deployment provides automated build *and* deploy

Continuous Integration is part of both processes

**Visualizing the pipelines**




## Canary

An idea where rather than submitting our deployed code directly to all users, we take a small subset of users

Continuous pipelines are no simple feat

Many additional tools exist to aid in such automated systems

Canary helps deploy new code slowly, to user groups incrementally

## Conclusion

**Continuous Delivery/Deployment**

The natural extension of CI

Continuous Delivery produces deployment ready releases

Continuous Deployment automatically deploys new builds to users

Vast array of tools which support this process

> 💡 Blue/green deployment