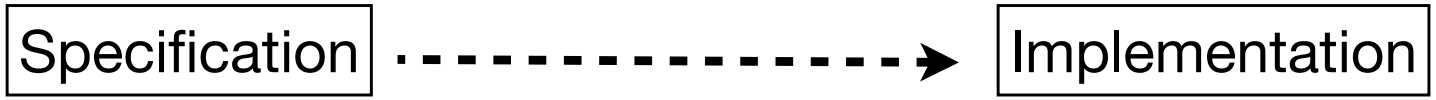
Introduction to Software Processes

How do we build software systems?



The **specification** may be in many forms:

- an idea in our head,
- a set of requirements written in English,
- a formal specification.

The **implementation** will be a software system.

The arrow represents the approach we use, often called a **Software Development Process**.

Is Software Process important?

Ever since we started building more complicated software systems (c. 1960s), there has been an interest in **software development processes**.

The "holy grail" in this area is a process that transforms a specification into an implementation.

Formal Methods solve this problem for simple systems, but this doesn’t address real systems building issues.

Software process remains a fluid area, largely because we aren’t (ever) satisfied with its success rate.

Software Failure

It is a popular complaint (true or not) that software projects typically:

- run overtime
- run over budget
- fail to meet their requirements,

if indeed they manage to produce a working product at all.

Some software projects fail very dramatically, and there are many examples. We'll look at just one briefly.

AT&T 1990 Network Crash

AT&T's long distance network crashed and remained down for nine hours in January, 1990.

A failure in one switch caused a propagation of error to all the other switches.

75,000,000 phone calls failed and 200,000 airline reservations were lost.

The failure was traced back to the C coding error on the next slide.

Source of the AT&T Network Crash

```
while (expression){
  switch expression{
    ...
    case (value):{
      if (logical) {
        sequence of statements
        break
      }
      else {
        other statements
      }
      statements after if...else statement
    }
  }
  statements after switch statement
}
statements after while statement
```

What is the effect of this break statement?

Note: The switch in question was the AT&T 4ESS model. 100 million lines of C code were reportedly written in the development of its successor, the 5E55

Let's Google "software failure"

www.cio.com › article › it-project-success-rates-finally-improving

IT project success rates finally improving | CIO

Feb 27, 2017 - According to the PMI research, across all industries, the average **percentage** of **projects** that are deemed **failures** is 14 percent; the average for IT **projects** deemed **failures** in 2016 also is 14 percent, the research revealed.

14% failure

www.quora.com › What-percentage-of-software-projects-fail

What percentage of software projects fail? - Quora

7 answers

Apr 25, 2015 - The **statistics** for **software project** delivery show that one in three **software projects** are truly successful. According to Standish Group's 2015 Annual CHAOS ...

What is the **failure rate** of corporate custom **software projects**? 22 Dec 2014

Why do so many **software projects fail**? 2 Mar 2017

More results from www.quora.com

19% failure

www.zdnet.com › article › study-68-percent-of-it-projects-fail

Study: 68 percent of IT projects fail | ZDNet

Jan 14, 2009 - Study: **68 percent** of IT projects fail. A new report, notes that success in **68 percent** of technology projects is "improbable".

68% failure

www.theserverside.com › opinion › Dont-contribute-to-the-high-IT-p...

Don't contribute to the high IT project failure rate

Dec 14, 2018 - Enterprise **software development** is difficult, so it's no surprise to discover there is a high IT **project failure rate**. Here's how to buck the trend and ...

"70% of the companies polled reported having at least one project failure in the last year"

www.objectstyle.com › software-projects-failure-statistics-and-reasons

Why 50% of IT projects fail, and how to NOT let that happen to ...

Feb 14, 2018 - **Software projects** seem to follow the analogy, if distantly. ... Source: A Replicated Survey of IT **Software Project Failures** by Khaled El Emam and ...

50% failure

www.information-age.com › News › Topics › Business & Strategy

Why IT projects continue to fail at an alarming rate

Feb 16, 2018 - Why IT **projects** continue to **fail** at an alarming **rate** ... biggest barrier to successfully deploying modern enterprise **software** is custom coding.

"UK Wasting £37 Billion a Year on Failed Agile IT Projects"

What is a Software Process?

In general, we can view a software development process as comprising three main components:

A **set of models** to be produced.

A **notation** for describing these models.

A **process** to be followed.

We'll see that the Waterfall process involves creating analysis and design models, while Agile processes involve no modelling at all.

We now explore *process* in more detail.

Aside: Formal Methods

Through the 80s and early 90s there was a great interest in **Formal Methods**.

The view was that software development could never produce reliable software components.

Thus approaches based on formal specification and proof were developed:

- formal specification of requirements

- formal proof that the software was a correct implementation of the specification

What happened to Formal Methods?

The software community proved the formalists wrong: reliable software could be developed without use of formal specification/proof.

The venerable formalist, Tony Hoare, conceded this in 1996:

> "... the problem of program correctness has turned out to be far less serious than predicted... the techniques employed to achieve reliability are little different from those which have proved effective in all other branches of modern engineering..."

Note: C. A. R. Hoare, "How did software get so reliable without proof?", Invited Lecture, International Symposium of Formal Methods Europe, 1996

Roadmap of this Section

We start by looking at two “traditional” methods, both of which have considerable limitations:

Code and Fix

The Waterfall process

We then consider the key aspects of all more recent methods:

Incremental

Iterative

Finally we examine the contemporary approach:

Agile Processes

(e.g. Extreme Programming, Scrum, Kanban etc.)

"Code and Fix" (aka hacking)

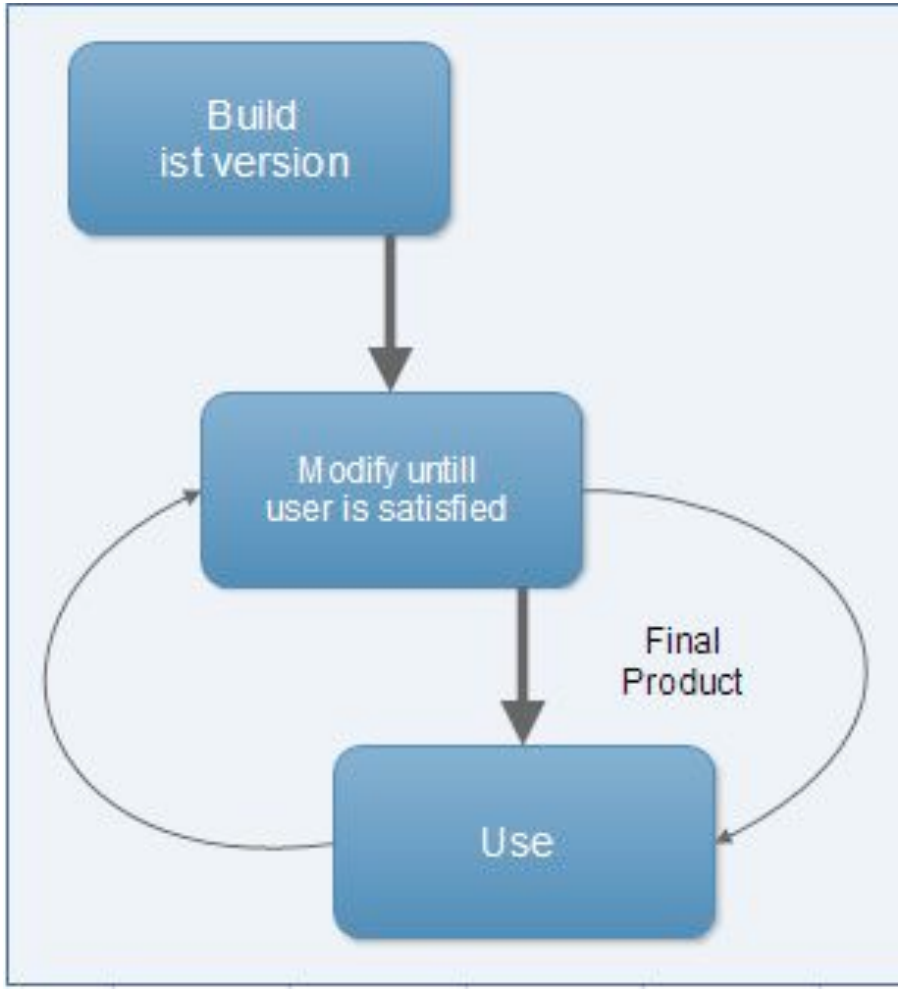
“Code and Fix” involves starting coding as soon as you have an idea of what to do.

When you encounter a bug/new requirement, you just fix it or code it up.

This is a minimalist process: **code** up what has to be done and **fix** bugs as you go. No design and no planning!

A bit like trying to get to Galway by heading West, and dealing with obstructions and challenges as they arise.

The Code and Fix Model



This resembles a primitive Agile process.

More on this later.

Can “code and fix” work?

This approach can work for a resourceful individual, but is impossible to apply when a large team is involved.

Even when a working product is achieved, further development is very challenging as the software was not developed to permit extension.

It may also be the chosen method for a small team with a very tight deadline.

Types of Systems

Bertrand Meyer proposed a useful categorisation of systems:

Acute: Failure may have a huge impact, possibly including loss of life (medical systems, transport control systems, critical banking systems, etc.).

Business: Commercial software, most web applications, games, almost all the software you use.

Casual: Student assignments, experimental software, informal websites.

A: Requires specialist treatment, not part of this module

B: The focus of this module

C: Code&Fix will suffice!

The Waterfall Process

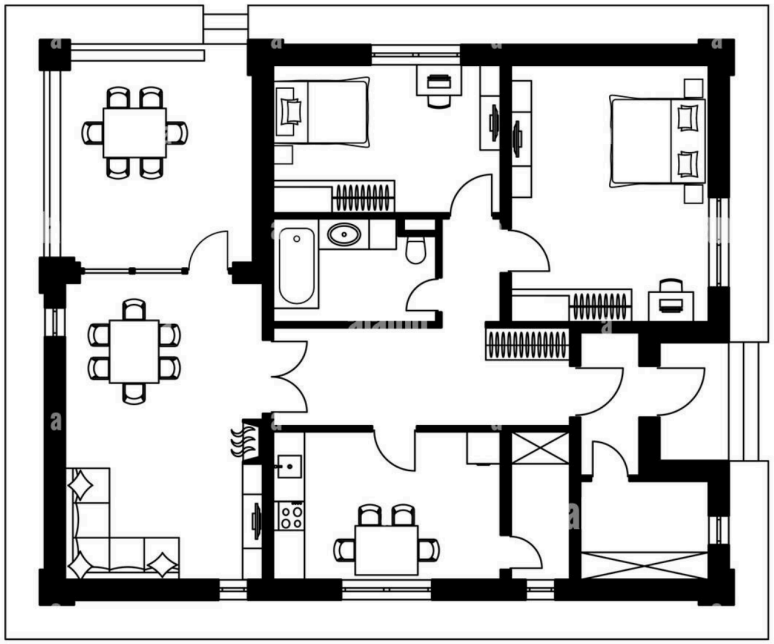
Initial attempts (~1960s) to define a software development process were based on approaches used in accepted engineering practice.

Waterfall is still in use today, and understanding it helps in grasping the now commonplace Agile processes.

Before delving into the Waterfall process, let's clarify what is meant by "accepted engineering practice."

Although the Waterfall process is largely discredited, note that in a 2018 StackOverflow developer survey, 15% of the 57,000 respondents stated that they used a Waterfall development process.

How construction works



Let's say we're building a house (or a manufacturing plant, bridge, road, rocket, etc.)

We start with high-level designs like the ones above.

Low-level designs add more detail.

Eventually we have enough detail that physical construction can begin.

How construction works

0 - Strategic definition.
1 - Preparation and brief.
2 - Concept design.
3 - Developed design.
4 - Technical design.
5 - Construction.
6 - Handover and close out.
7 - In use.

Source: RIBA

Requirements,
analysis and design
all happen on paper

Hard hats, boots, concrete

Passing building on
to customer

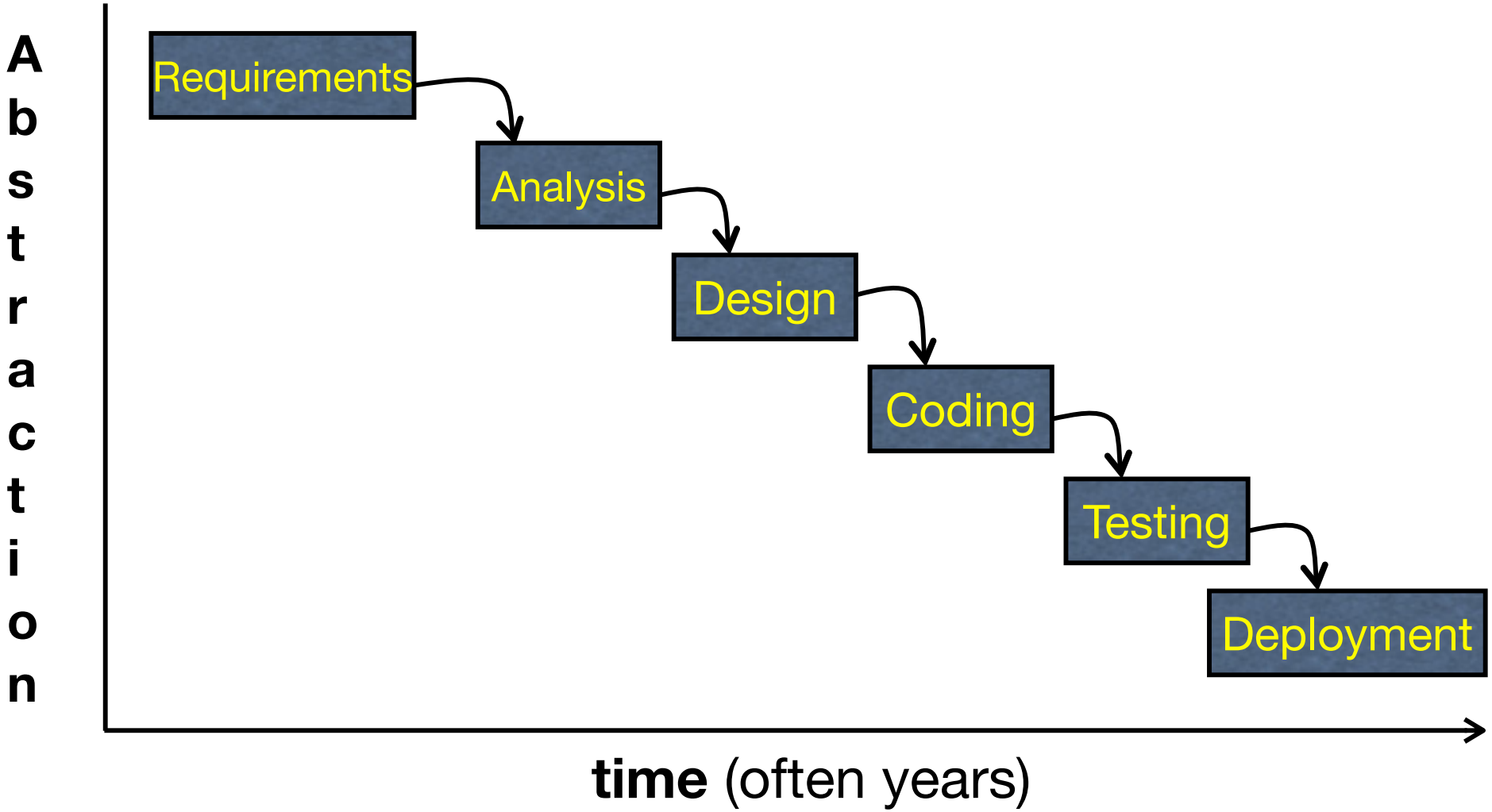
Steps 0 to 7 happen in sequence.

Steps 0 to 4 involve "blueprints" -- paper designs that are relatively easy to change. Each step adds more detail.

Once Construction is underway, changes become practically impossible.

The Waterfall Process

The Waterfall Process mimics the typical construction process we just saw.



Phases of Waterfall

Requirements: Figuring out the customer's needs.

Analysis: Building a model of ***what*** the system is to do.

Design: Building a model of ***how*** the system is to do it.

Coding: Write the code.

Testing: Test the code.

Deployment: Making the system available to customers.

Properties of the Waterfall Model

Involves a number of **distinct phases** that are completed **in order**: requirements, analysis, design, implementation, testing.

The output of each phase serves as input to the next phase (this is the "**waterfall**").

Although appealing in many ways, the Waterfall process has several critical shortcomings.

Note: Interestingly, one of the first descriptions of the Waterfall Process [Royce, 1970] already identified many of its shortcomings.

Weaknesses of the Waterfall Model

There are three main problems with the Waterfall approach:

1. Handles changes in requirements badly.
2. Testing happens only very late in the process.
3. Real customer feedback comes only very late in the process.

Other issues include:
- Developers spend months/years not writing software
- Keeping documentation/code consistent is a burden
- Analysis->Design mapping is famously challenging
- No attention is paid to integration of software
- Human error discovered only late in the process

1. Changes in Requirements

Waterfall assumes that the requirements can be fixed at the start of the process.

In the dynamic environment where most software lives, this is an unrealistic assumption.

Changing a requirement e.g. in the design phase involves revisiting the requirements/analysis/design phases -- a lot of work.

The alternative is not to accept the new requirement, but this leads to delivering an obsolete system to unhappy customers.

2. Late Testing

The best way to uncover errors in software is:

TESTING

Testing analysis/design documents is possible,
sort of...

... but testing is never more conclusive than when
testing executable code.

In Waterfall, testing happens at the end of the process,
so it is only here that errors are discovered.

Handling an analysis error at this stage involves
revisiting the analysis/design/implementation phases.
Tedious and time consuming.

3. Late Customer Feedback

Customers can give feedback at any stage.

However, incisive feedback only comes when they get to use the system.

In Waterfall, this only happens when the system is finally deployed.

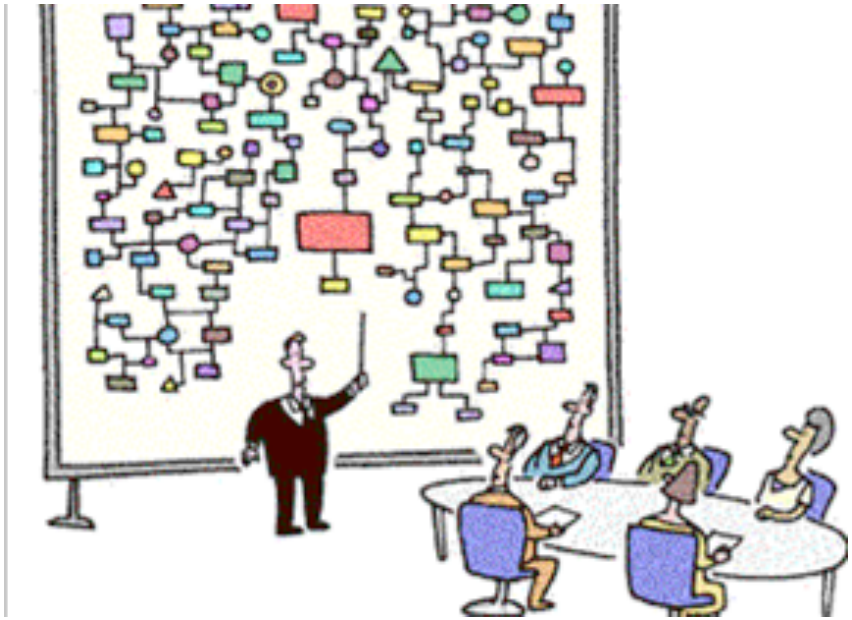
This is far too late in the process, and the typical customer response is something like:

"*Oh, this isn't what I expected at all...*"

Analysis Paralysis

The problems outlined often led to waterfall projects suffering from what is jokingly called **analysis paralysis**.

The Analysis has to be perfect before Design starts. Perfection is impossible, so progress is halted.



With the project deadline looming, the analysis&design was typically ditched and the developers just start coding.

The result was little better than code-and-fix.

Improving on Waterfall

Current software development processes aim to manage change and human error better, and place a heavy emphasis on early deployment and testing.

Two key features of any contemporary software process are:

Incremental
Iterative

We'll look at these properties separately before considering concrete development processes that are based on them.

(Although they are distinct properties, they go very much hand-in-hand)

Incremental Development

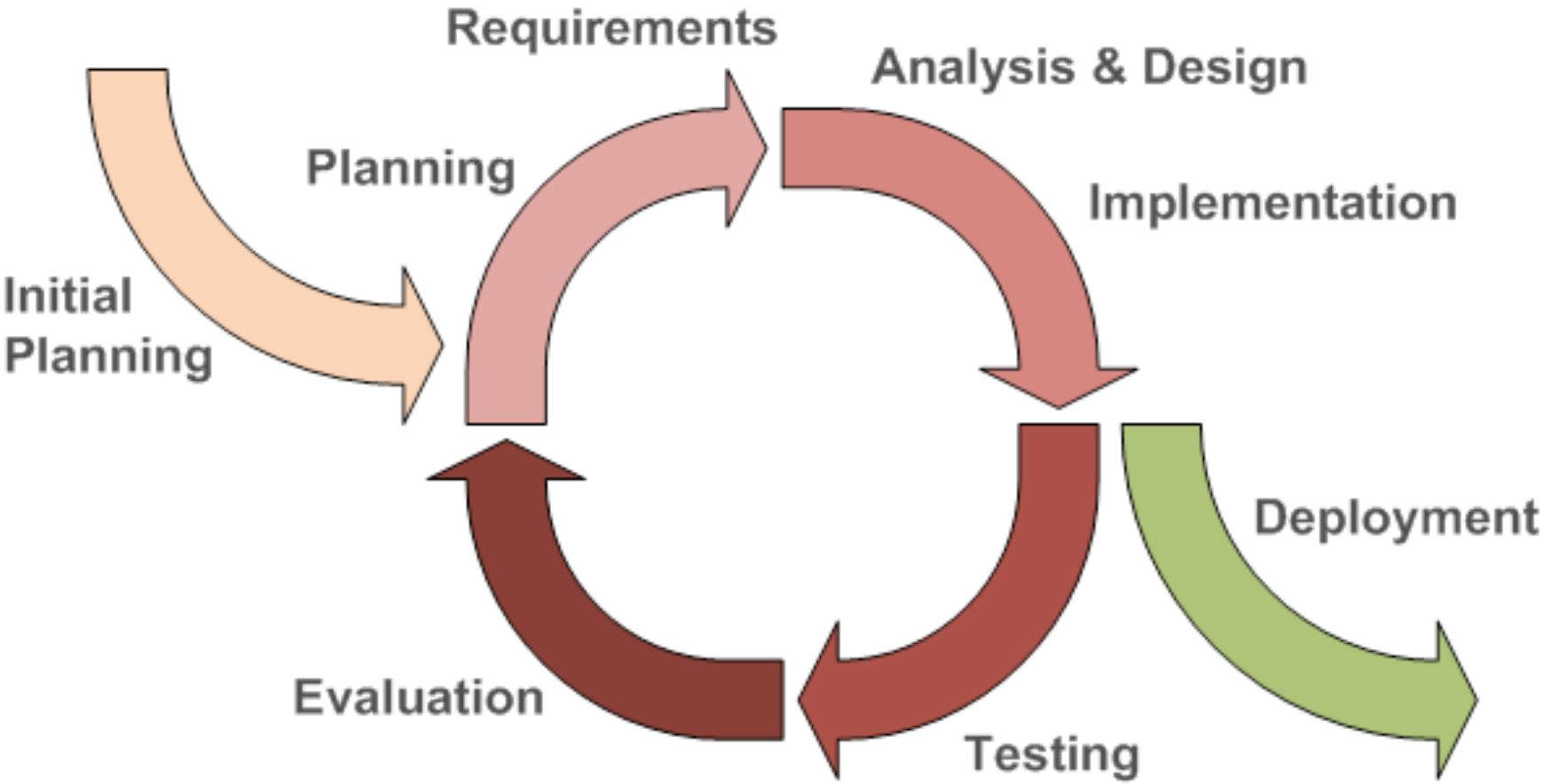
In an **incremental** (or evolutionary) process, a simple working prototype is initially produced and this is gradually extended (evolved) to the final system.

Advantages of incremental development:
- Early testing and feedback from customers
- Easy integration of new requirements

Risks of incremental development:
- Hard to manage -- will the increments ever result in the final system?
- How is the overall system design developed?

Iterative Development

The key property of an **iterative** model is that several passes through the various stages of the software process are made, e.g.



Iterative vs. Incremental Development

Iterative and *Incremental* are distinct properties.

It is possible (in theory) to have one without the other.

In practice they always go together.

As we'll see, contemporary software processes (Agile processes) are invariably both iterative and incremental.

BDUF

Big **D**esign **U**p **F**ront -- a pejorative term used by Agilists to refer to Waterfall-like processes.

Joel Spolsky* wrote on BDUF (in 2005):

> *I can't tell you how strongly I believe in BDUF... I'm proud to use it, no matter what the XP fanatics claim. They're just wrong on this point and I can't be any clearer than that.*

It all depends on context. Full Waterfall is rarely appropriate, but BDUF/LDUF definitely has its uses.

* Joel Spolsky: Creator of Trello and StackOverflow.

What is *Software Maintenance*?

Common definition:"Software maintenance is the process of changing, modifying, and updating software to keep up with customer needs."

What percentage of software development falls under this definition?

Traditional answer: ~70%

The truth is closer to 99%

The vast majority of software developed these days falls under the "maintenance" definition.

i.e. development ~= maintenance

Let's ask ChatGPT!

What is the difference between software development and software maintenance?

Partial Answer (abridged):

Development is the initial creation of the software. It involves:
- Planning & Requirements Gathering
- Designing the software
- Implementation (Coding)
- Testing: Ensuring the software works as expected.
- Deployment: Releasing the software for use.

This answer is wrong (as expected).

It's based on the Waterfall process, which was largely discarded by the industry in the early part of this century.

Beware of using web sources, or AI tools, for this module!

Agile Processes

See separate slide deck
for this topic.

Snapshot of Irish Software Industry

Searching LinkedIn IT jobs in Ireland for terms related to software processes:

	Jan'20	Jan'21	Jan'23	Jan'24	Jan'25
"**Agile**"	3,006	3,033	6,583	2,382	1,826
"**Scrum**"	965	984	2,821	1,459	1,283
"**Waterfall**"	185	162	291	23	**?**

Not a definitive measure of course, but gives an idea of the popularity of these techniques.

Final Remarks

Software Development Process is a dynamic area where research struggles to uncover any generalisable conclusions.

In this type of space, advocacy flourishes and you'll find many "influencers" online promoting their own perspective.

Most are worth listening to; none have the whole truth.

Summary

The process we use for developing software is a critical issue, and there is no definitive best approach.

There has been an evolution both in terms of the metamodel used and development process used.

The object-oriented metamodel is dominant in the software industry.

Most current software processes are Agile in nature, including Scrum, Extreme Programming, Kanban and others.

Agile Processes

In the past two decades there has been a move towards lightweight software development processes, collectively known as **Agile** processes.

Waterfall tries to **prevent** new requirements by negotiating a contract upfront.

Waterfall tries to **prevent** error by adding more process.

Agile **accepts** new requirements and errors occur, and remains flexible (agile) so as to be able to deal with them.

Agile Manifesto

The **Agile Manifesto** was proposed in 2001 by a number of people, including well-known names like **Kent Beck, Robert Martin, Ward Cunningham, Dave Thomas**.

These were real software people, trying to express how software was actually being developed.

In a sense, Agile was attempt to create a scalable methodology from code&fix.

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

individuals and interactions *over processes and tools*
working software *over comprehensive documentation*
customer collaboration *over contract negotiation*
responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

12 Principles of Agile

1. Our highest priority is to satisfy the customer through **early and continuous delivery of valuable software**.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

12 Principles of Agile

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

Note: Some of notions are dated, e.g. "face-to-face conversations" have been very successfully replaced by online interactions in physically distributed teams.

12 Principles of Agile

7. Working software is the primary measure of progress.

8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to **technical excellence** and **good design** enhances agility.

12 Principles of Agile

10. Simplicity – the art of maximising the amount of work not done – is essential.

11. The best architectures, requirements, and designs emerge from **self-organising teams**.

12. At regular intervals, the team **reflects on how to become more effective**, then tunes and adjusts its behaviour accordingly.

Agile Principles Summarised

Deliver valuable software regularly.

Be open always to new requirements.

No barriers between product team and developers.

Motivated developers.

Communication.

Well-designed, working software.

Little documentation.

Sustainable development.

Self-organising teams.

Reflection.

Agile Processes

There are many agile processes. Three of the best known are:

Scrum: This is a pure agile process, not only for software We'll look at it in a bit more detail.

Extreme Programming: Very much focused on software. We'll look at some of its widely-accepted engineering practices in more detail.

Kanban: Very lightweight. More suitable for "endless" processes like services, also suits better when there is little customer involvement.

Scrum

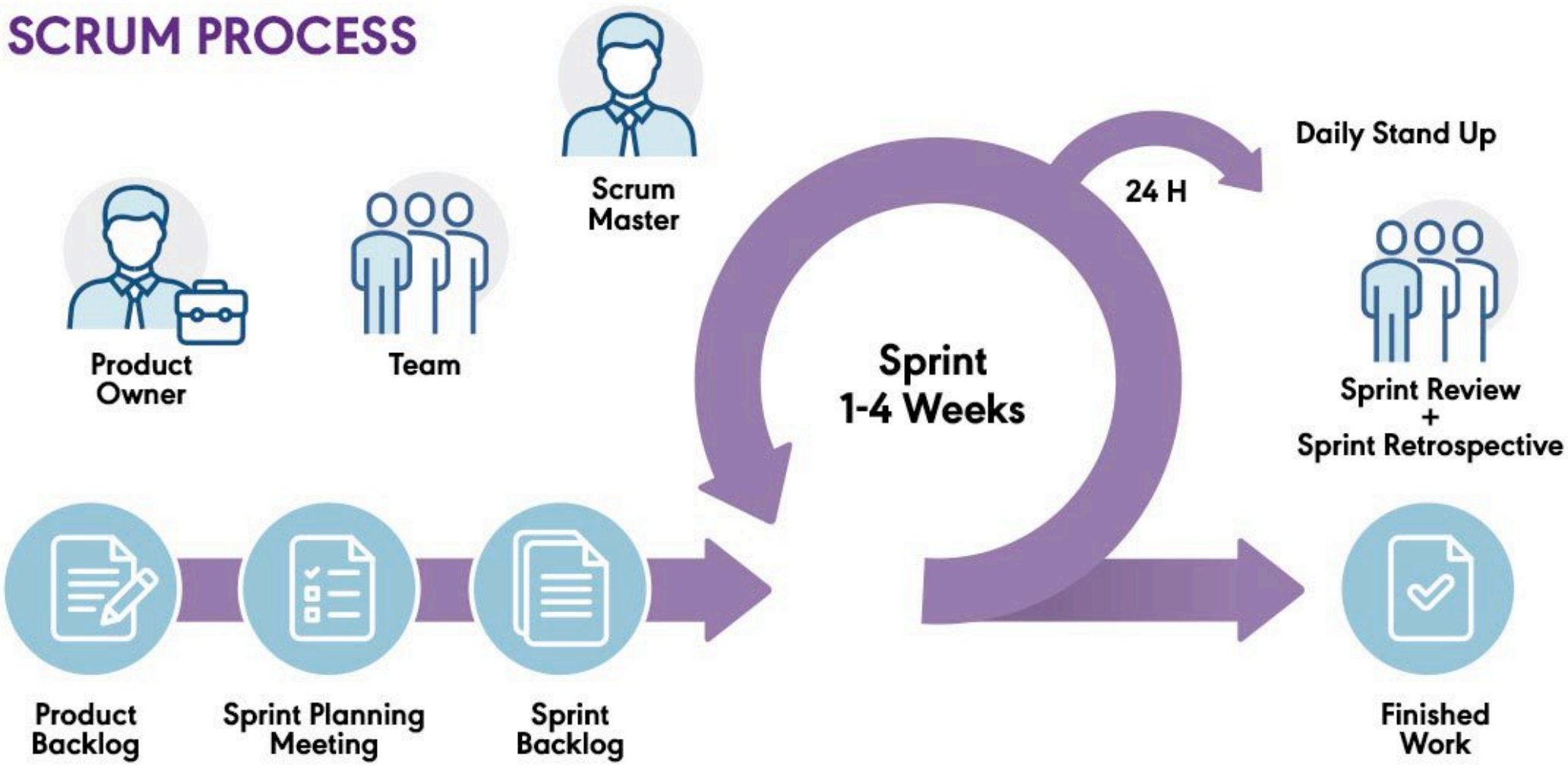
First proposed in 1995.

Scrum is very commercialised and the area is heavily trademarked.

Scrum is unrelated to software -- you can use it to build non-software artefacts as well.

It's become the most popular agile process.

Overview of Scrum



Thanks to pm-partners for the image.

Scrum Roles

Product Owner: Non-technical role. Has a clear **vision** for the product being developed. Writes user stories, answers team's product-related questions.

Scrum Master: A Scrum expert. Ensures that the team adheres to the Scrum framework. May also be a developer.

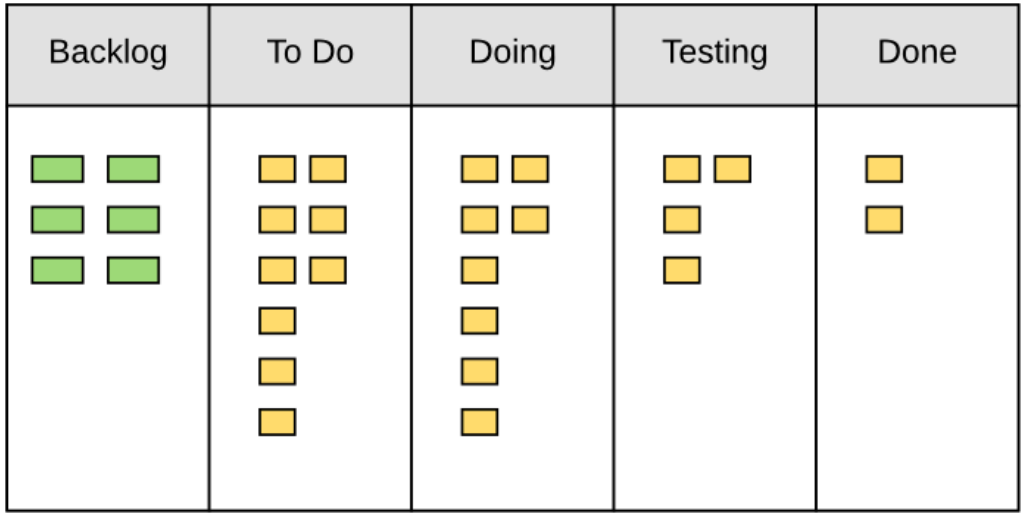
Software Developer: A software expert focussed on implementing the user stories written by the Product Owner.

Scrum Artifacts

Product Backlog: Prioritised list of user stories to be implemented, possibly other tasks as well.

Sprint Backlog: A subset of the product backlog to be implemented on this sprint, decomposed into a set of tasks.

Scrum Board: A visual representation of the current state of the sprint, e.g.



Scrum Events

Sprint: A time-boxed iteration (~2 weeks) leading to potentially a new software release. Aka **iteration**.

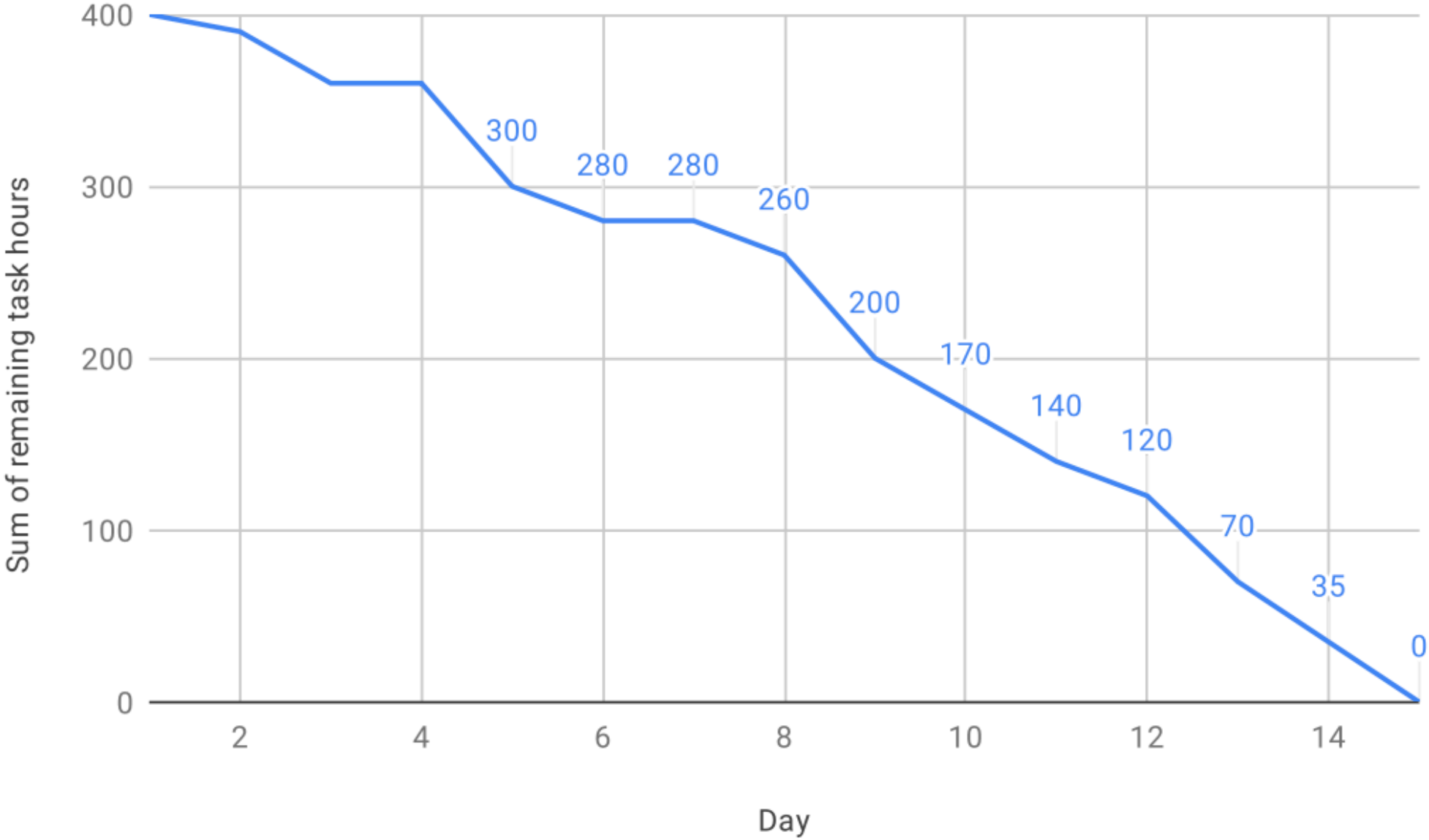
Sprint Planning: Meeting at start of sprint to decide on stories to be implemented, and their decomposition into tasks.

Stand-Up: *Short* daily meeting to start the day. Everyone summarises what they achieved yesterday, plans for today, and obstacles to progress.

Retrospective: Meeting to reflect on how the sprint went, how the team could do better etc.

Burndown Chart

Shows how much work is left to do to complete the tasks of the sprint, e.g.



Sample User Story

SWITCH WEAPONS WHILE SPRINTING

2 POINTS

AS A PLAYER,

I WANT TO SWITCH BETWEEN MY EQUIPPED WEAPONS WHILE SPRINTING

SO THAT I DON'T HAVE TO STOP TO SEE MY INVENTORY

Velocity

Each story has an associated number of points representing the difficulty in implementing it.

The number of points is assigned by the developers during sprint planning.

The **velocity** of the team is the number of story points it can complete during a sprint.

Obviously this is not an exact measure, but it gives a rough idea of how the team is doing.

More importantly, it informs the Product Owner how many stories they can expect the team to implement in a sprint.

Other Aspect of Scrum/Agile

Cross-functional Teams: Team must have all the expertise required to deliver functionality.

"done": The notion of "done" must be defined. Implementation must at least pass all tests, may also be other QA/UI/UX etc. issues

Servant Manager: Managers are not traditional bosses. Their main role is to facilitate the dev team.

Framework: Scrum is really a tailorable framework, not a fully-specified process.

But how about the software?

Scrum is often used on software projects, but is not intended just for software.

This is why it has very little to say about critical issues like testing, quality, usability, security etc.

Extreme Programming (XP) is very much about software.

We now look at some engineering practices, mostly from XP, which are mainstream in all agile processes.

Let's consider what happens within a sprint...

No BDUF

Little or no upfront design is performed.

UML diagrams may be drawn, but only for the purpose of immediate communication. Design documentation is not created or maintained.

Everything is kept as simple as can be. Several Agile acronyms reflect this philosophy:

YAGNI: You Ain't Gonna Need It

KISS: Keep it Simple Stupid

TSTTCPW: The Simplest Thing That Could Possibly Work

Testing

Testing is essential -- it's how we know that our software works.

Our focus here is on **unit testing** -- the tests a developer creates as they work.

The test suite can be run at any time to provide assurance that the existing code is still correct.

(XP goes a step further and demands that tests be written first. More on this later.)

We'll come back to **testing** later in the module.

Unit tests for correctness

The test suite is actually an **executable specification** of what the system is to do.

Contrast this with a complex design where it's impossible to assess if a given program is a correct implementation or not.

The promotion of testing to this level of importance has a vital characteristic that can be exploited:

> Whenever the code is updated, it is easy to test if a new bug has been introduced.

How is the software designed?

So far we have a Scrum process, with developers coding and testing.

How can we build a large (~1M LOCs) software system by just coding and never considering software architecture and design?

We use an approach called **emergent design** or **pay-as-you-go design**.

Emergent Design

When a developer tackles a new story, they check if it can be implemented easily in the current program.

If not, they **change the design** of the software so that:

(1) it becomes easy to implement the new user story.

(2) the behaviour of the software remains unchanged

(1) makes the task at hand easy

(2) ensures that no new bugs have been created

This process is called **refactoring**.

Running the test suite after a refactoring is likely to detect any new bugs.

We'll come back to **refactoring** later in the module.

Simple Example

You're tasked with implementing a story.

You figure out how to do it. It involves adding new functionality to a method that is already quite long.

Code&Fix: Just do it and move on!

Agile Development:

(1) Refactor: Split the long method into several shorter more cohesive methods.

(2) Test: Run your existing unit tests. This reassures you that nothing is broken.

(3) Implement: Code up the new story, writing unit tests for it of course.

We'll come back to **code smells** (incl. Long Method) software properties like **cohesion** later in the module.

Floss and *Radical* Refactoring

Ideally the software design emerges in baby steps as described on the previous slide.

Sometimes deeper problems with the software design are uncovered, requiring a more radical approach.

Radical refactoring occurs (rarely!) when the whole program design is overhauled.

This process may take a whole sprint and involve the entire development team.

Note: **Radical Refactoring** is aka **Root Canal** refactoring.

Agile Code

The code is the essential artefact produced in an agile process.

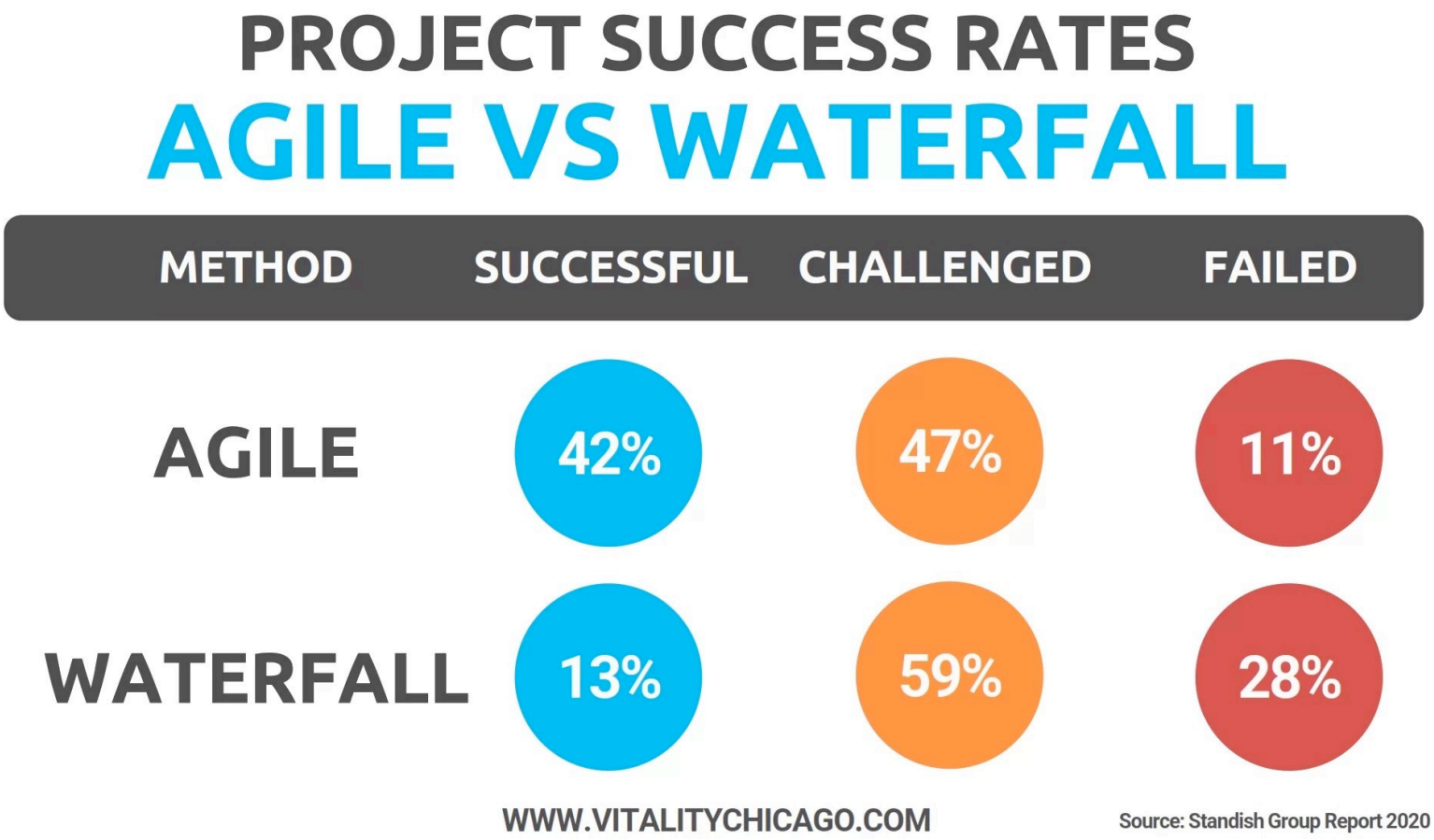
It has to be of high quality.

The **code is elegant and clear**. It has to be because developers work with it continually.

The **design is simple**. The design is only for the current system, not for possible futures.

The code works. This can be easily verified by running the test suite.

Does Agile work well?



From the so-called *CHAOS report*, an annual survey of IT projects in the US. To be taken with a large grain of salt.

Why does Agile work for software?

You can't build a house in an Agile way

"OMG this is so nice!"

"Just rotate it 30 degrees to catch the morning sun"

Software vs. Physical Construction

Waterfall

- Requirements
- Analysis
- Design
- Implementation
- Testing
- Release

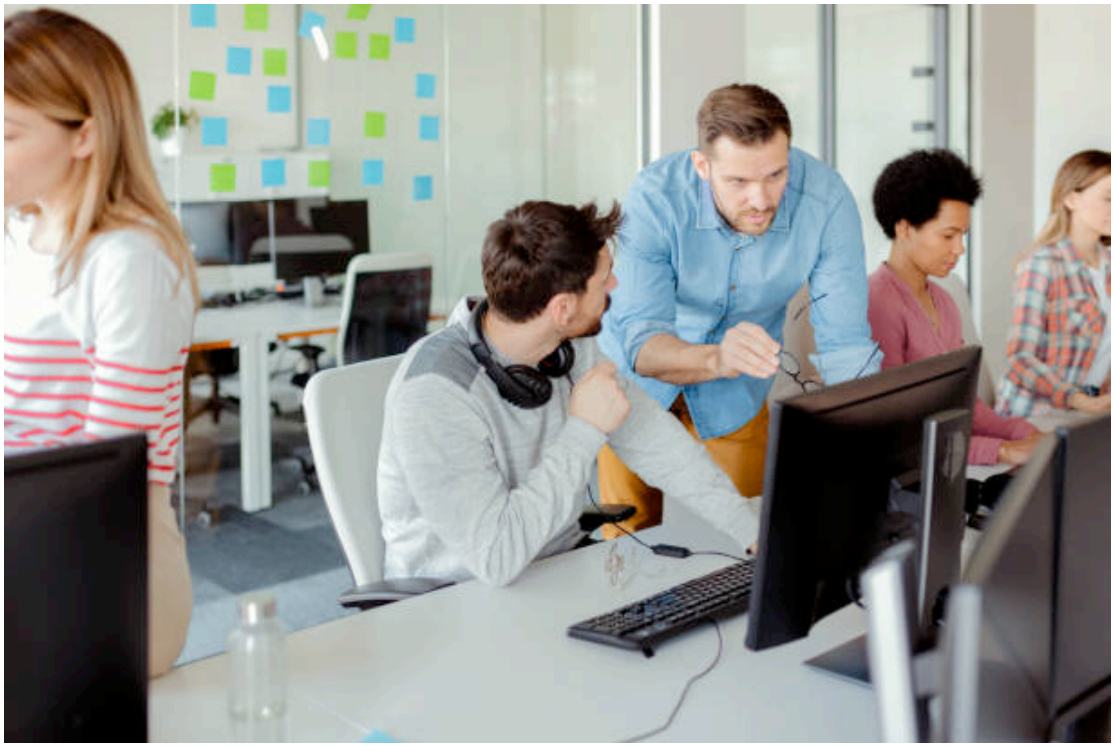
Physical Construction

- Strategic definition
- Prep. and brief
- Concept design
- Developed design
- Technical design
- Construction
- Handover and close

Yellow: Creating models (blueprints) on paper
Orange: Actually building the system
Green: Releasing the system to customer

The modelling & blueprint phases look similar!

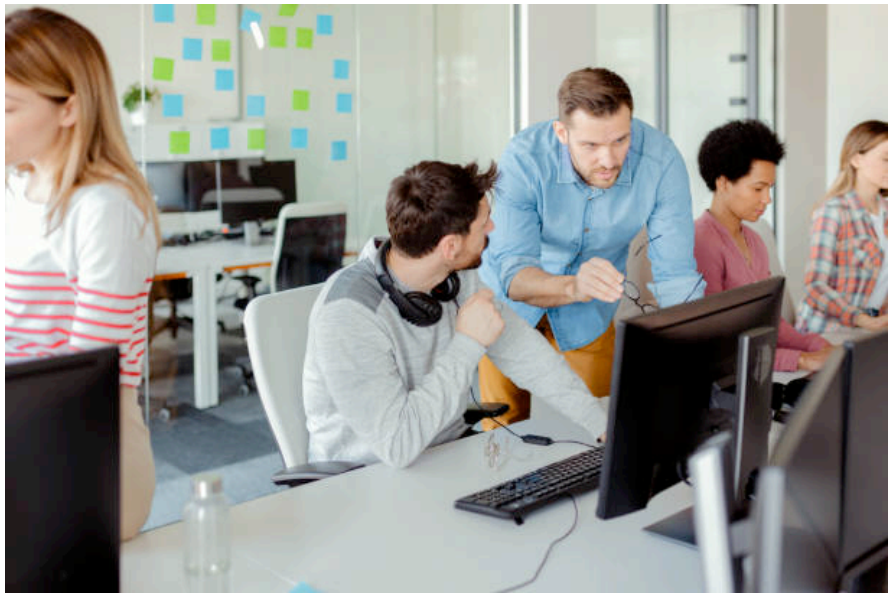
The modelling phases of both software and physical construction processes look similar



How about the construction and implementation phases?

Software Implementation vs. Physical Construction

Software Implementation



Physical Construction

In essence, the process of transforming our detailed blueprints (software) into what the customer wants (a running system) is fully automated!

This is what makes an agile approach possible in software.

Resources

Read the relevant parts of the recommended textbook: *Software Engineering: A Modern Approach* by Marco Tulio Valente.

http://www.agilelearninglabs.com/resources/scrum-introduction
is a clear introduction to Scrum.

https://www.atlassian.com/agile
Atlassian provide clear "no nonsense" tutorials on Agile.

Summary

Agile most suited to small teams (<=12 people) on projects where the requirements are likely to change frequently.

In ways, it is the antithesis of the Waterfall process.

Agile has worked well in practice, and no doubt this field will continue to evolve.

DevOps

Comp 47480: Contemporary Software Development

What is DevOps?

Development involves writing software.

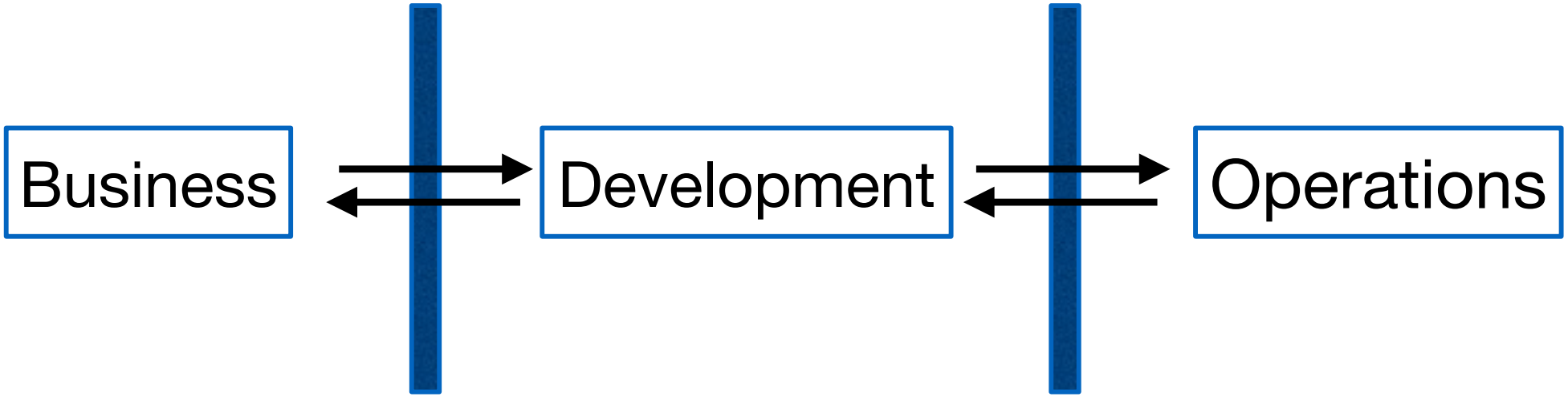
Operations (**Ops**) is the management, monitoring and operation of a system in production.

DevOps is the practice of operations and development engineers working together in the entire software lifecycle, from development through to release.

In this section we look *briefly* at what DevOps is.

Where did DevOps come from?

The traditional view of software development and its context:

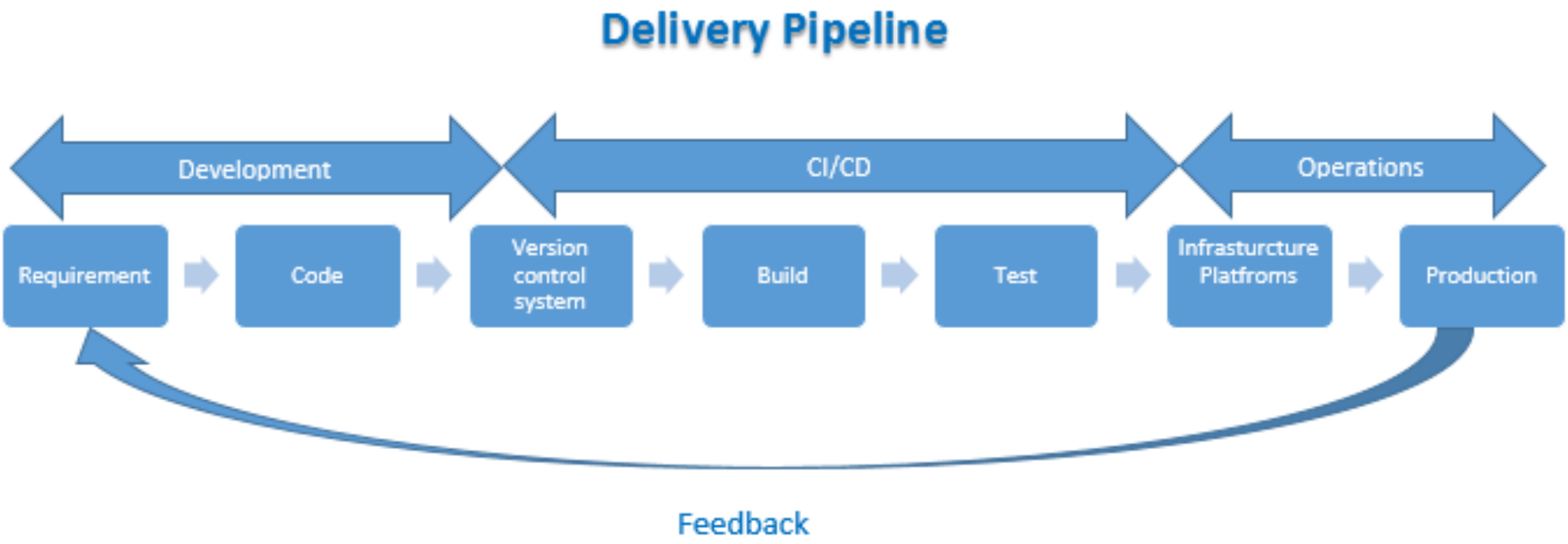


Agile aims to dissolve this wall

DevOps aims to dissolve this wall

DevOps Delivery Pipeline

DevOps is based on a pipeline, e.g.:



There are many versions of this pipeline. The key feature is that, ideally,
the process from commit to production is automated.

DevOps Principles 1

Continuous Integration, Continuous Delivery. Get new releases into production smoothly and quickly.

Microservices architecture. System designed as a number of containers interacting through well-defined APIs, typically using HTTP.

(Not essential for DevOps, but is the most common architecture employed in a DevOps context.)

DevOps Principles 2

Infrastructure as Code. Provisioning and managing servers is performed by code, so version control is possible and the tasks become standardised and repeatable.

Monitoring and Logging. Active monitoring is essential to keeping services running efficiently 24x7.

Automate Everything! Testing, workflows, infrastructure, deployment, monitoring ...

Integration

Integration is where we bring together software components that have been developed separately.

This seemingly innocuous activity has led to huge problems in practice ("integration hell").

Essentially we are testing if multiple developers had **precisely** the same understanding of the contract between them (answer: no).

One of the most famous integration problems is described on the next slide.

NASA's Mars Climate Orbiter

The Mars Climate Orbiter was a robotic space probe launched by NASA in 1998.

On approach, it got too close to the planet and was destroyed by the atmosphere.

Why?

> The NASA-developed on-board software used the standard *SI* system.
>
> The ground software was developed by Lockheed-Martin used the *United States customary units* system.

Impulse as measured in *pounds-force seconds* and *newton seconds* vary by a factor of ~4, so the orbiter was doomed.

Aside: Blame Culture

The spec clearly stated that SI units (newton seconds) were to be used. NASA used them. Lockheed-Martin didn't. Who did NASA blame?

"People sometimes make errors," said Edward Weiler, NASA associate administrator for space science. "The problem here was not the error; it was the failure of NASA's systems engineering, and the checks and balances in our processes, to detect the error. That's why we lost the spacecraft."

This attitude is evident in all healthy software development environments. In a retrospective, the issue is never "who is to blame" but "how can we do it better?"

The root cause of the loss of the orbiter was the lack of integration testing.

Continuous Integration

The traditional practice is "big bang" integration.

Each feature under development is committed to a different branch ("feature branch") in the (Git) repository.

When the time comes for a new release, a protracted, painful integration phase is required to get the different features to work together.

DevOps demands instead *continuous integration*, where integration happens frequently, perhaps multiple times per day.

Repository Organisation

A large company may store its software in multiple Git repositories.

This may lead to inconsistencies between repos, and make the software harder to refactor, e.g. a method renaming may require multiple, coordinated commits.

"Extreme" DevOps suggests a single **monorepo**, with just **one single branch** (main).

So every commit, by any developer, is to the head of main.

Sounds insane, but this is the practice in several of the best-known software companies.

Automation

Automation is key. CI is very challenging if human intervention is required.

- Automated builds.
- Automated unit testing.
- Automated integration testing.

If e.g., performance is a concern, testing this should also be automated.

CI Challenges

CI may be hard to achieve in certain contexts.

End-user tests: This type of test may be script-based and not automatable.

External Mandates: Software developed in certain domains, e.g. medical, finance may be required to go through certain non-automatable checks.

Open source: Developers are typically less reliable and not available on a daily basis.

Code Reviews

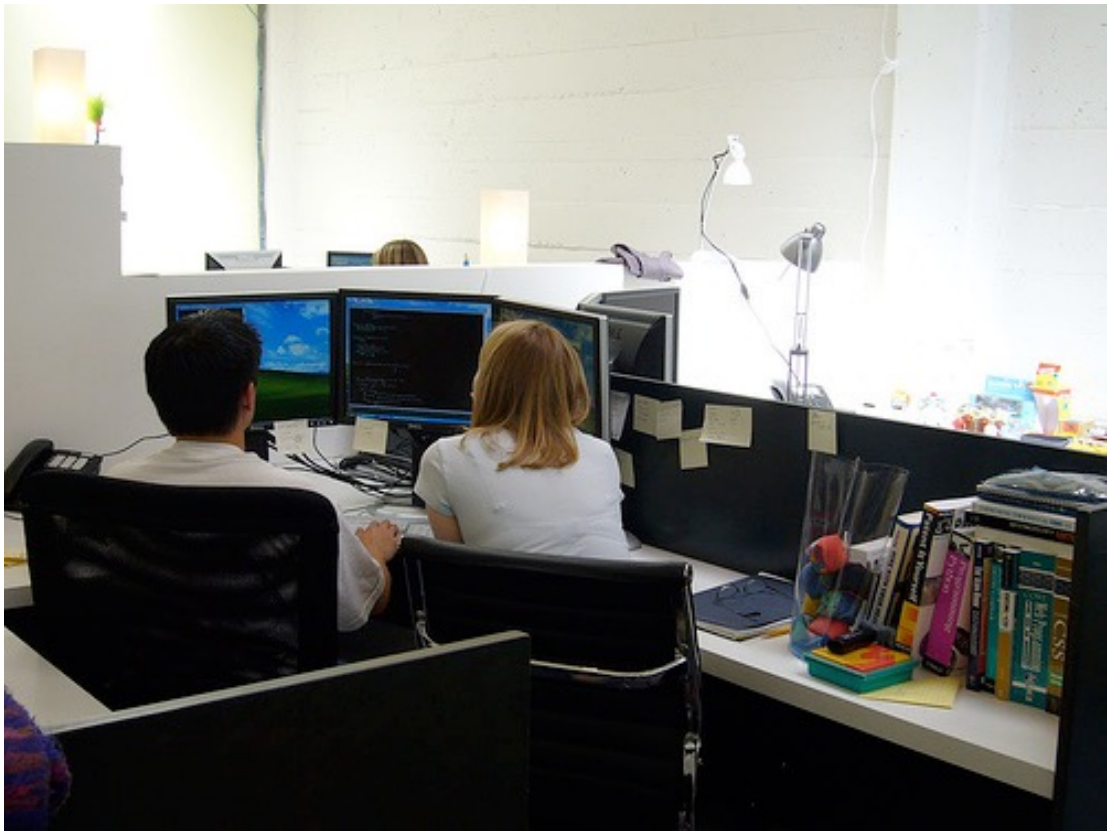
How can code reviews fit into a a CI process? They require another developer to review your code.

Extreme Programming proposed **pair programming** as a solution (see next slides).

In a CI context, code reviews may be replaced with automated tools like SonarQube that try to measure the quality of the committed software.

Pair Programming

Pair programming: Code is written by two people; one writes the code, the other considers the effect of the code on the larger system.



One of the more controversial XP practices.

Pair programmers are peers!

Continuous Deployment

Continuous Deployment takes CI a step further: once a developer makes a commit, the process to deployment to the customer is fully automated.

This is natural extension to CI.

It has many benefits, but also some risks:

- Possibility of deploying buggy software
- Developer stress (every commit may lead to a "production system down" event)

Feature Flags

What if you commit code that only partially implements a feature. How do you stop this going to production?

You can hide the code behind a **feature flag**, e.g.

```
featureX = false;
...
if (featureX)
   "here is my incomplete code for X"
...
if (featureX)
   "more incomplete code for X"
```

`featureX` is set to `true` for development

In a real context, feature flags will be managed centrally.

Multiple feature flags can greatly increase code complexity.

Other uses for feature flags

Canary release: release a new version to a small number of users to see if it works.

Mimics the use of canaries to detect poisonous gases in coalmines.

A/B Testing: release a new version to a small number of users to see the new feature performs better than the previous one.

In A/B testing, the new version may provide a new feature, or may simply have a button in a different place in the UI.

In case you feel bad for the canaries...

Typical DevOps Tooling

Source Code Management: Git, GitHub, Bitbucket.

Build: Gradle, Maven, Ant

CI/CD: Jenkins, Bamboo

Containerisation: Docker (for deploying microservices), Kubernetes (for managing containers at scale)

Infrastructure as code (configuration management): Puppet, Ansible, Chef

Monitoring: Nagios, Prometheus, Zabbix

Introduction to Modelling and UML

Comp 47480: Contemporary Software Development

Modelling

Models are used heavily in engineering.

Models may be physical, like a scale model of a house, road or bridge.

We're more interested in abstract models, as this is what we use in software.

We use models so much software, so we often don't see them anymore e.g.

```
def foo(i: Int): String
```

is a model of a function that takes an integer argument and returns a string. Details are omitted.

What is a Model?

A model of X captures some properties of X and omits other properties.

Here's a model of this room:

COMP B106	
Seating Capacity: 96	AV Equipment:
Seating type: Tables and Chairs (ALE)	Data Projection
	Computer
	Plasma Screen x8
	HDMI Cable Input Supplied
	Whiteboard
	Wireless Microphone
	Lecture Capture

What does it capture? What does it omit?

Models and Correctness

Models never complete, but may be useful.

> *All models are wrong, but some models are useful. So the question you need to ask is not "Is the model true?" (it never is) but "Is the model good enough for this particular application?"* – George Box

We'll be looking at models in software, but keep this aspect of models in mind.

Models and Software Development

Modelling is a huge part of the Waterfall process.

The Requirements, Analysis and Design phases are all about building more and more detailed models of the software to be built.

Modelling is also part of Agile, but in a different was as we'll see.

What is UML?

UML stands for **U**nified **M**odeling **L**anguage.

UNIFIED MODELING LANGUAGE ™

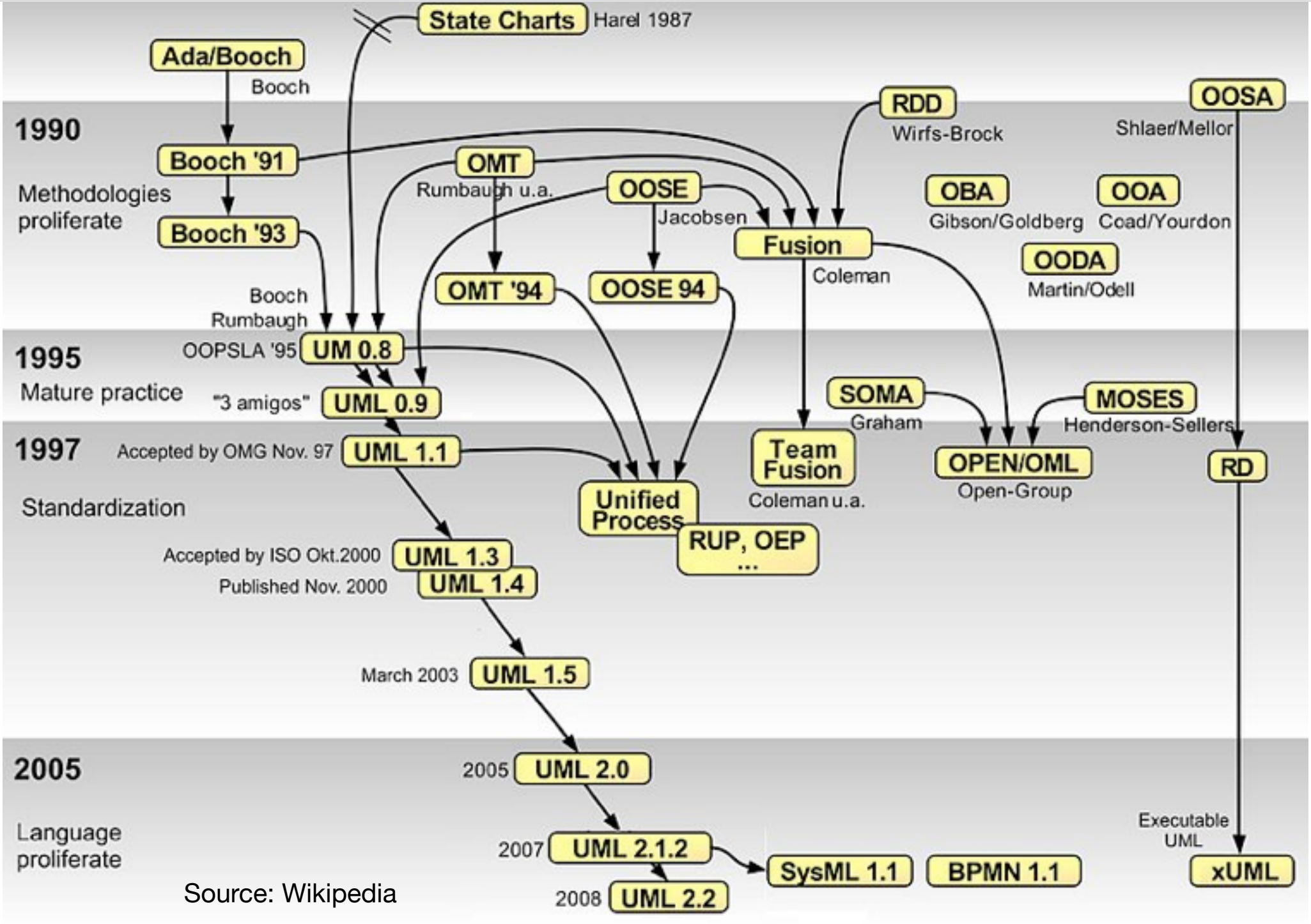
UML is a general-purpose modelling language used to create visual models of object-oriented software systems.

UML can be used to model:

- system structure
- system behaviour
- system architecture

and more.

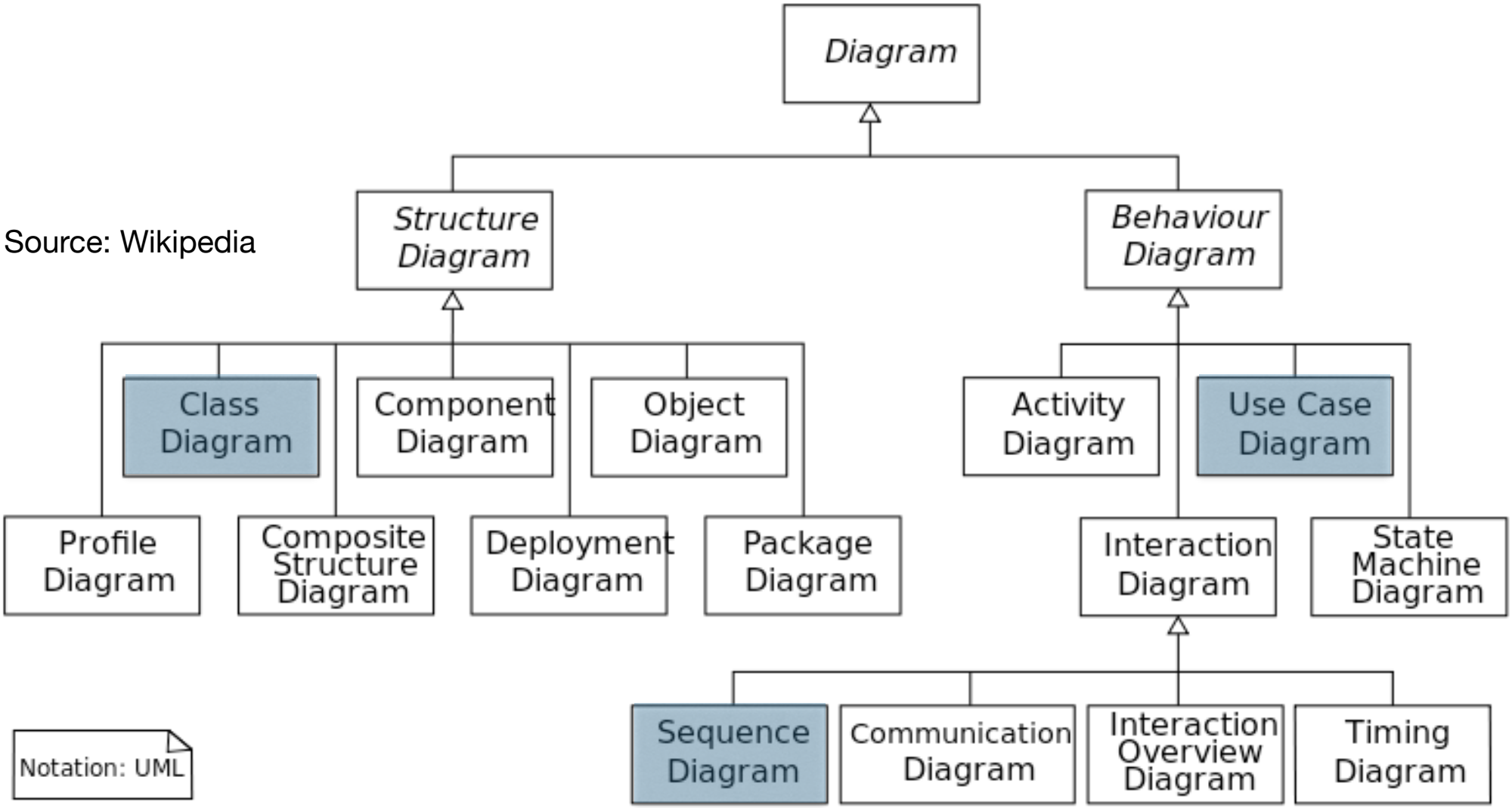
Where did UML come from?



Source: Wikipedia

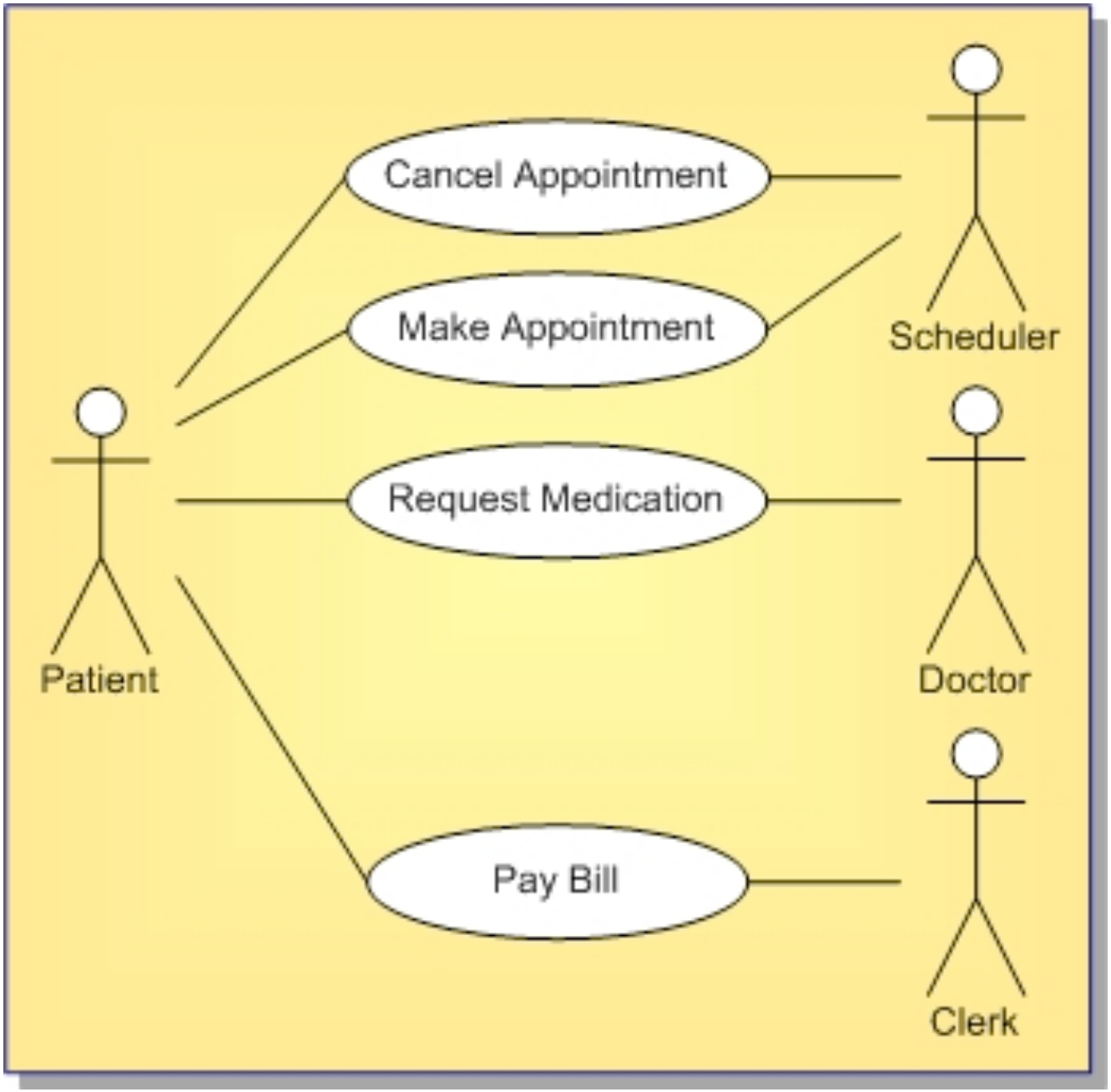
UML Diagram Types

Source: Wikipedia



14 diagram types in total! We'll look only at the three shaded diagram types.

Sample UML Use Case Diagram



Source: Embarcadero Technologies

Use Case or User Story?

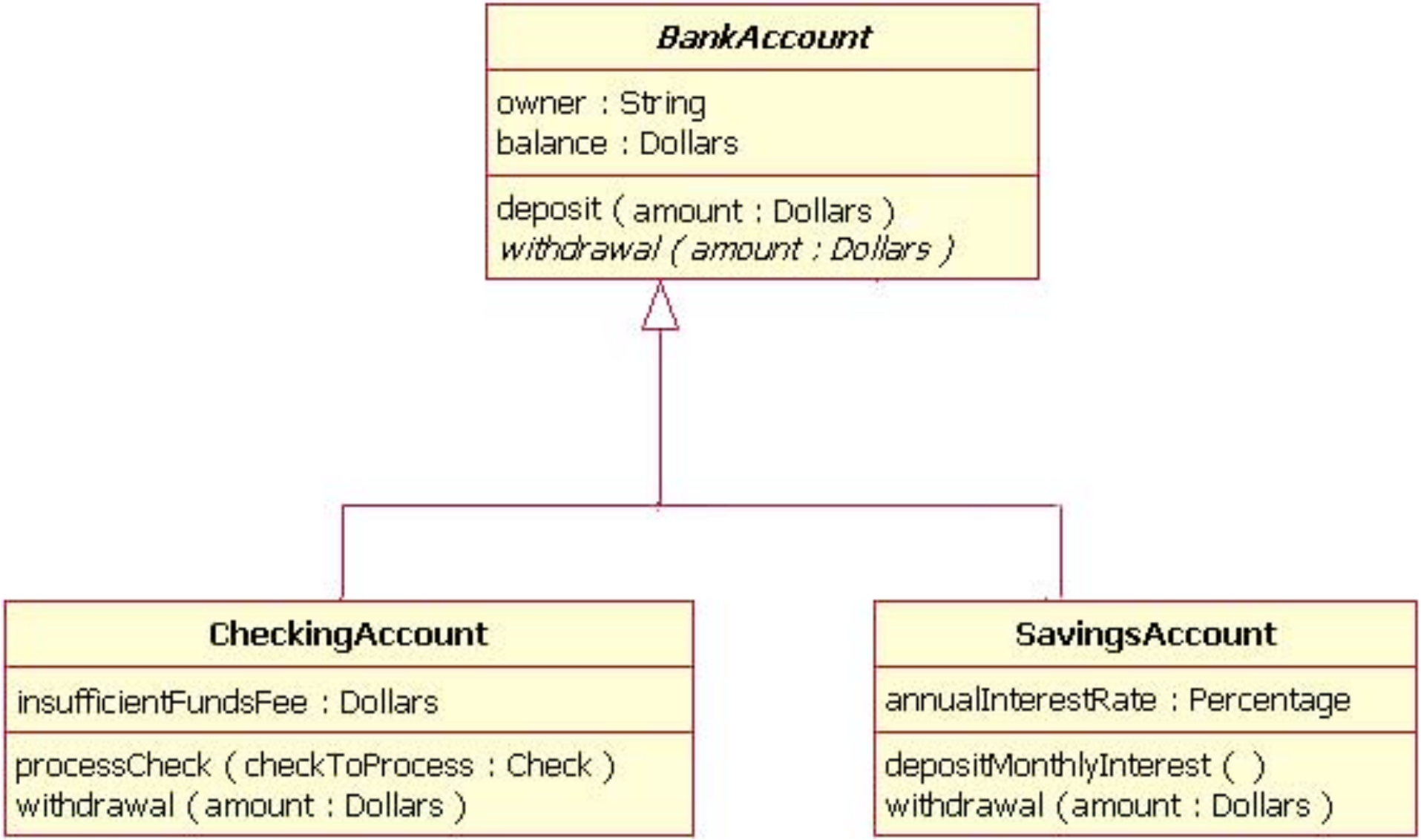
There is no exact definition that distinguishes between the two.

Both describe units of functionality from the end user perspective.

Use Cases tend to be higher-level blocks of functionality, and are usually documented more formally. Associated with a Waterfall process.

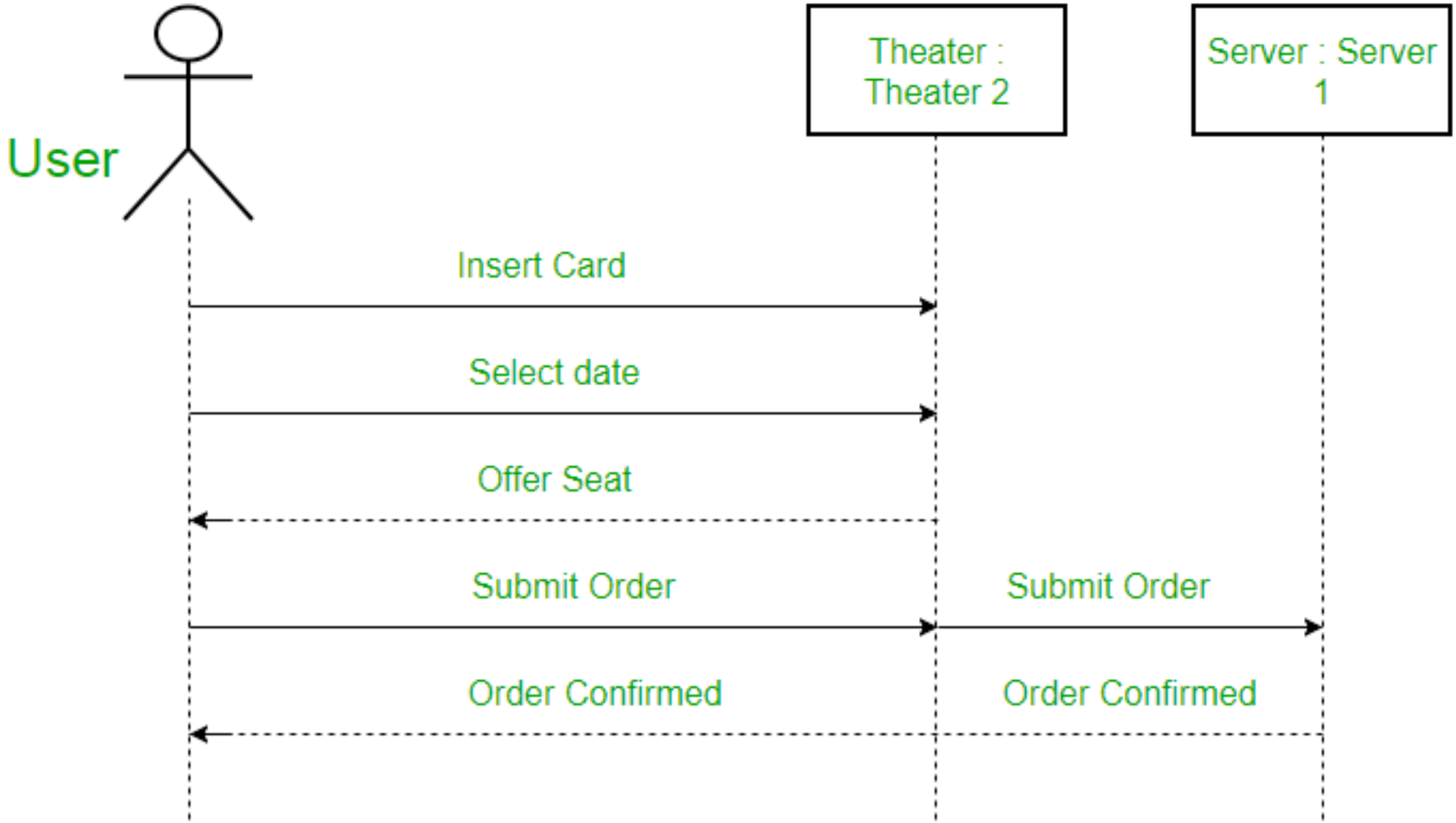
User Stories are minimal units of functionality. Associated with an Agile process.

Sample UML Class Diagram



Source: Rational

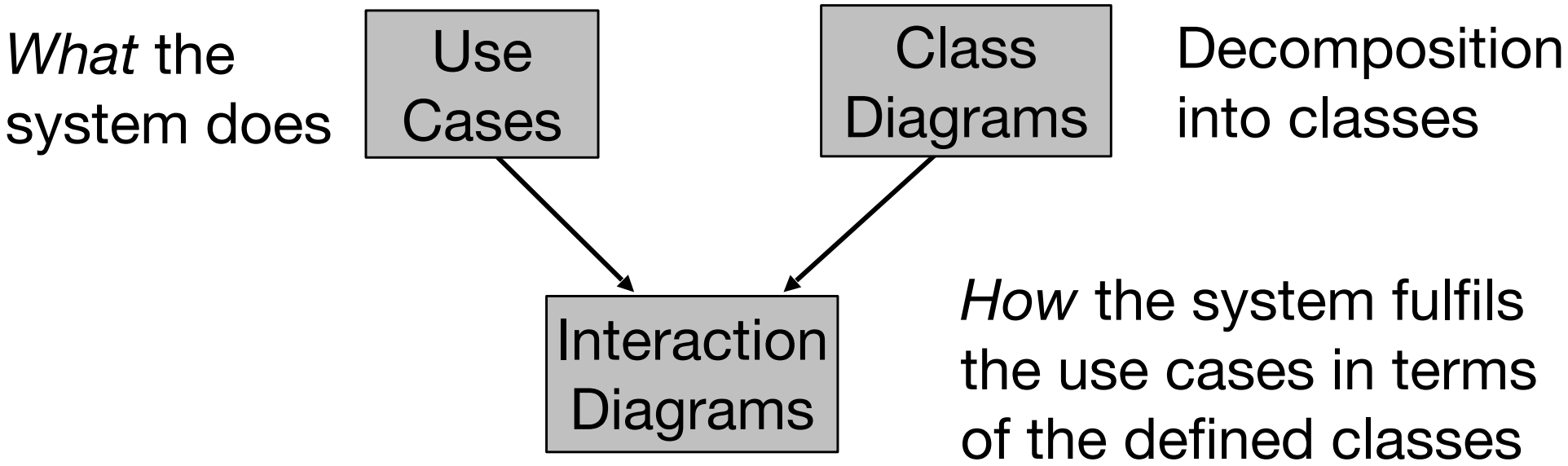
Sample UML Sequence Diagram



Source: geeksforgeeks

How are the diagrams related?

The relationship between the key diagrams we have seen can be visualised as follows:



Note: You would only ever build these models fully as part of a BDUF/Waterfall process.

UML and Agile

UML fits well with the Waterfall/BDUF approach -- it's the modelling language of choice for analysis & design.

Few teams use Waterfall anymore, so what is the role of UML in an Agile context?

(Recall the Agile manifesto: "**working software over comprehensive documentation**").

UML *is* used when discussing design options. So a basic familiarity with it is essential.

Roadmap

In the coming slides, we'll briefly look the three most commonly-used UML diagram types:

1. Use Case Model

2. Class Model

3. Interaction Model (Sequence Diagram)

Note for 2025: Will cover only the **Class Model** as it's the most relevant.

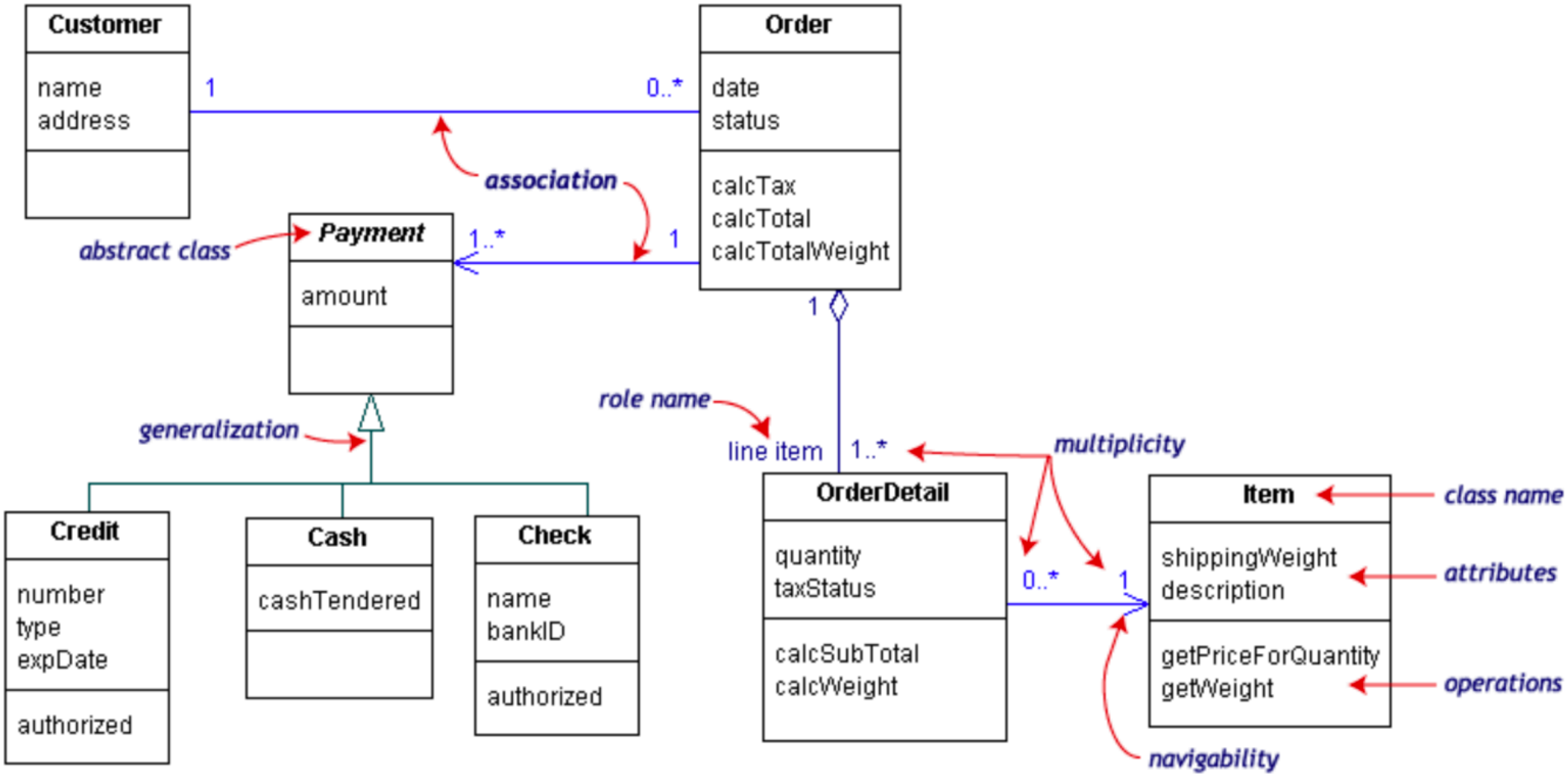
2. Class Diagrams

Class Diagrams depict the main classes and their associations.

High-level modelling is the goal.

Leave implementation details to the code.

Sample Class Diagram



Possible Cardinalities

Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, *n*	*m* or *n*
m..*n*	At least *m*, but not more than *n*
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

Domain Models

A **domain model** is a class model that shows only the key abstractions of the domain.

In low-level design, new implementation-related classes may be added to the class model.

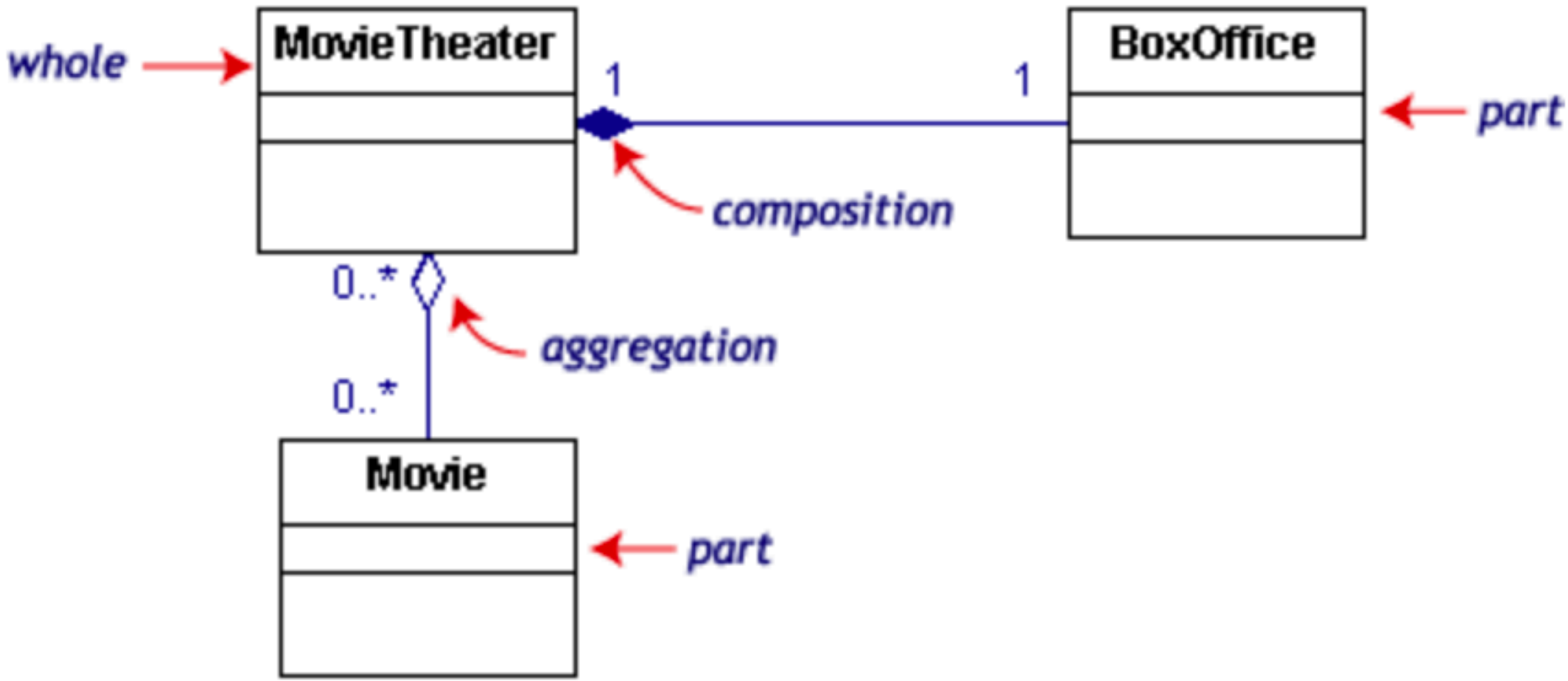
E.g. the earlier Customer/Order example is a domain model.

The implementation will require e.g. a List of Orders, but that is not part of the domain model.

Aggregation and Composition

Aggregation means a whole-part association.

Composition means full ownership. Deletion of the composite => deletion of components



Some Advice

“All models are wrong, some are useful.” — George Box

Modelling with UML can help you understand the problem better, or help you in developing a solution.

If you feel you could solve the problem more easily in code, just do it!

Modelling is pervasive in engineering (blueprints, drawings etc.), why don’t we do it more in software?

Empirical data on UML Usage (1)

Summary of a 2013 study of UML usage:

TABLE I. DECLARED CURRENT UML USE AMONG 50 PROFESSIONAL SOFTWARE DEVELOPERS FROM 50 COMPANIES.

Category of UML Use	Instances of Declared Current Use
no UML	35
retrofit	1
automated code generation	3
selective	11
wholehearted	0

It seems UML is not heavily used in practice.

Source: Petre, Marian, UML in practice, In: 35th International Conference on Software Engineering (ICSE 2013), 18-26 May 2013, San Francisco, pp. 722–731.

Empirical data on UML Usage (2)

From another 2014 study of ~400 software professionals:

"Informal" UML *is* used.

Sketches and diagrams mainly used for designing, explaining, and understanding.

Usually done on paper or whiteboard. Often archived.

Most sketches relate to methods, classes, or packages, but not to source code artefacts.

Source: Baltes and Diehl, Sketches and diagrams in practice, Proceedings of the International Symposium on Foundations of Software Engineering, 2014.

"Diagramming"

While the industry has turned away from UML, *diagramming* or *sketching* is gaining some interest.

The idea is to develop diagrams in specific situations where code is less effective, e.g.

- to visualise the system architecture
- to describe complex object interactions
- to visualise the effect of a multi-step refactoring

The Baltes and Diehl paper from the previous slide provides evidence to support this.

Notes: SE-Radio Episode 566 (May 2023) covers the use of diagramming in software development. Based on the book *Creating Software with Modern Diagramming Techniques*, Ashley Peacock, Pragmatic Bookshelf, 2023.

Diagramming Tooling

While a UML-like language is used, notation details are not important.

The main diagram types used are **class diagrams** and **sequence diagrams**.

Tools like PlantUML or Mermaid enable diagrams to be drawn using a markdown-like language.

- Easy to store alongside code
- Can be versioned easily in a Git repo
- Example on next slide

Mermaid Example

Mermaid | Code | Config | Auto sync | DOCS

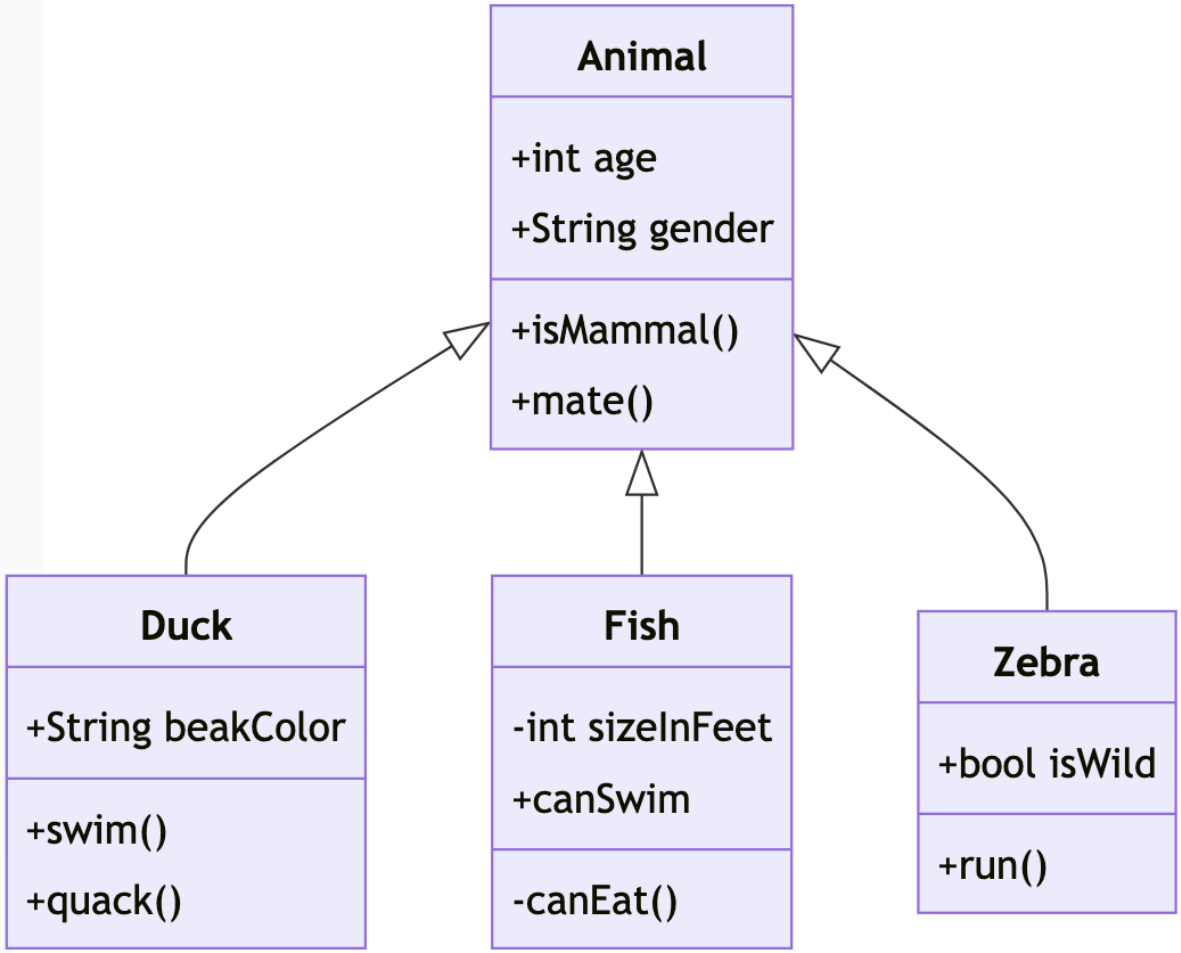
```
classDiagram
    Animal <|-- Duck
    Animal <|-- Fish
    Animal <|-- Zebra
    Animal : +int age
    Animal : +String gender
    Animal: +isMammal()
    Animal: +mate()
    class Duck{
      +String beakColor
      +swim()
      +quack()
    }
    class Fish{
      -int sizeInFeet
      -canEat()
      +canSwim
    }
    class Zebra{
      +bool isWild
      +run()
    }
```

> Sample Diagrams

> History

> Actions

Diagram | Pan & Zoom | FULL SCREEN | SAVE TO MERMAID CHART



Summary

UML is the standard object-oriented modelling language.

Although it is not used to drive development, developers are usually acquainted with it.

We looked at three key UML diagram types:
Use Cases: what the system does
Class Diagrams: key classes of the system
Sequence Diagrams: how objects interact

You should be able to do simple modelling using these diagram types.

Software Testing

Comp 47480: Contemporary Software Development

Roadmap of this Topic

- Introduction to Testing
- Black-box Testing
- White-box testing:
  - Statement coverage
  - Path coverage
  - Branch testing
  - Condition coverage
  - Cyclomatic Complexity
- Mocking
- Mutation Testing
- Test-Driven Development
- Summary

Why do we test software?

We test to evaluate properties of software, e.g.

Correctness

Reliability

Performance

Memory Usage

Security

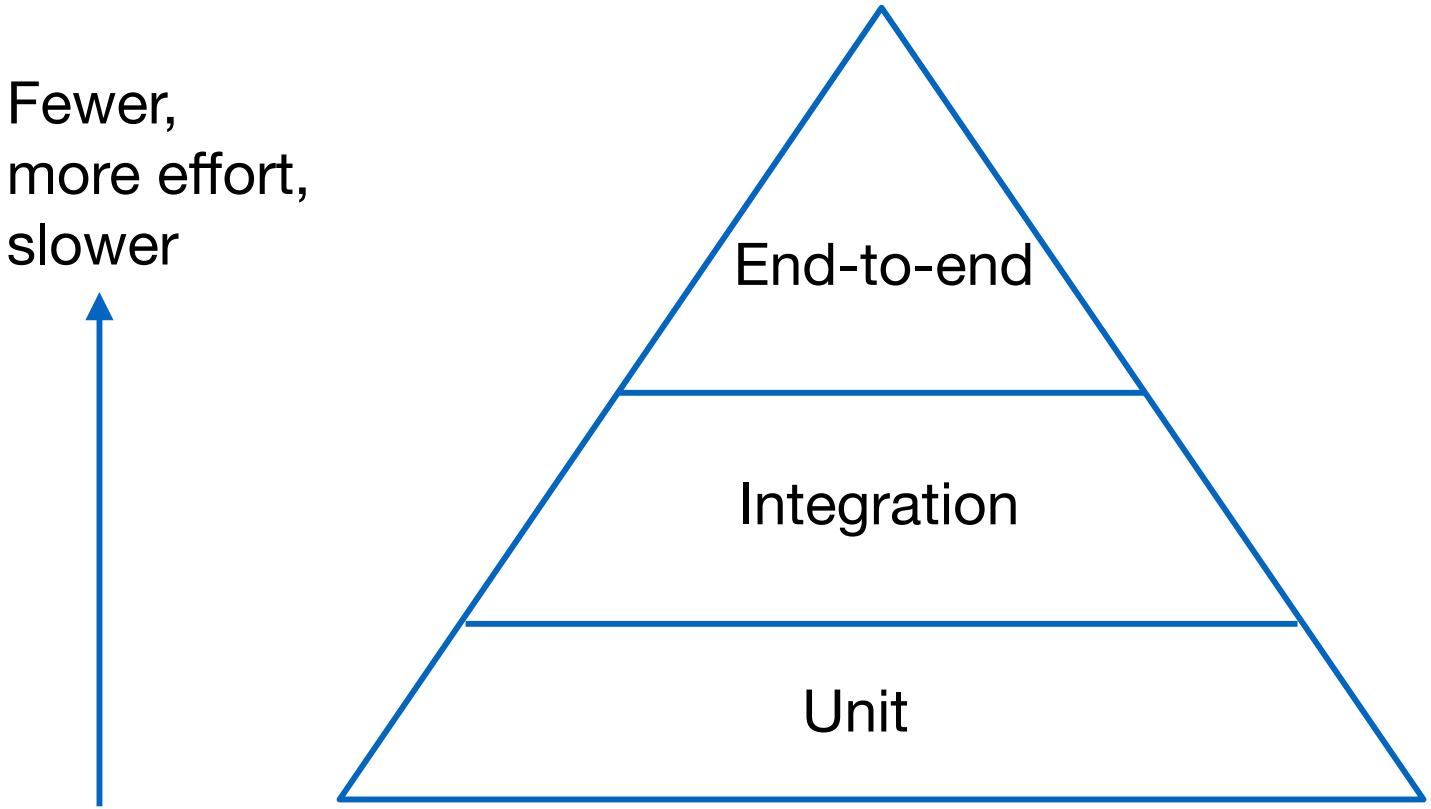
Usability

...

Our focus will be on testing for **correctness**.

"Code does what it should"

Testing Pyramid

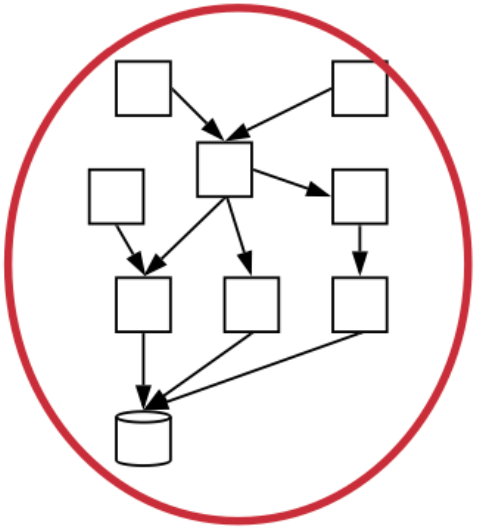


End-to-end testing: testing from UI to database.

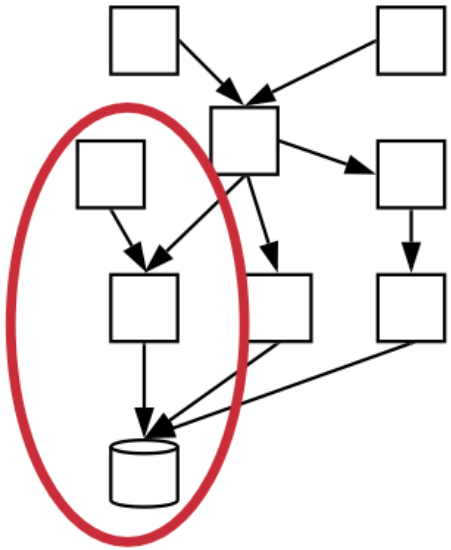
Integration testing: testing that components work together correctly.

Unit testing: testing that individual components work correctly.

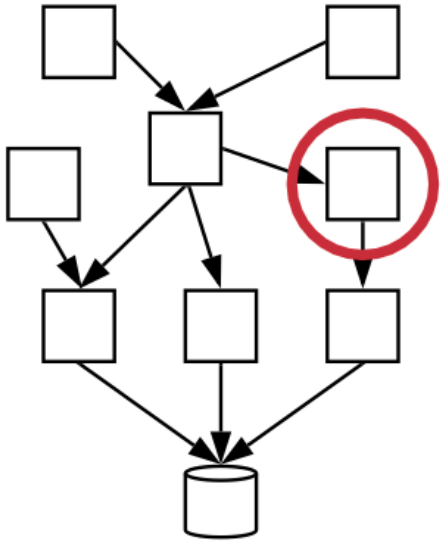
Testing Scope



End-to-end testing



Integration testing



Unit testing

Our focus in this module will be **unit testing**.

Recall too that we're focussed on non-critical systems -- bugs are undesirable but not catastrophic.

Other Types of Testing

Beta testing: testing in the real world prior to full release.

User acceptance testing: testing with real users.

Smoke testing: superficial test to ensure that basic functionality is correct.

Canary testing: releasing a new version of the software to a small group of users.

Black-box testing: testing based on a functional specification (aka *functional testing*)

e.g. integration testing

White-box testing: testing based on the structure of the software.

e.g. unit testing

Bugs and Failures

A **bug** (aka **defect**) is a mistake in the code.

A **failure** is a manifestation of an error (e.g. incorrect output or a crash).

Presence of a bug may or may not lead to a failure.

Test cases and Test suite

Software is tested using a set of carefully designed **test cases**. The set of all test cases is called the **test suite**.

A **test case** is a triplet [I, S, O]:

- I is the data to be input to the system,
- S is the state of the system in which the data is input, aka the test **fixture**
- O is the expected output from the system.

It simplifies testing greatly if S can be ignored.

Test case is a generic term. For this module, our test cases will invariably be **unit tests**.

Regression Testing

Does it ever make sense to create unit tests when we don’t know a priori what the outputs should be?

Regression in software is where updating the software introduces bugs into code that was working.

Regression testing involves testing if the software has regressed.

Creating a test suite even without knowing the correct output creates a 'snapshot' of expected system behaviour.

This test suite can be used for regression testing at a later stage.

Exhaustive Testing

Exhaustive testing is usually impossible as the input domain is simply too large.

E.g. how long would it take to test this method **on all possible inputs**? Assume 32-bit integers and a billion tests per second:

```
// Return GCD of x and y
int gcd(int x, int y) {
   ...
}
```

About 585 years

Desirable Features of a Test Suite

Fast: unit tests are run frequently.

Independent: Order of execution should not matter.

Repeatable: Unit tests should always provide the same result (ignoring **flaky tests** for now).

Self-checking: Unit tests should be able to check if they have passed, no developer intervention required.

Timely: Created by the developer as the production code is being written, not in a later test phase.

Also:

Fully-automated: Can be run with a single keystroke.
Self-contained: No external dependencies.

Test Suite Examples

Consider this method and two possible test suites for it:

```
// Return maximum of arguments
int maxInt(int x, int y) {
   ...
}
```

Test suite 1: `{(3,2), (2,3)}`

Test suite 2: `{(3,2), (4,3), (5,1)}`

Design of Test Cases

Now consider this buggy implementation and decide which test suite is better:

```
// Return maximum of arguments
int maxInt(int x, int y) {
  int max;
  if (y > x)
    max = y;
  max = x;
  return max;
}
```

Test suite 1: `{(3,2), (2,3)}`

Test suite 2: `{(3,2), (4,3), (5,1)}`

Design of Test Cases

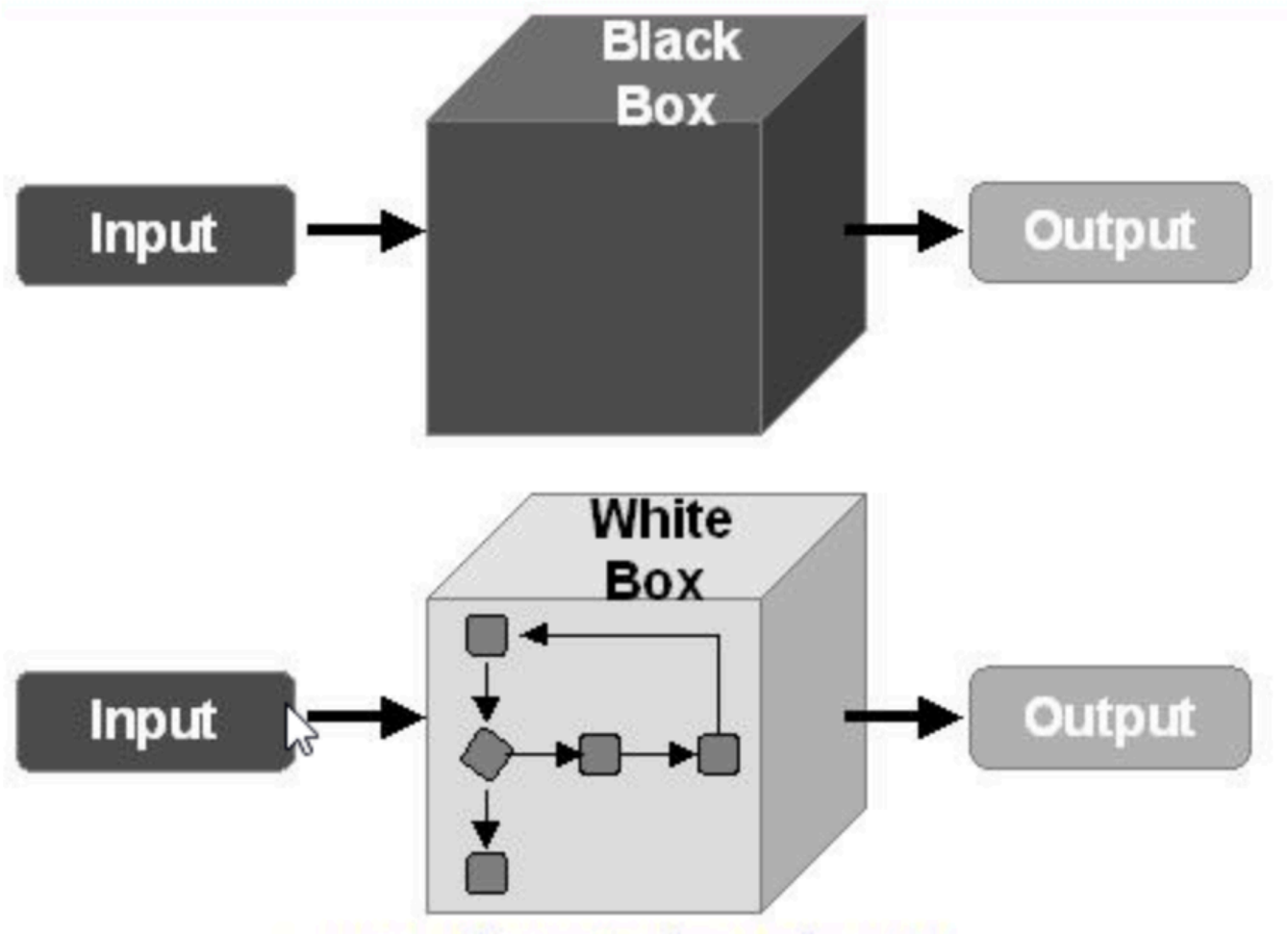
A systematic approach is required to design an effective test suite:

- each unit test should detect different errors.
- the tests should be "complete" in some way

- Two main approaches to designing unit tests:
  - **Black-box** approach
  - **White-box** approach

From here on, we assume we're testing a single, stateless **method**.

Black-box versus White-box Testing



Black-box testing relies solely on a functional description of the SUT.

White-box testing: based on the code structure of the SUT.

Black-box Testing

There are two basic approaches to designing black box test cases:

- **Equivalence class partitioning**
- **Boundary value analysis**

Equivalence class partitioning example

Consider a functional specification of a simple income tax system:

Salary	Tax
€0 to €20K	15% of total income
€20K to €50K	€3K + 25% of income over €20K
Above €50K	€5K + 40% of income over €50K

There are obviously three **equivalence classes**.

At least one case from each class should be tested.

Testing many cases from the same equivalence class is probably a waste of time.

Boundary value analysis example

Boundaries are areas where errors frequently lie:

- Iterating once too many or once too few
- Overrunning an array, or not processing the last element
- Not considering zero or negative values.

Many of these involve off-by-one bugs.

In the previous example, there are clearly three boundaries to be tested explicitly:

€0
€20,000
€50,000

Aside: Hard Problems in Computer Science

*There are only **two hard problems** in Computer Science:*

1. *Naming things*
2. *Cache Invalidation*
3. *Off-by-one errors*

Note: Quotation partly attributed to Philip Karlton

The zero/one/many rule

Where the SUT takes an aggregation (list, array etc.) as argument, it should be tested where:

The aggregation has **zero** elements.

The aggregation has **one** element.

The aggregation has **many** (>1) elements.

Obviously these cases may be processed differently in the SUT, and so should be tested separately.

White-Box Testing

What are we trying to achieve in white box testing?

Coverage. Making sure that the code is well **covered** by unit tests.

If some code isn't covered by one or more unit tests, no claim can be made about its correctness.

covered ~= executed by a unit test.

Code Coverage

In general, code that is “well covered” is better tested.

not covered => no correctness claim can be made

covered => improved confidence that code is correct

Many approaches to code coverage exist. We will look further at:

statement coverage
branch coverage
condition coverage
path coverage

We also look at a related technique: **mutation testing**.

Statement Coverage

We design test cases so that every statement in a method is executed at least once, e.g.

```
int foo(int x, int y) {
  int z = 0;
  if ((x>0) && (y>0)) {
    z = x;
  }
  return z;
}
```

Testing with {(1, 1)} => 100% statement coverage.

Statement Coverage Example

```
  int gcd(int x, int y) {
1.   while (x != y) {
2.     if (x > y)
3.       x = x - y;
4.     else
5.       y = y - x;
6.   }
7.   return x;
  }
```

Euclid's Greatest Common Denominator algorithm.

How can we achieve 100% statement coverage?

E.g. choosing the test set {(4, 3), (3, 4)} yields 100% statement coverage.

Rely on structure, not semantics!

```
  int gcd(int x, int y) {
1.  while (x != y) {
2.    if (x > y)
3.      x = x - y;
4.    else
5.      y = y - x;
6.  }
7.  return x;
  }
```

Don't try to exploit the semantics of the code to reduce the number of tests required -- it won't work in general.

E.g. {(4,3)} covers all statements, only because of the semantics of this particular example.

Dead Code

100% statement coverage is always achievable, unless the system has **dead code**.

Dead code can never be executed and can be safely deleted from the system. It may, or may not, be testable

Aside: dead code is remarkably common!

- e.g. a study of one PHP application found that 30% of the files contained **entirely dead code** (about 2,500 files)

Source: Boomsma et al., "Dead code elimination for web systems written in PHP: Lessons learned from an industry case", Proceedings of the 28th International Conference on Software Maintenance, pp. 511-515, 2012.

Tool Support -- JaCoCo

```
public class Stack<T> {
  private ArrayList<T> elements = new ArrayList<T>();
  private int size = 0;

  public int size() {
    return size;
  }

  public boolean isEmpty() {
    return (size == 0);
  }

  public void push(T elem) {
    elements.add(elem);
    size++;
  }

  public T pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException();
    T elem = elements.remove(size - 1);
    size--;
    return elem;
  }
}
```

All statements are covered (except) for the `throw` (red).

Tool Support -- JaCoCo

```
public String foo(){
  System.out.println("Working!");
  int x = 10;
  if (x > 10)
    System.out.println("10");
  if (x == 10)
    System.out.pri
  return "foo";
}
```

1 of 2 branches missed.

Yellow indicates partial coverage.

The first `if` statement is never executed where its condition is true, so its body is red.

The first `if` statement is only executed where its condition is true, so its body is green.

Neither `if` statement has **branch coverage**, so they are coloured yellow.

Recall the CI/CD Toolchain

JaCoCo is the sort of tool you might add to your CI/CD toolchain.

As part of preparing software for deployment, its test coverage is evaluated.

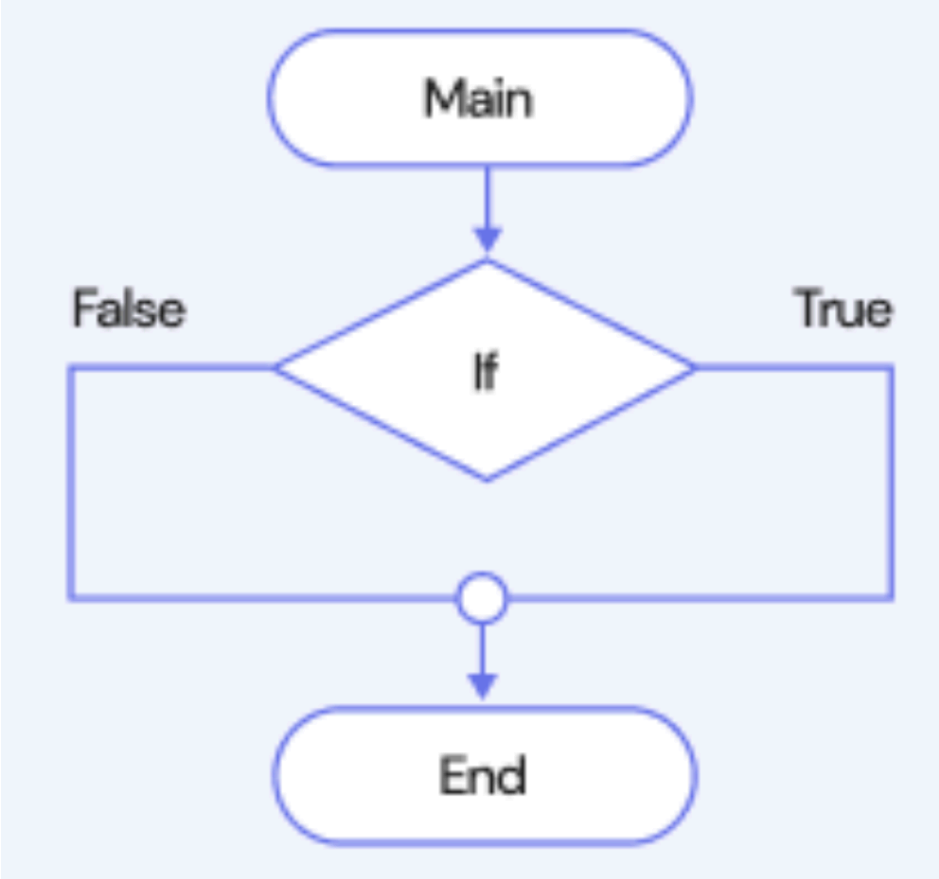
If coverage is poor, the developer can be alerted.

If coverage is very poor (e.g. <50%), perhaps the deployment should be rejected.

- Suggests that not all the units test were found and executed.

Branch Coverage (aka Decision Coverage)

=> testing each branch of each control structure,
e.g. `if`, `case`, `while` statements



Test all branch decisions with both true and false values.

Branch versus Statement Coverage

You can have statement coverage but not branch coverage, e.g.

```
int foo(int x, int y) {
  int z = 0;
  if ((x>0) && (y>0)) {
    z = x;
  }
  return z;
}
```

{(2,2)} gives statement coverage but not branch coverage

However

branch coverage => statement coverage

Branch Coverage Example

```
  int gcd(int x, int y) {
1.  while (x != y) {
2.    if (x > y)
3.      x = x - y;
4.    else
5.      y = y - x;
6.  }
7.  return x;
  }
```

How can we achieve 100% branch coverage?

Possible test suite that gives **branch coverage** would be:
{(3,3), (3,2), (3,4)}

Conditions and Decisions

A **condition** is a boolean expression that cannot be decomposed into a simpler boolean expression, e.g. (where **`a`** is boolean):

```
true
x == 10
b
```

A **decision** is a compound of conditions with zero or more Boolean operators, e.g.

```
a && b
(a || b) && (x == 35)
```

These are **decisions**, but not conditions

Basic Condition Coverage

In **basic condition coverage** each condition of a decision is tested with both true and false values.

So for this code fragment:

```
if (A && B) {
    ...
}
```

Basic condition coverage can be achieved with two unit tests.

Calculating Basic Condition Coverage

Basic condition coverage is usually expressed as the percentage of tested conditions, e.g.

```
if (A && B || C) {
   ...
}
while (X || Y && Z) {
    ...
}
```

6 conditions => 12 cases to be tested, so if e.g. the unit tests cover 4, we have 4/12 => 33% condition coverage.

(For simplicity, we ignore **lazy evaluation**, e.g. for `A && B`, `B` will not be evaluated if `A` is false.)

Does Basic Condition Coverage => Branch Coverage?

Unfortunately not.

Consider this simple case

```
if (A && B) {
    ...
 }
```

Testing with

```
A == true,  B == false
A == false, B == true
```

yields basic condition coverage, but not branch coverage.

Complete Condition Coverage

In **complete condition coverage** unit tests cover all possible combinations of boolean values

- requires 2^n test cases for n conditions
- very demanding criterion

So for this code fragment:

```
if (A && B) {
    ...
 }
```

Four test cases are required to achieve complete condition coverage.

Condition Coverage

Basic condition coverage is easy to achieve, but does not even guarantee branch coverage.

Complete condition coverage guarantees branch coverage, but requires many test cases.

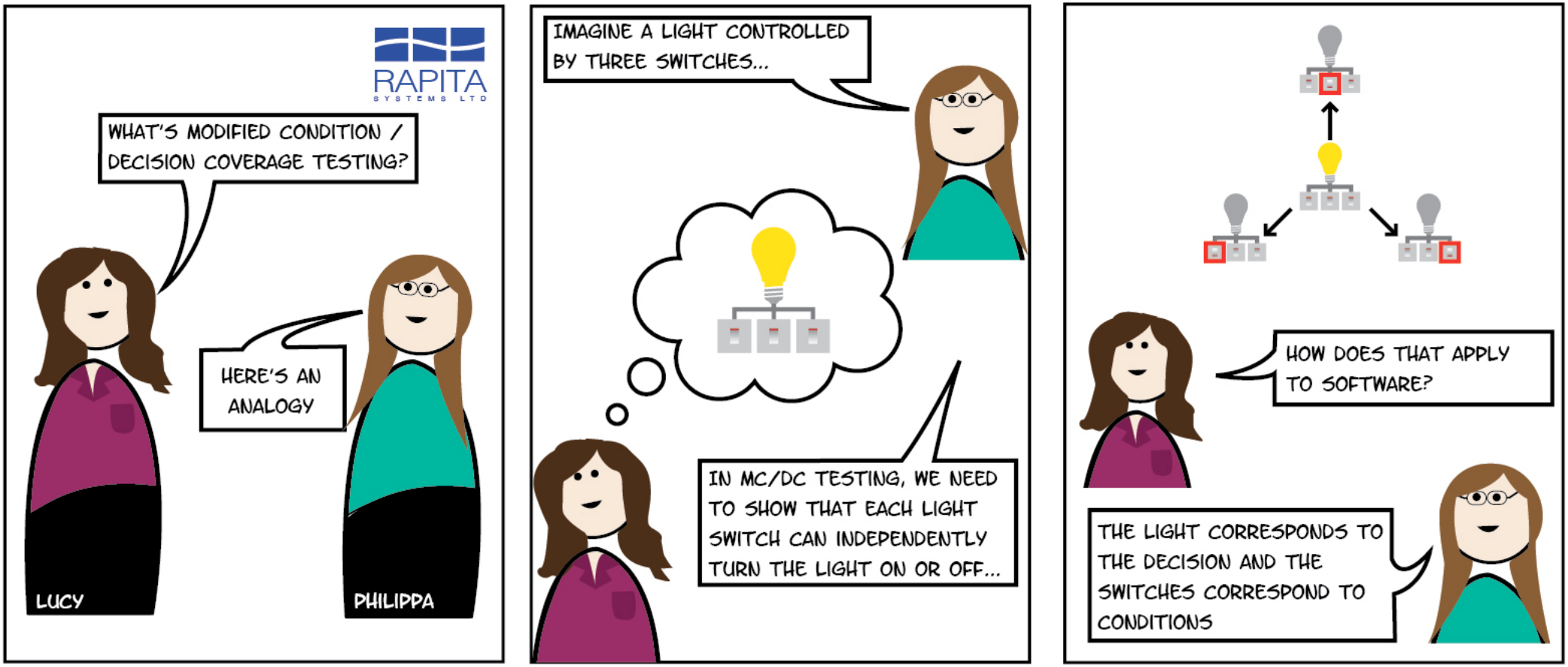
This led to the development of a condition coverage criterion that is less demanding than complete condition coverage, but guarantees branch coverage…

Modified Condition/Decision Coverage

- **MC/DC** is an “in between” technique, better than basic condition coverage but not as demanding as complete condition coverage.

- MC/DC coverage requires
  - basic condition coverage is satisfied, and
  - **each condition must affect the outcome independently, both when true and when false**

MC/DC Explained



www.rapitasystems.com

Each condition must be tested both being true and being false, in a context where its truth value determines the outcome of the overall decision.

MC/DC Example

- Consider the decision: `((a || b) && c)`

- **basic condition coverage** is achieved with two test cases, e.g.
  (a = true, b = true, c = true)
  (a = false, b = false c = false)

- **complete condition coverage** requires eight test cases

What testcases are sufficient
to yield MC/DC coverage?

MC/DC Example

What testcases are sufficient to yield MC/DC coverage?

a	b	c	(a||b)&&c	use?
FALSE	FALSE	FALSE	FALSE	
FALSE	FALSE	TRUE	FALSE	
FALSE	TRUE	FALSE	FALSE	
FALSE	TRUE	TRUE	TRUE	
TRUE	FALSE	FALSE	FALSE	
TRUE	FALSE	TRUE	TRUE	
TRUE	TRUE	FALSE	FALSE	
TRUE	TRUE	TRUE	TRUE	

MC/DC Example

Highlight the cases where toggling the condition (a, b or c) changes the decision ((a||b)&&c).

a	b	c	(a||b)&&c	use?
FALSE	FALSE	FALSE	FALSE	n
FALSE	FALSE	TRUE	FALSE	y
FALSE	TRUE	FALSE	FALSE	opt
FALSE	TRUE	TRUE	TRUE	y
TRUE	FALSE	FALSE	FALSE	n
TRUE	FALSE	TRUE	TRUE	y
TRUE	TRUE	FALSE	FALSE	opt
TRUE	TRUE	TRUE	TRUE	n

Here's one solution

a	b	c	(a||b)&&c
FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE

All conditions take on both true and false in situations where they affect the decision outcome independently (the shaded boxes).

Path Coverage

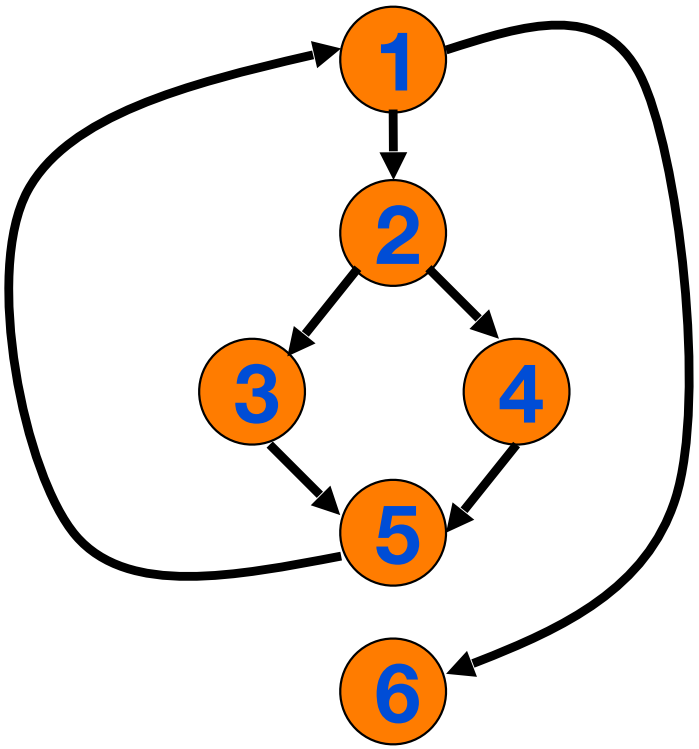
A **path** is a route that control may flow from the method entry point to an exit point.

Testing **all paths** through a method is desirable, but is usually impossible in the presence of loops.

To explain this further, we first look at what a **Control Flow Graph** is.

CFG Example

```
  int gcd(int x, int y){
1.  while (x != y) {
2.     if (x > y)
3.        x = x - y;
4.     else y = y - x;
5.  }
6.  return x;
  }
```

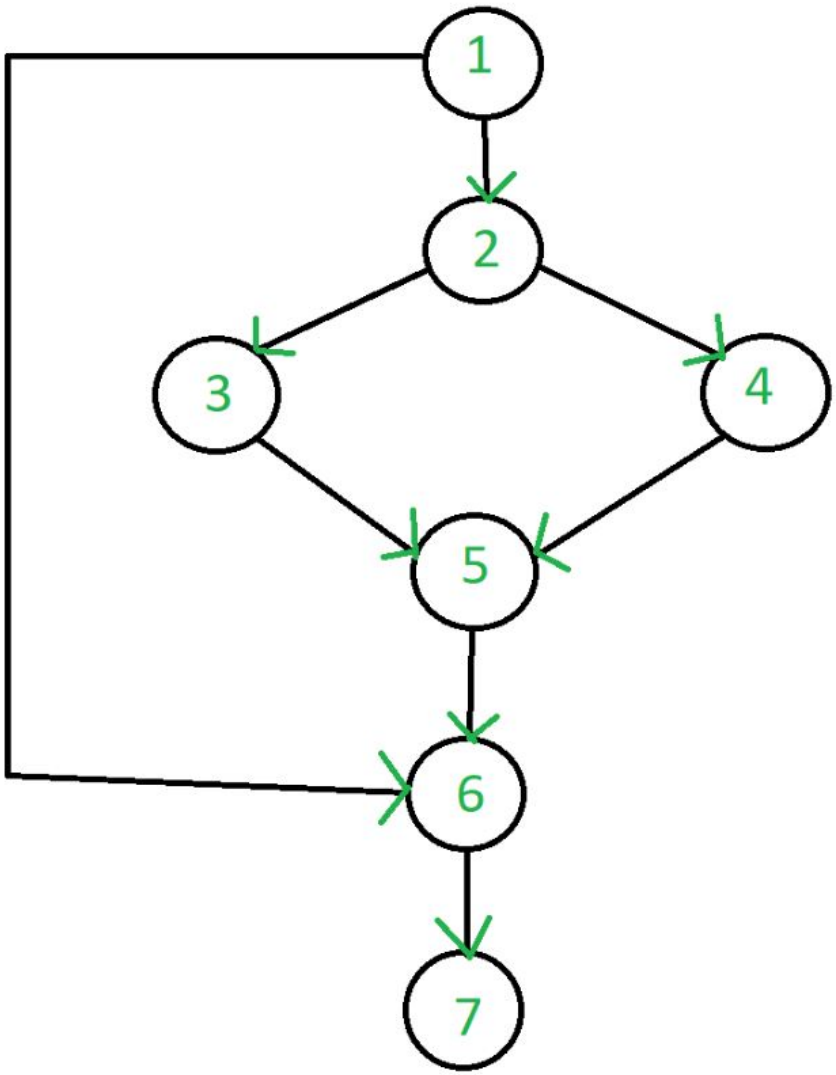


Note the CFG structure for decisions and loops.

Paths

- A **path** through a program/ method is a node and edge sequence from the starting node to a terminal node of the CFG.

- There may be several terminal nodes (`return` statements), but for simplicity we'll assume there's only one



Control Flow Graph

Basis Path Testing

Testing all paths is impractical/impossible but is there a useful subset of paths we can test?

In 1976, Thomas McCabe proposed a solution, something he termed **basis path testing**.

This idea still has considerable traction in the software industry.

For example, until ~2020 JaCoCo reported if the number of unit tests was sufficient for basis path testing.

Basis path testing in a nutshell

View the set of paths through a method as a set of vectors.

The elements of the vectors are the nodes in the CFG. 1 indicates the node is on the path, 0 means it is not.

Find a linearly independent set of these vectors (using e.g. Gaussian Elimination).

Map these linearly independent vectors back to paths through the CFG.

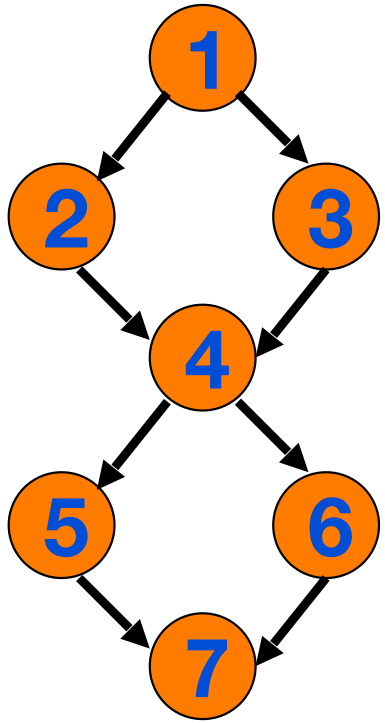
Create unit tests to traverse these paths.

The number of paths required is a popular metric, employed to measure the complexity of a method.

Linearly Independent Paths Example

	1	2	3	4	5	6	7
P1	1	1	0	1	1	0	1
P2	1	0	1	1	0	1	1
P3	1	1	0	1	0	1	1
P4	1	0	1	1	1	0	1

<=



The paths {P1, P2, P3, P4} are **not** linearly independent as e.g. P4 = P1 + P2 - P3

The paths {P1, P2, P3} **are** linearly independent.

McCabe's *Cyclomatic Complexity* Metric

The number of paths through a method required for basis path testing has become a popular way of measuring the complexity of the method.

For a CFG G, **Cyclomatic Complexity** is defined:

V(G) = E - N + 2,

where N is the number of nodes in G and E is the number of edges in G.

This is, very simply, the number of decisions in a method (`if`, `while` statements etc.) plus 1.

Cyclomatic Complexity as a metric

Cyclomatic Complexity (CC) is a commonly-used **software quality metric.**

Proponents claim code with a high CC score is:
- more error-prone
- harder to comprehend

However, in spite of its popularity it's of dubious merit.

Theoretical Issues with CC

Nesting is not considered, even though e.g. nested loop are clearly a greater cognitive burden than sequential loops.

`else` clauses do not contribute to CC

`switch` statements contribute disproportionately to CC (example next slide)

statement sequences do not contribute to CC, regardless of length

Source: Martin Shepperd, *A Critique of Cyclomatic Complexity as a Software Metric*, Software Engineering Journal, 1988.

CC and Case Analysis

This method is simple to understand, in spite of CC of **14**.

```
String getMonthName (int month) {
    switch (month) {
        case 0: return "January";
        case 1: return "February";
        case 2: return "March";
        case 3: return "April";
        case 4: return "May";
        case 5: return "June";
        case 6: return "July";
        case 7: return "August";
        case 8: return "September";
        case 9: return "October";
        case 10: return "November";
        case 11: return "December";
        default:
            throw new IllegalArgumentException();
    }
}
```

Source: https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity

Empirical Issues with CC

Theoretical issues can be ignored if a technique is found to be of practical value.

Several studies have investigated if CC correlates with bug rate, with inconclusive results:

- Some correlation with bug rate is apparent
- Correlation between CC and LOCs very strong
- Correlation with bug *density* not apparent

So the only claim supported by empirical studies is "larger programs have more bugs" which is unremarkable of course.

How is CC used?

Some companies set a threshold for method complexity, e.g. VisualStudio documentation promotes a maximum value of 10.

More commonly (and sensibly), CC is used only as a *guide*.

E.g. some teams in Microsoft, Dublin use CC to detect outliers.

The industry-standard software quality tool, SonarQube, reports on CC values.

(For some explanation of the popularity of this maligned metric, see *Cyclomatic Complexity* (invited content), IEEE Software, Nov/Dec 2016.)

Back to Testing and Coverage...

We want our test suite to detect bugs.

Coverage (statement, branch, path) is only a *proxy measure*.

Statement coverage is essential to make any claim about correctness.

However it is simply not clear what stronger forms of coverage really mean in terms of bug detection.

How can we we better evaluate if our tests are good at bug detection??

Mutation Testing

Mutation Testing is a technique to make a test suite better at catching bugs.

In a nutshell, mutation testing involves:

1. Create a copy of the method you are testing
2. *Mutate* this new method (e.g. invert a decision)
3. Run the unit tests for the method under test
4. If they fail, all is good! Your suite found the bug.
5. If they pass, further exploration is needed.

Multiple mutants are created for each method.

Original GCD Method

```
public class GCD {
  public static int gcd(int x, int y) {
    while (x != y) {
      if (x > y)
        x = x - y;
      else
        y = y - x;
    }
    return x;
  }
}
```

... and one possible Mutant

```
public class GCD {
  public static int gcd(int x, int y) {
    while (x != y) {
      if (x >= y)
        x = x - y;
      else
        y = y - x;
    }
    return x;
  }
}
```

This is the mutation. ">" has been changed to ">=".

Mutation Testing -- terminology

Each mutated program is called a **mutant**.

If a mutant causes the tests to fail, is it said to be **killed**.

A mutant that causes the tests to pass, is termed a **surviving mutant**.

A surviving mutant that has the same behaviour as the original is termed an **equivalent mutant**.

A collection of mutants is termed a **horde**.

"Useless" Mutants

Many mutants won't help improve test cases. Some examples:

Replacing division with subtraction, but in a logging statement.

`log.Infof("Found %d (%.2f %%)!", e, float64(e)*100.0 / total)`	`log.Infof("Found %d (%.2f %%)!", e, float64(e)*100.0 - total)`

Replacing *greater than with less than* when comparing length of a collection to zero.

`showCart := len(cart.GetItems()) > 0`	`showCart := len(cart.GetItems()) < 0`

Changing a tuning parameter.

`slo = (20 * time.Second)`	`slo = (20 * time.Second) + 1`

Note: Thanks to Goran Petrović for these examples.

Equivalent Mutants

Equivalent mutants can be ignored, as they are simply different implementations of the original method, e.g.

```
int compareTo(Object o) {
  ...
  if (...) return +1;
  else return -1;
  }
  ...
}
```

original program

=>

```
int compareTo(Object o) {
  ...
  if (...) return +2;
  else return -1;
  }
  ...
}
```

mutant

Deciding that a mutant is equivalent is theoretically impossible in general. In practice, tool support can help.

One study found 40% of surviving mutants to be equivalent mutants! It's a serious challenge.

Surviving, Non-equivalent Mutants

Surviving, non-equivalent mutants are the interesting ones!

We've introduced a bug (the mutant), but the test suite failed to detect it. i.e. all unit tests passed.

Solution: Add a new unit test that kills this mutant.

The central thesis of mutation testing is that this new unit test makes the test suite better at catching real bugs.

Is this a valid assumption?

Does mutation testing improve testing?

A 2021 study performed in Google found:

1. Developers using mutation testing write more tests, and improve their test suites so that fewer mutants remain.

so what?

2. Mutants tend to be coupled with real faults.

interesting...

In (2), ~1500 real high-priority bugs were analysed to determine if mutation testing would have led to the creation of a unit test that would have caught the bug.

In 70% of cases it would have.

wow!

Source: Goran Petrović et al., "Does Mutation Testing Improve Testing Practices?," 43rd International Conference on Software Engineering (ICSE), 2021.

Mutation Testing Challenges

Even with tool support, this is a labour-intensive technique (surviving mutants have to be inspected).

Creating and running the mutants is automated, but time-consuming nevertheless.

Do the mutants represent the type of bugs that occur in practice? This is a key question; the Google study on the previous slides suggests yes.

Applying multiple mutations (higher-order mutation testing) aims to be more realistic.

Testing Summary

Unit testing is central in the development of reliable software.

Coverage is an easy way to evaluate a test suite, but the worth of this evaluation is unclear.

Cyclomatic Complexity has some use in determining how many unit tests are required, and is a weak measure of method complexity.

Mutation testing is more focussed on making the test suite better at fixing bugs.

Test-Driven Development

Mel Ó Cinnéide
School of Computer Science
University College Dublin

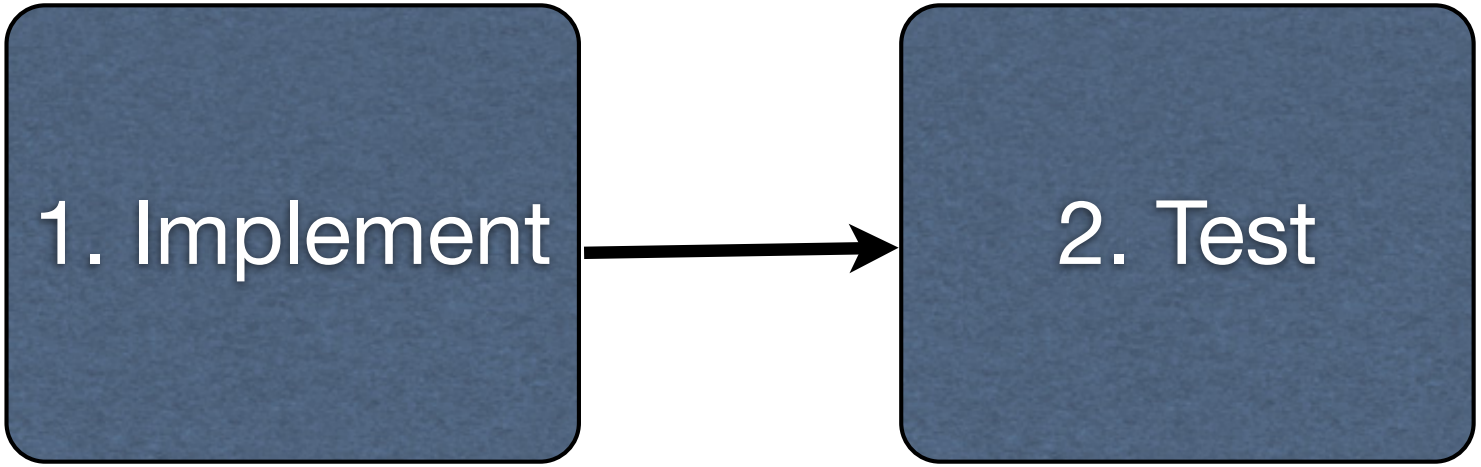
Test Driven Development (TDD)

The idea of TDD was created ("rediscovered" in his words) by Kent Beck and formed a key part of the **Extreme Programming** agile process.

TDD has since become a mainstream way of developing software, not just as part of XP.

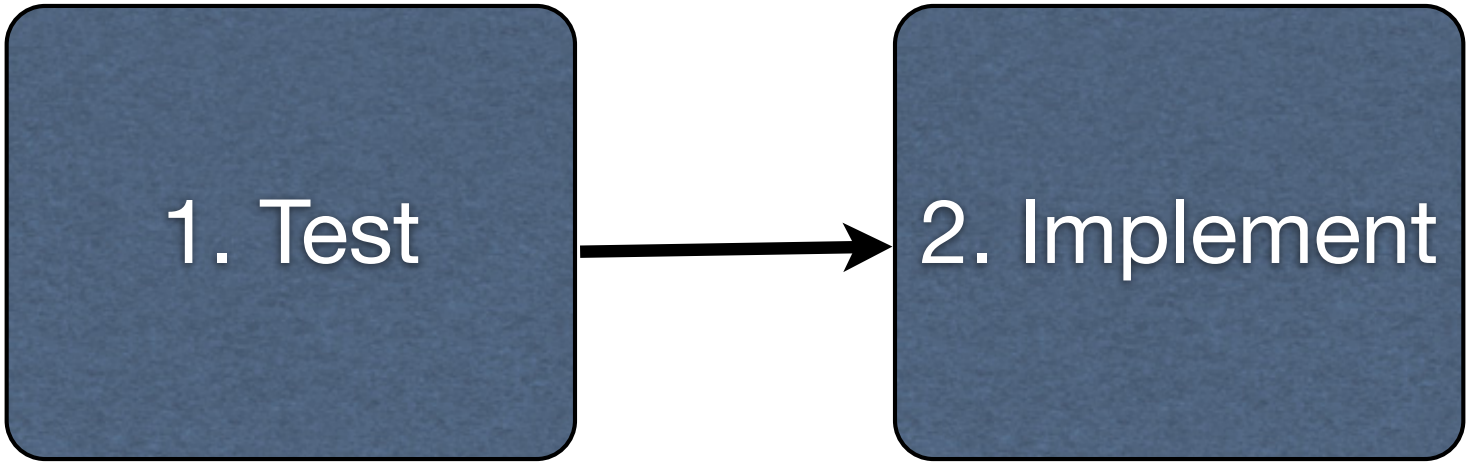
We'll explore at this idea in the coming slides...

“Traditional” Order of Implementation and Testing



Test-Driven Development (TDD)

Aka **Test-First** Development



Why other engineers don't use TDD

Why test first?

Firstly, note that unit tests are essential, so the TDD-or-not question is purely a matter of order.

A unit test states *what* the program should do. It's useful to think about this before deciding *how* it should do it.

Developers often write methods that they subsequently find hard to unit test. TDD avoids this.

TDD, applied well, guarantees excellent code coverage.

A User Story is like a test

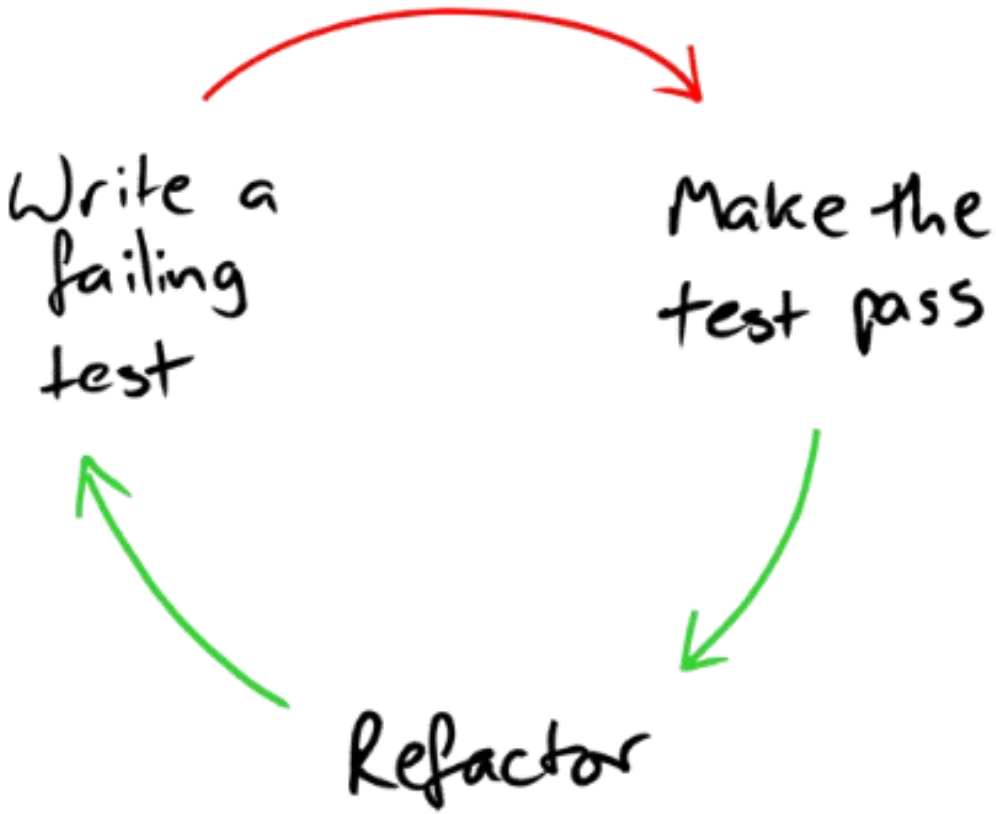
Any specification or user story implicitly provides tests that the program must pass, e.g.

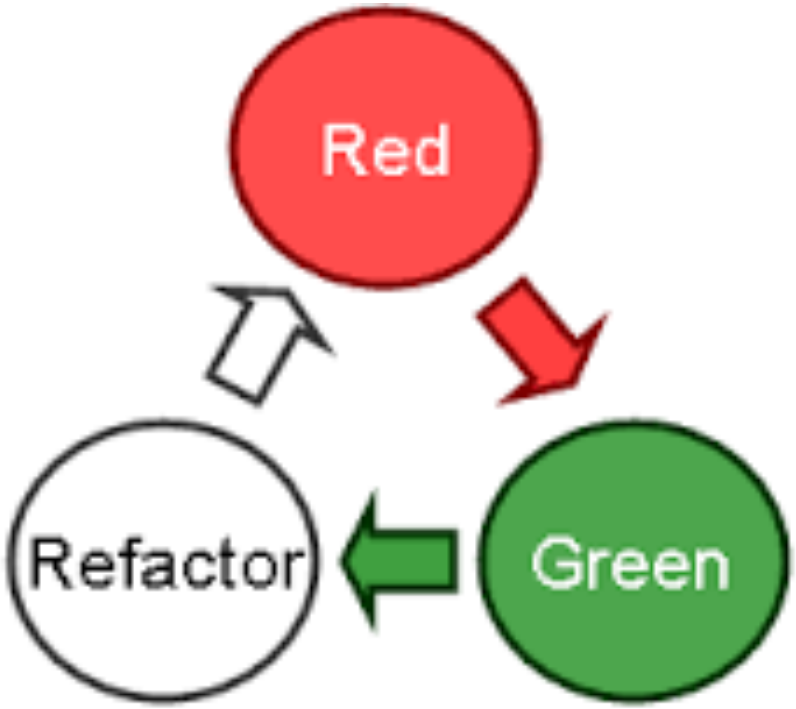
> *Write a method that accepts a array of integers and returns their average value, e.g. for the input [1,2,3,4,5]*
> *the value 3 will be returned.*

A user story may describe a whole range of input/output pairs. In this case it even provides a concrete example.

The key ingredient with test-first is that we use the unit tests to *drive* the development of the software.

Process of TDD





Red	Write a test that fails.
Green	Write code to pass the test.
Refactor	Clean up your code and the test.

Unit test structure

This is the typical structure for a unit test:

1. **Setup**: Put the system into the required state to execute the test.

2. **Execution**: Execute the method that is being tested and collect the results.

3. **Validation:** Ensure that the results of the test are as expected.

4. **Teardown:** Restore the system to its pre-test state.

For simple tests, **setup** and **teardown** may not be necessary.

JUnit

JUnit is a industrial-strength unit testing framework, originally written by Kent Beck and Erich Gamma.

It’s a large, quite complex framework that supports different styles of unit testing.

We’re only using the absolute basics of JUnit, so don’t get led astray when reading more about it online.

(I'm using JUnit 5, but JUnit 4 is fine to use in practicals if that's what you have installed.)

Erich Gamma: Best known as first author of the seminal *Design Patterns* text. Co-creator of JUnit, team lead for Eclipse Java Development Tools (JDT).

Anatomy of a JUnit 5 Unit Test

use the **JUnit** testing framework

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class FooBarTest {

  @Test
  public void testOne(){
    // execute unit under test...
    assertTrue(...);
  }

  @Test
  public void testTwo(){
    // execute unit under test...
    assertEquals(...);
  }
}
```

Convention: `FooBarTest` has unit tests for the `FooBar` class.

tells JUnit that this is a unit test

Convention: unit tests are named **`testXxx`**.

Test methods always contain **assertions**.

For more on JUnit, see http://junit.org

Summary of basic JUnit 5 assertions

assertEquals(expected, actual): Asserts that the expected and actual values are equal (also **assertNotEquals**).

assertTrue(condition): Asserts that the given condition is true (also **assertFalse**).

assertNull(value): Asserts that the given value is `null` (also **assertNotNull**).

assertSame(expected, actual): Asserts that expected and actual refer to the same object (also **assertNotSame**).

assertThrows(exceptionType, executable): Asserts that `executable` throws an exception of the specified type.

`@BeforeEach` and `@AfterEach` annotations

There may be some setup tasks that all test methods have in common; there may similarly be teardown tasks that they all require.

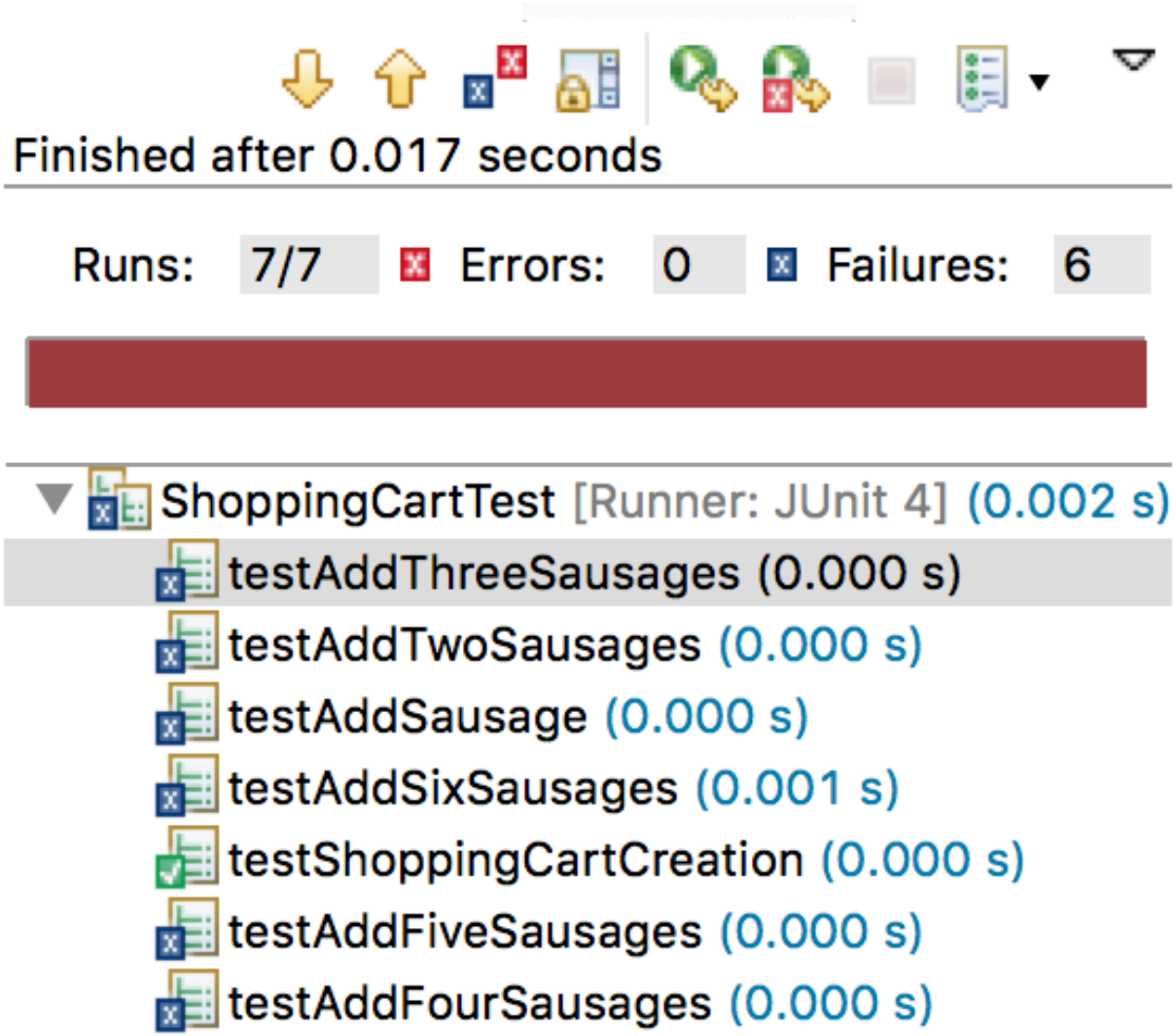
Methods with the **`@BeforeEach`** annotation are executed **before** each unit test:

```
@BeforeEach
void setUp(){
  myCart = new Cart();
}
```

Methods with the **`@AfterEach`** annotation are executed **after** each unit test:

```
@AfterEach
void tearDown(){
  myCart = null;
}
```

Write a Unit Tests that fails -- RED



Unit Tests should fail at first...

This provokes you to write code to make them pass.

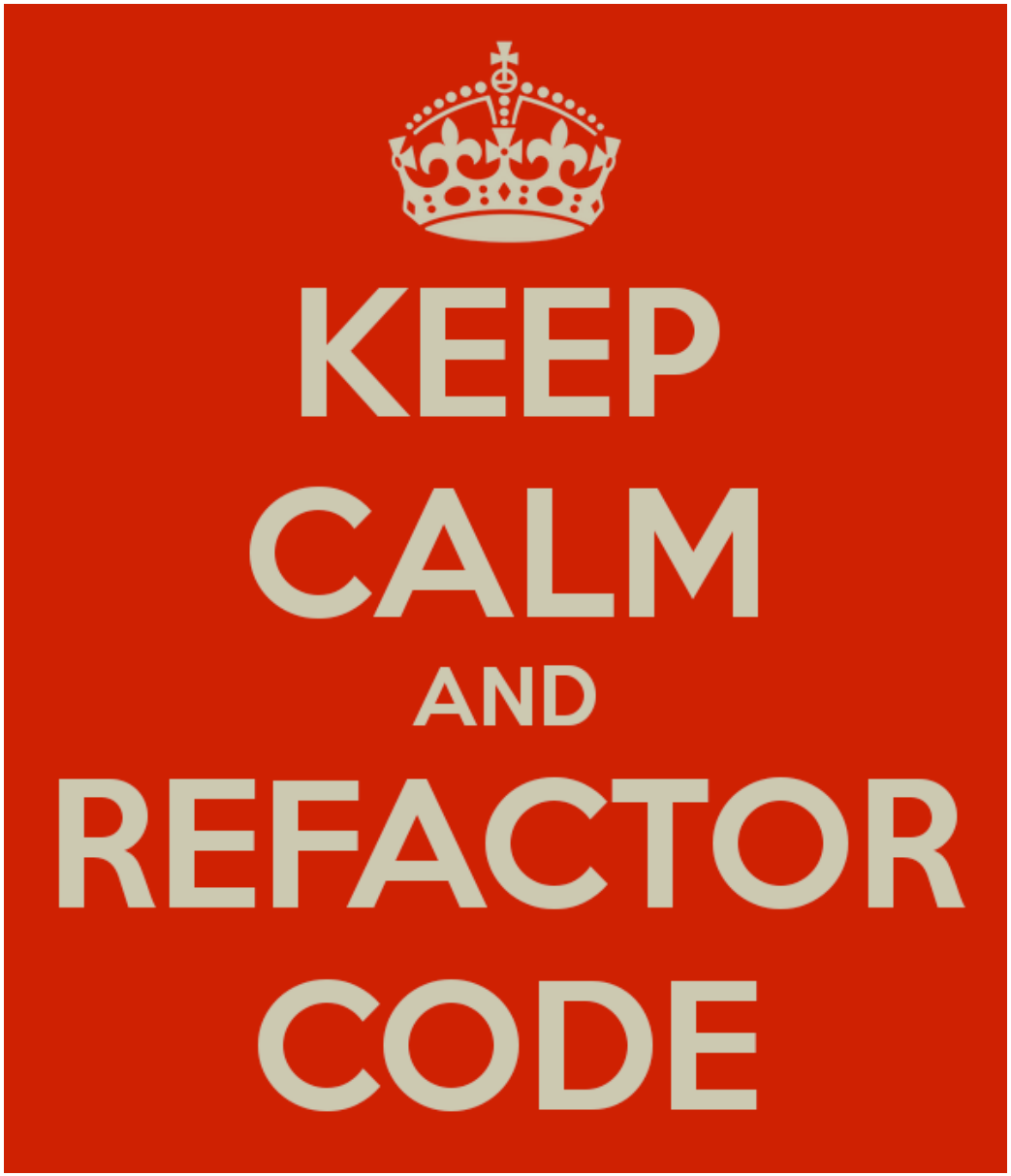
Write code to make the Unit Test pass -- GREEN

Runs: 7/7 Errors: 0 Failures: 0

ShoppingCartTest [Runner: JUnit 4] (0.001 s)

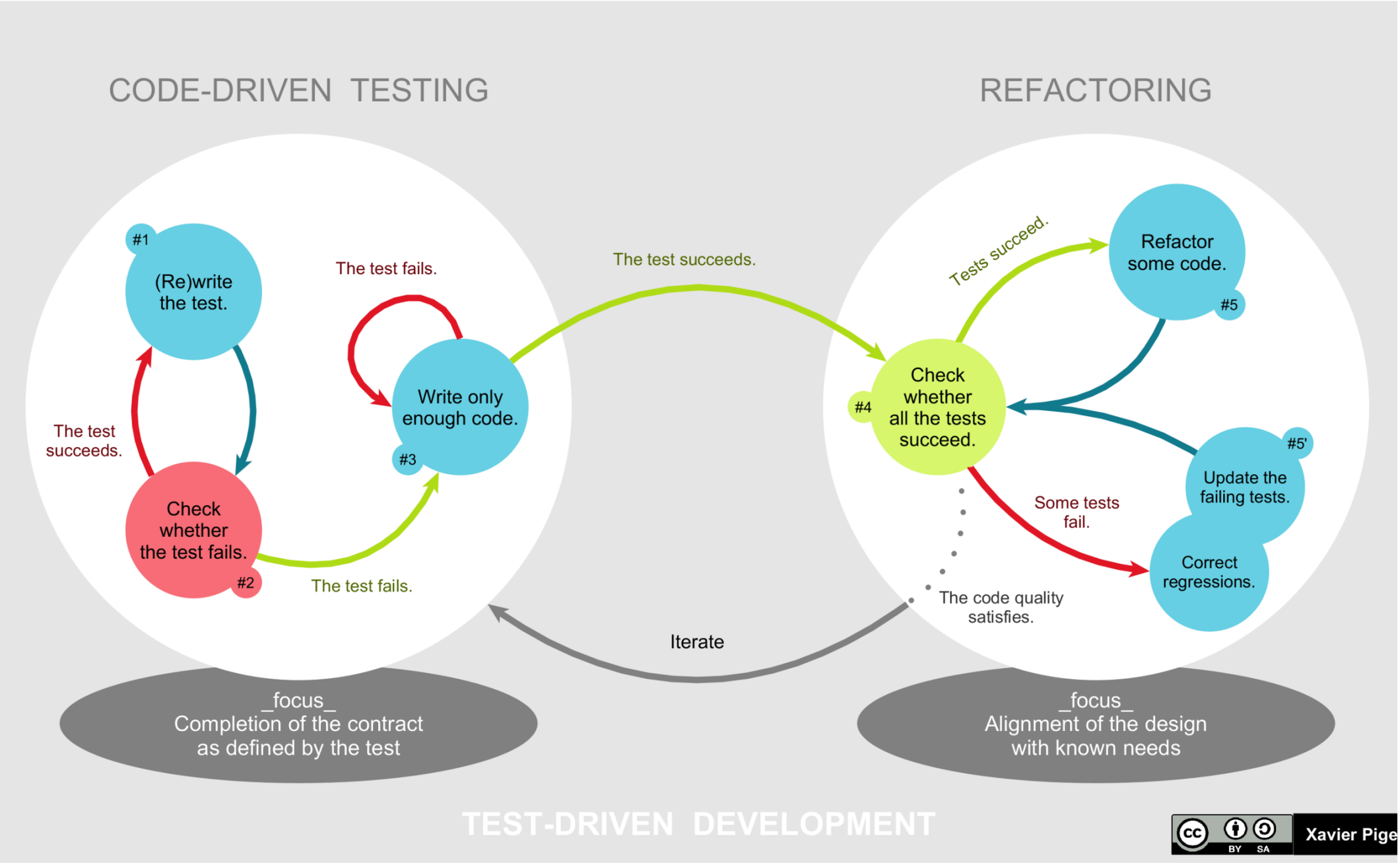
- testAddThreeSausages (0.000 s)
- testAddTwoSausages (0.000 s)
- testAddSausage (0.000 s)
- testAddSixSausages (0.000 s)
- testShoppingCartCreation (0.000 s)
- testAddFiveSausages (0.001 s)
- testAddFourSausages (0.000 s)

Now tidy up your code -- **REFACTOR**

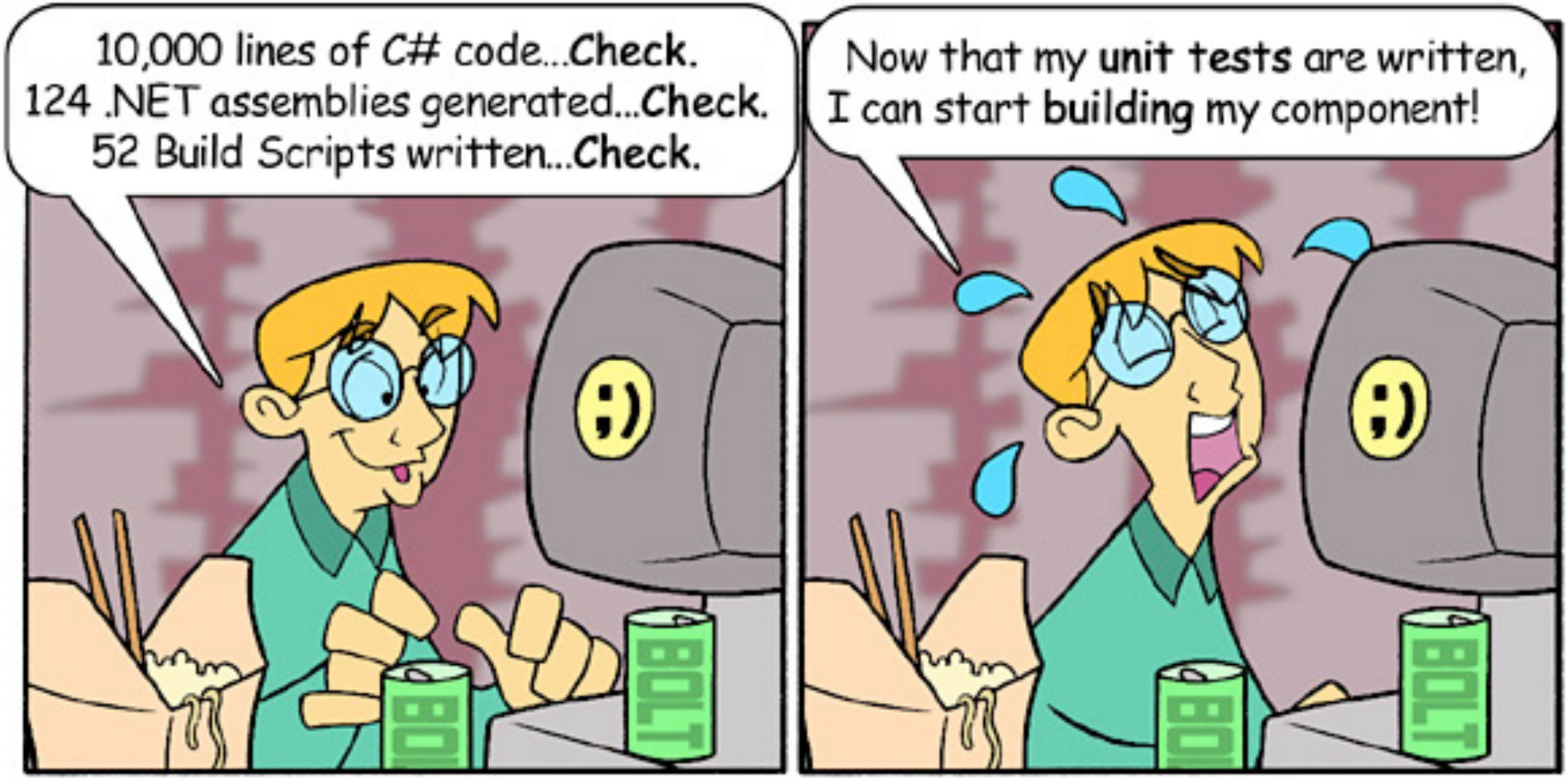


Tidy up your code, make sure there is no duplicated code or ugly bits. (We return to **refactoring** later in the module.)

An overview of the TDD process



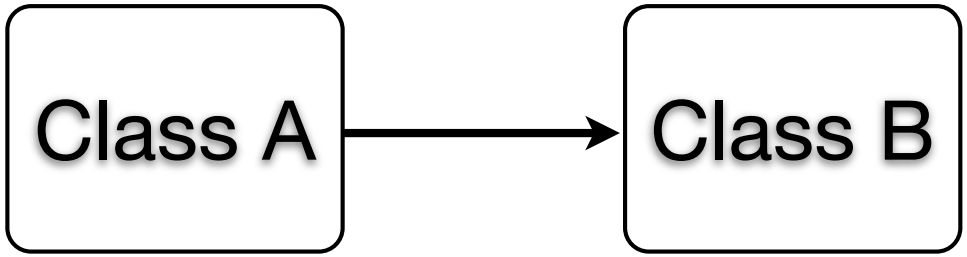
Test cases can take up a lot of code



... but this is not a bad thing.

Styles of TDD

Say we have this simple structure, where class A uses class B:

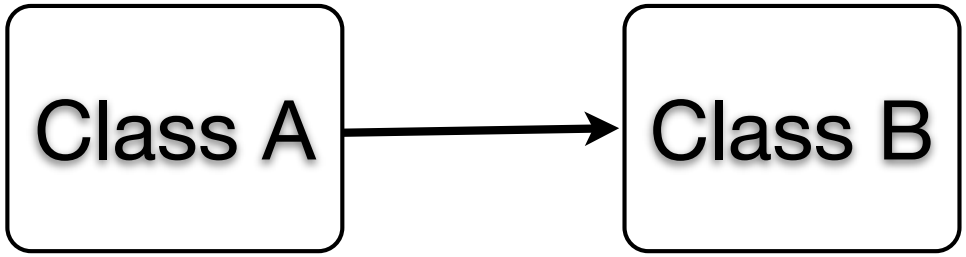


Which class do we test & develop first?

There are two main approaches:

Classicist (Detroit style, inside-out)
Mockist (London style, outside-in)

Classicist Approach

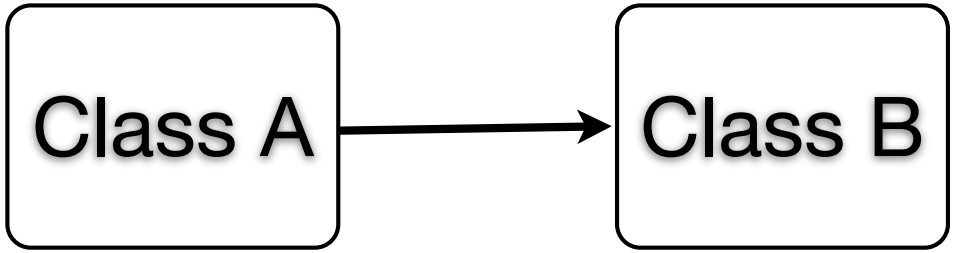


Apply TDD to Class B first.

Then apply TDD to Class A, relying on the correct implementation of B.

We typically use `asserts` to ensure that the A object is in the correct **state** at the end of the test.

Mockist Approach



Apply TDD to Class A first.

How can we test Class A, when we don't have a Class B?

We **mock** B.

A **mock object** mimics the behaviour of a real object in a controlled way.

In the mockist approach, we ensure that objects **interact** together correctly.

A Taste of Mocking

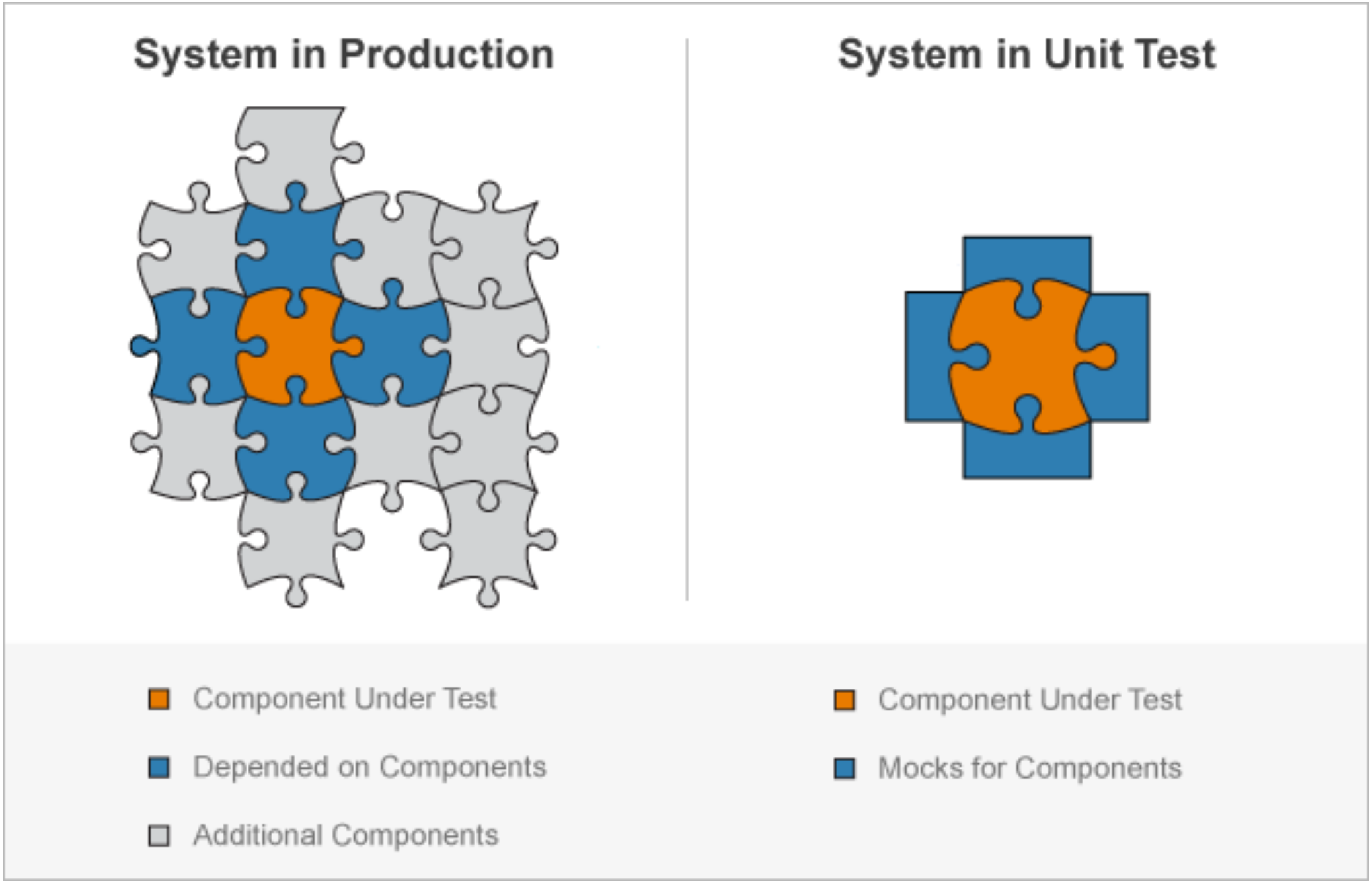
A mock object pretends to be a real one during a test.

It provides the same interface as the object it's mocking.

It can also check e.g. that it is sent the right messages, in the right order, with the right arguments.

So mocks can test **interaction**, not just state.

Mocking isolates the component under test



An Example

```
class Counter {
   private int count = 0;
   public void increment() {count++;}
   public int getCount() {return count;}
}
```

A simple class for maintaining a count.

```
class MonitorTemp {
   private Counter counter;

   public MonitorTemp(Counter counter) {
     this.counter = counter;
   }

  public void newTemp(int temp) {
    if (temp < 0) {
      counter.increment();
    }
  }

  public int countFreezing() {
    return counter.getCount();
  }
}
```

A class that uses the `Counter` class above.

TDD using the Classicist approach

1. Use TDD to develop the `Counter` class.

2. Use TDD to develop the `MonitorTemp` class, using the existing implementation of `Counter`.

The `MonitorTempTest` class will contain unit tests that test **state** like this:

```
monitor.newTemp(12);
monitor.newTemp(-1);
monitor.newTemp(1);
assertEquals(1, monitor.countFreezing);
```

Note how this is also testing the `Counter` class behind the scenes.

Pros and Cons of the Classicist approach

Cons

We're testing the `Counter` class unnecessarily. It already has its own unit tests.

If the `Counter` class is updated to a buggy implementation, the unit tests for both `Counter` and `MonitorTemp` will fail.

> (If 100 classes use `Counter`, the one bug will lead to 101 classes with failing tests.)

Pros

We can refactor fearlessly.

> E.g. we can move the counting functionality into the `MonitorTemp` class, delete the the `Counter` class and its tests, and all will be well.

TDD using the Mocking approach

1. Use TDD to develop the `MonitorTemp` class, using a **mock** implementation of `Counter`.

The real implementation of `Counter` is not used and can be developed afterwards (the 'outside in' approach).

We'll see how the `MonitorTempTest` class could be implemented using **Mockito** on the coming slides.

Mocks can also be developed by hand. A framework like **Mockito** makes the task easier.

Mocking the `Counter` class

The unit testing of `MonitorTemp` would look like this:

```
Counter counterMock = mock(Counter.class);
MonitorTemp monitor = new MonitorTemp(counterMock);
monitor.newTemp(12);
monitor.newTemp(-1);
monitor.newTemp(1);
verify(counterMock, times(3)).increment();
```

`counterMock` keeps track of its interactions with other objects.

The `verify` statement tests if `increment` has been called on `counterMock` three times.

If it hasn't, the test fails.

The pros and cons of mocking are the opposite of the classicist approach. Main con: it makes refactoring hard.

Mocking by hand

Let's see how to do that example using hand mocking (i.e. without Mockito).

First, create an interface that reflects how the **`MonitorTemp`** class views the **`Counter`** class:

```
interface CounterInf {
  public void increment();
}
```

Create the Mock

E.g. to enable checking how many times **increment** is called:

```
class CounterMock implements CounterInf {
  private int numOfCallsToIncrement;

  public CounterMock() {
    numOfCallsToIncrement = 0;
  }

  public void increment(){
    numOfCallsToIncrement++;
  }

  public int getIncrementNumOfCalls() {
    return numOfCallsToIncrement;
  }
}
```

Using the Mock

Sample unit test:

```
@Test
void testForThreeReadings() {
  CounterMock counter = new CounterMock();
  MonitorTemp monitor = new MonitorTemp(counterMock);
  monitor.newTemp(12);
  monitor.newTemp(-1);
  monitor.newTemp(1);
  assertEquals(3, counter.getIncrementNumOfCalls());
}
```

You can add stub methods to `CounterMock` to return certain values, check that methods are called a certain number of times, check that method arguments have the correct values -- anything you want.

When to Mock?

Any class that the class under test depends on *could* be mocked.

Any class that performs an external action like database access or sending an email *should* be mocked.

If there's interesting interaction between the class under test and a class it depends on, consider mocking the dependent class.

After that, how much mocking you should do is a matter of judgment.

Remember that the more you mock the interaction between objects, the harder the code will become to refactor.

Advantages of TDD

Code is written with testability in mind.

Excellent code coverage tends to happens naturally*.

Thinking about testing first helps the developer to focus on *what* the method does before thinking about *how* to do it.

Reduces the risk of time/customer pressure leading to testing being delayed or even omitted.

* In SE-Radio Episode 324, the creator of JaCoCo, Marc Hoffmann, strongly promotes TDD as the best way to achieve meaningful test coverage.

TDD Summary

Testing is a vital part of software development.

Unit testing involves writing test cases for individual classes. A **unit testing framework** like JUnit can manage these test cases and run them on command.

Test-Driven Development (TDD) is where test cases are written first, as a way of driving the development of software.

Two main styles of TDD: Classicist and Mockist.

Introduction to Software Quality

Comp 47480: Contemporary Software Development

Is Programming Art, Science or Engineering?

Programming is not Art

Is Programming Engineering?

This a debatable topic.

The processes we use to develop software don't have the same rigour as we'd expect in an engineering process.

Physicist Freeman Dyson wrote, "A good engineer is a person who makes a design that works with as few original ideas as possible."

However many software companies break new ground in their development and deployment of software.

Also, software is expected to evolve continuously, unlike the artefacts typically created in physical engineering.

Programming is often viewed as a **craft**

Craftsmanship

One definition:

Craftsmanship is the quality that something has when it is beautiful and has been very carefully made.

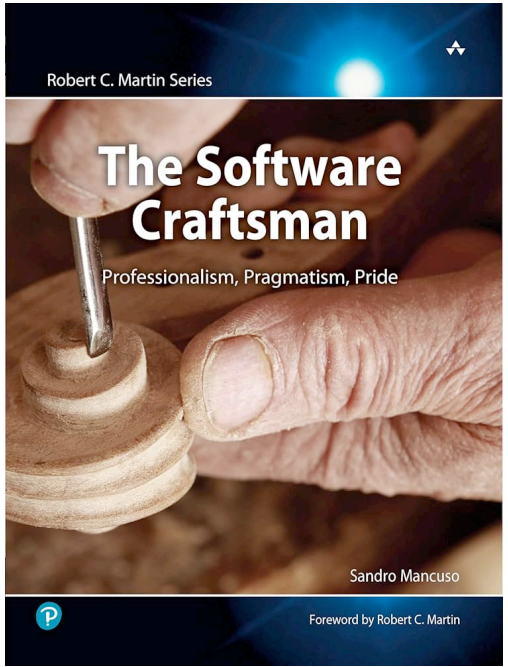
Source: Collins Dictionary

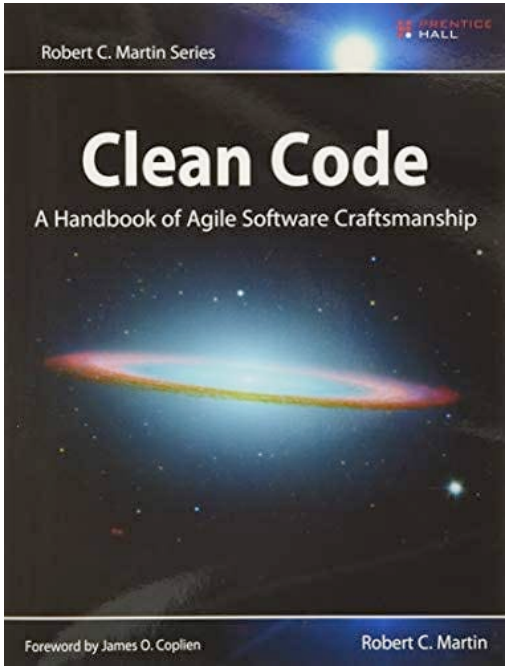
Software Craftsmanship

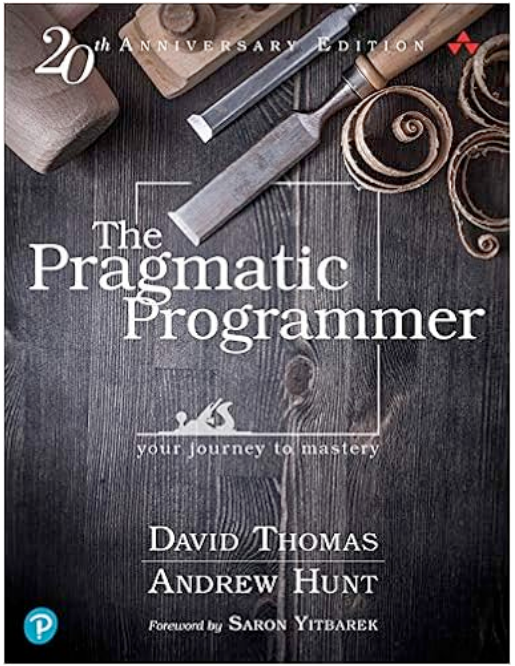
Extends Agile with a focus on the production of high-quality software that can be evolved easily.

It views programming as a medieval craft, rather than a mathematical or engineering process.

Much of the material in the reminder of this module comes under the 'craftsmanship' umbrella.







Doesn't software just have to work correctly?

Remember that real software systems **evolve continuously**.

- New features
- Bug fixes
- Changing technology

Code needs to be crafted to enable

- Understandability
- Flexibility
- Extendibility

etc.

collectively termed "ilities"

Principles and Patterns

The activity of design can be informed by:

principles: general statements of what properties a good design should have

heuristics: guidelines for creating a good design

metrics: how to measure the quality of a design

patterns: concrete examples of reusable, successful designs for particular purposes.

We'll explore these topics in the coming lectures.

We now look at several over-arching **software principles**.

Coupling and Cohesion

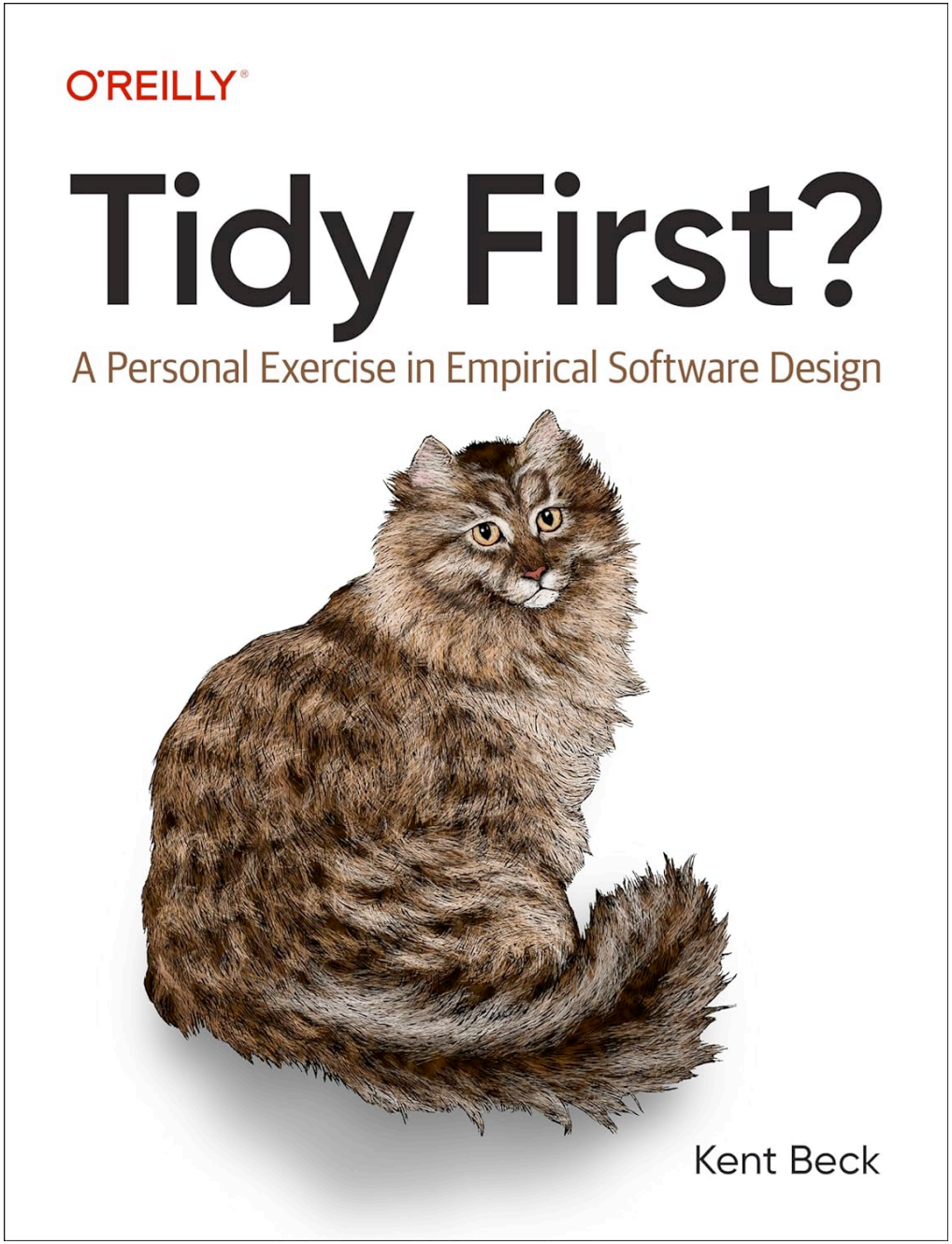
Two foundational principles can be said to capture the essence of software quality:

coupling and **cohesion**

We'll look at these concepts very briefly now, and will return to them frequently.

"Tidy First?"

Kent Beck's recent book also makes use of **coupling and cohesion** as its starting point for discussing quality



Cohesion

A module is said to be **cohesive** if its components are closely interrelated.

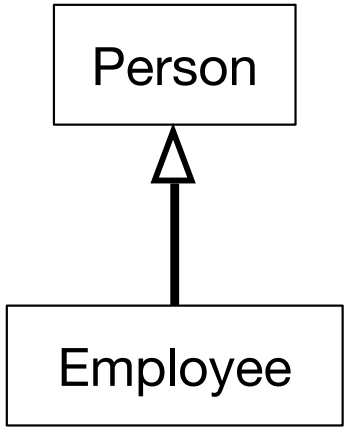
```
class Mailbox{
  public void addMessage(Message){...}
  public void removeMessage(int){...}
  public String getCommand(){...}
  public void printMessage(int){...}
   ...
}
```

This class is cohesive except for the `getCommand` method, which clearly belongs elsewhere.

Coupling

Two modules are coupled if a change in one is likely to lead to a change in the other.

`Employee` is a subclass of `Person`. What coupling exists here?

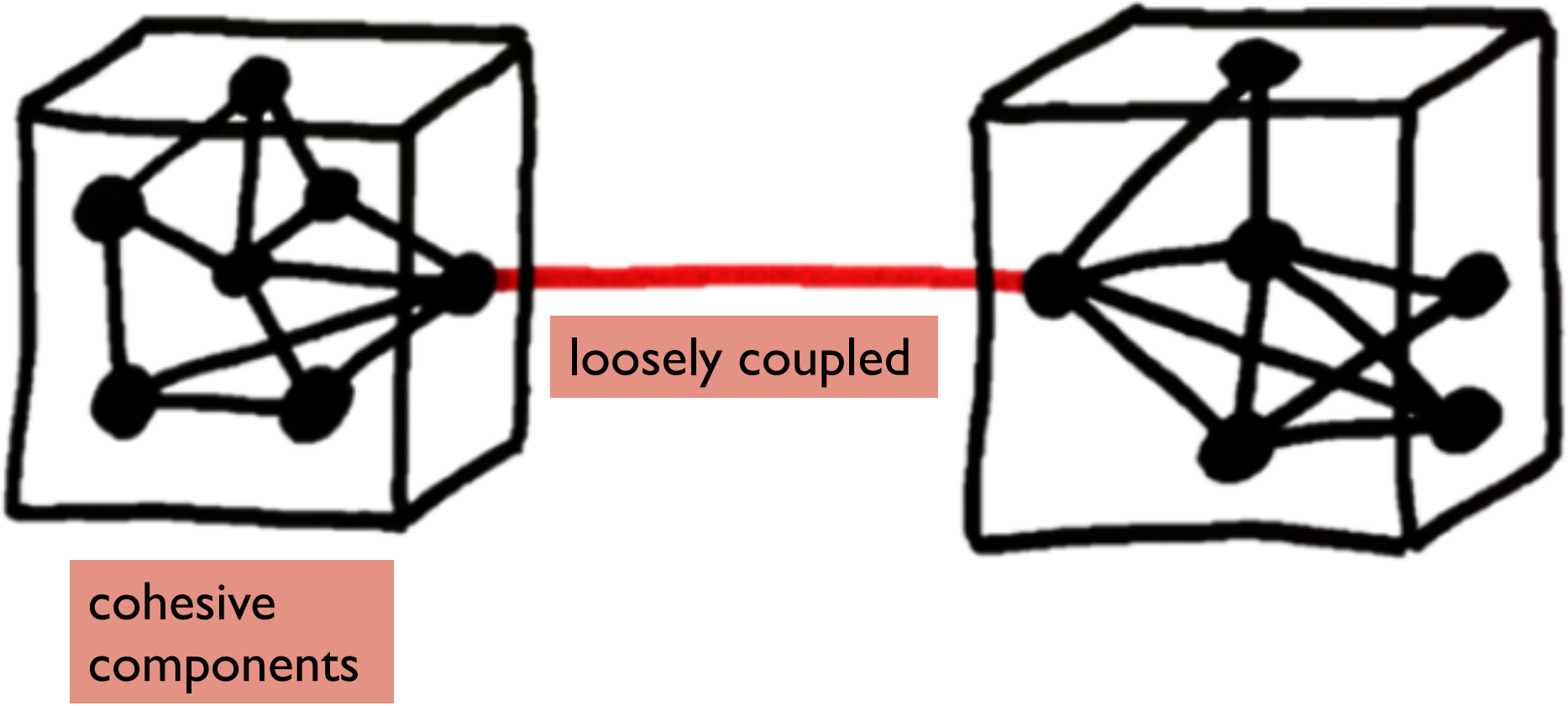


These two classes are *explicitly* coupled.

Recall the **fragile base class problem** from COMP30950.

Implicit coupling is more insidious. More on this later.

Tight Cohesion, Loose Coupling



Advantages of High Cohesion and Low Coupling

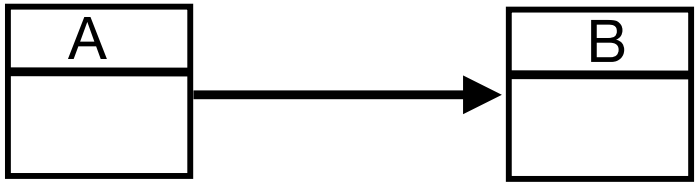
Classes are easier to **understand**. They are focussed on ‘one thing’ (high cohesion) and interact little with the rest of the system (low coupling).

Loose coupling means that a **change** in one class is less likely to require changes to another class. This facilitates **evolution**.

Classes are more easy to **reuse** because they can be understood on their own and rely little on their context.

Coupling and Dependency

Note: **coupling** and **dependency** are essentially synonymous.



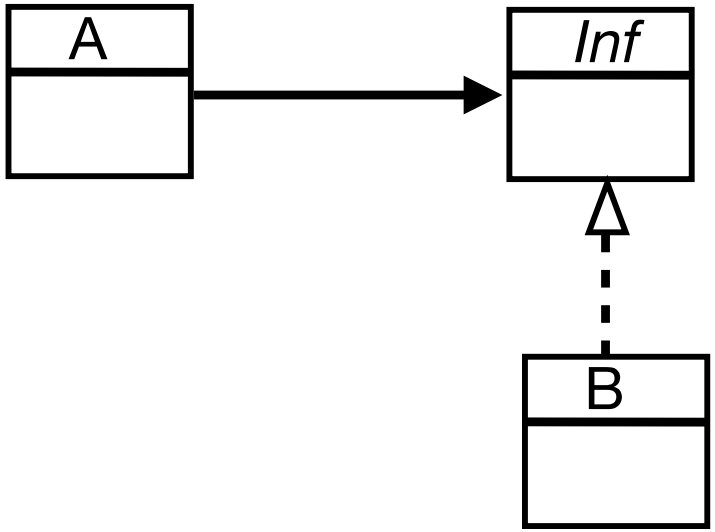
Here class A and B are coupled.

A depends on B.

Dependency is often the preferred term, as it also gives directionality.

Static Coupling, Dynamic Coupling

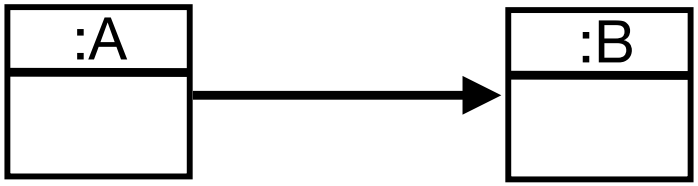
We're concerned with **static (compile-time)** dependency, not dynamic (run-time) dependency.



class diagram

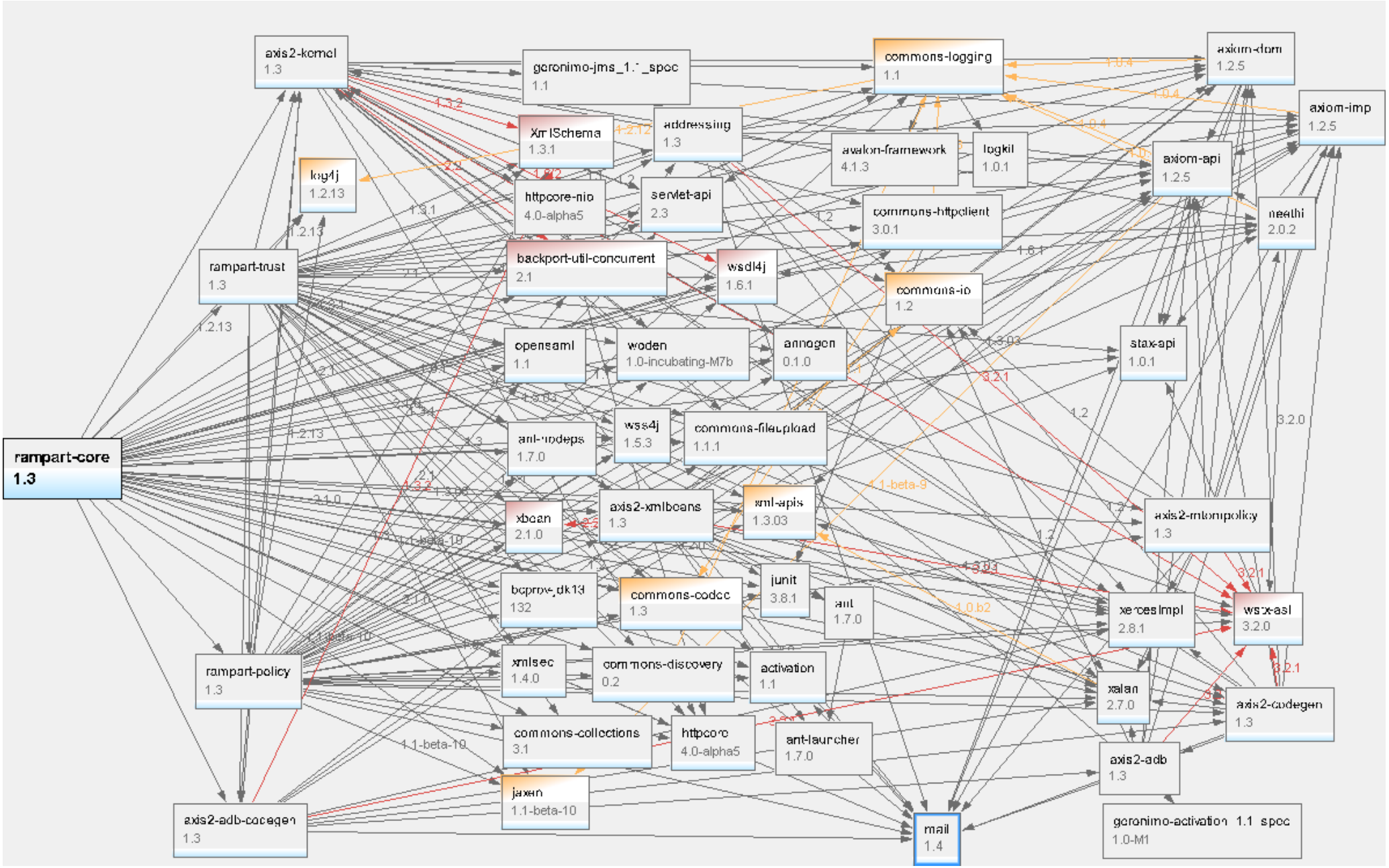
Class A does not depend on Class B directly.

However the run-time object structure is different:



object diagram

Dependencies in a real system



The huge amount of dependency would make this system hard to work with.

An über principle: Keep Dependency to a minimum

Say in a large code base, a new requirement means that Class A has to be changed…

Two important questions are:

1. what classes depend on Class A? If A is changed, they may need to be changed too.

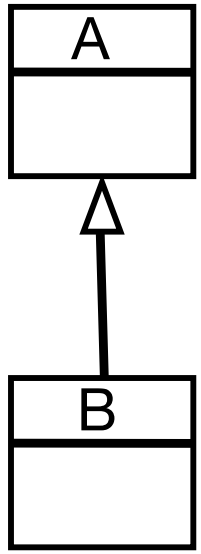
2. What classes does A depend on? To understand A, it may be necessary to understand them first.

Equivalently:
reduce coupling

More about **Dependency**

In designing anything (software, buildings, airplanes), we have components that **depend** on each other.

For example, when class `B` inherits from class `A`, what dependency is created?



As before, `B` depends on `A`.

Dependency

If a class `Foo` uses an instance of class `FooBar`, what dependency is created?

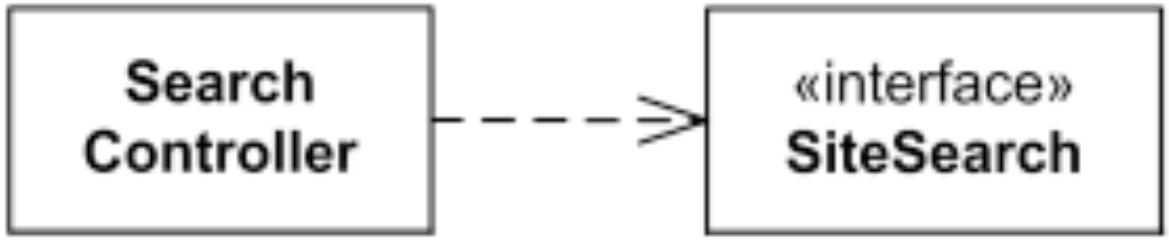
```
class Foo {
  public Foo {
    ... = new FooBar();

  }
  ...
}
```

`Foo` depends on `FooBar`.

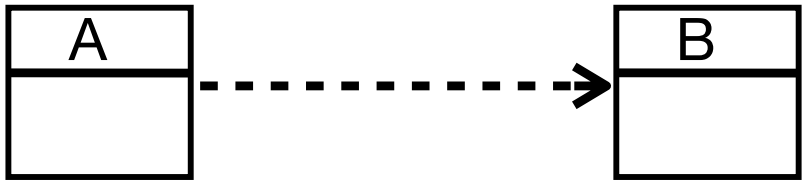
UML and Dependency

UML provides a syntax (dashed arrow) to depict general, non-specific dependency:



Other UML relationships such as uses, aggregation etc. are specialisations of dependency.

Why does dependency matter?



A depends on B. That means...

If a new requirement causes us to update B,
A may well have to be updated as well.

So if B is updated a lot,
A may have to be updated a lot as well.

Hold this thought!

Stable and Unstable

A **stable** component is one that doesn’t change much over time.

Conversely, an **unstable** component changes frequently over time.

("change over time" really means "needs to be changed as the software evolves.")

Note: The notion of stable/unstable comes from Bob Martin's *Clean Code* book.

Stability and Dependency

Guideline 1: **Separate the stable from the unstable.**

Guideline 2: **The stable must not depend on the unstable.**

If you ignore the 1st guideline, your whole system may become unstable.

If you ignore the 2nd guideline, your whole system may become unstable.

The unstable may depend on the stable, that's ok.

Stability and Dependency

Guideline 1: **Separate the stable from the unstable.**

Guideline 2: **The stable must not depend on the unstable.**

If you ignore the 1st guideline, your whole system may become unstable.

If you ignore the 2nd guideline, your whole system may become unstable.

The unstable may depend on the stable, that's ok.

An über principle: Don’t Repeat Yourself (DRY)

Aka Write Once and Once Only.

In a 2014 survey of 328 developers at Microsoft, **code duplication** was the only code smell mentioned as a reason to refactor code.

Duplicated code => **implicit** dependency!

Implicit dependency is the worst type

Source: M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. IEEE Trans. Softw. Eng., 40(7):633–649, July 2014. (Developers cited the main benefits of refactoring to be: improved readability (43%), improved maintainability (30%), improved extensibility (27%) and fewer bugs (27%). When asked what provokes them to refactor, the main reason provided was poor readability (22%). Only one ‘official’ code smell was mentioned, that being code duplication (13%).)

How common is duplication?

The Squeaksource ecosystem was analysed to determine how much code duplication existed*.

Squeaksource features thousands of Smalltalk projects, with more than 40 million versions of methods, across more than seven years of evolution.

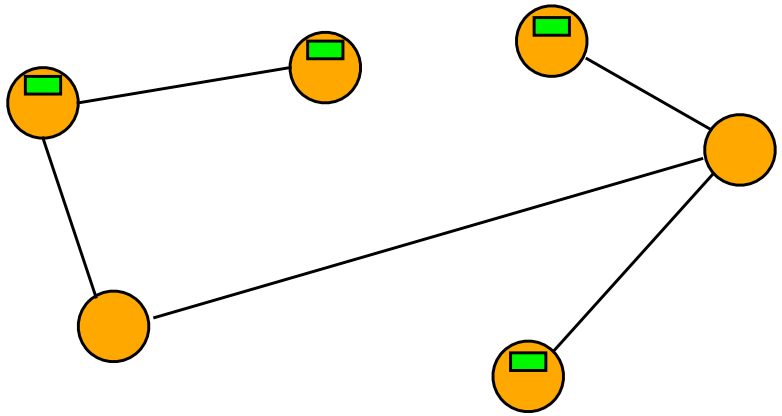
Key finding: More than 14% of all methods were copied from another package in another project.

(A 2017 study found that 70% of the code on GitHub consists of copies of previously created files.)

* Schwarz, N. et al., On how often code is cloned across repositories, International Conference on Software Engineering, 2012.

Duplication: the root of much software evil

Information that appears in several places around the system is contrary to this principle.



Implicit coupling between green boxes

The green boxes above represent duplicated information; if one has to change, they all have to change.

At best, updating takes longer than when there's no duplication. At worst, errors will occur if you omit one of the updates.

... and how it can be fixed (sometimes)

Summary

We considered the notion of programming as a craft, and then looked at some over-arching software principles.

Coupling (Dependency) and **Cohesion** are central to software quality.

Stability is important in practice, and interacts in an interesting way with dependency.

DRY is a valuable principle, and is in fact a consequence of dependency reduction.

In the coming lectures we'll look at a well-known set of Object-Oriented Principles, the SOLID principles.

Object-Oriented Principles

Comp 47480: Contemporary Software Development

SOLID Principles

- In this section we'll look at the so-called SOLID set of object-oriented principles identified by Bob Martin in the "early days" of OOP:
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen-Closed Principle (OCP)
  - **L**iskov Substitution Principle (LSP)
  - **I**nterface Segregation Principle (ISP)
  - **D**ependency Inversion Principle (DIP)

- We'll also look at two other principles/guidelines:
  - No Concrete Superclasses
  - Law of Demeter

Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Single Responsibility Principle

The Single Responsibility Principle states that:

(i) every class should have a single responsibility

Strongly related to **cohesion**

(ii) that responsibility should be entirely encapsulated by the class.

Related to **coupling**

This is a fundamental principle that expresses the core of good object-oriented design.

Following this principle has this very desirable effect:

There is only one [type of] reason for a class to change.

SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

A Simple SRP Violation

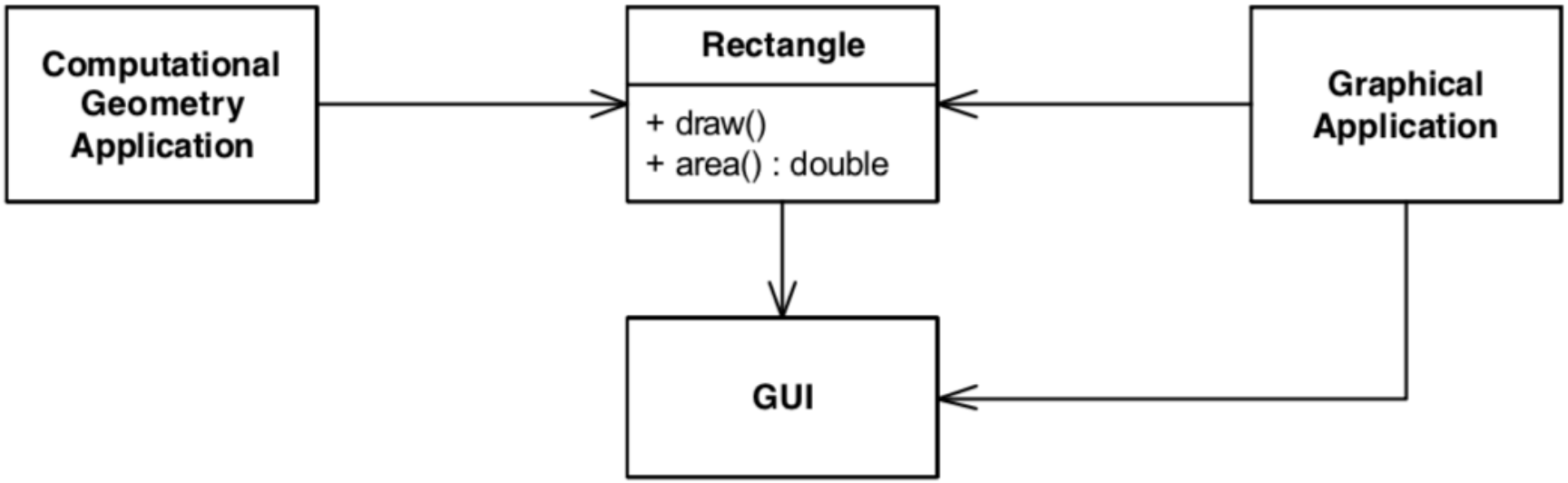
```
class Course {
  void calculateDropoutRate() {
    double rate = ...; // compute dropout rate
    System.out.println(rate);
  }
}
```

It is computing the dropout rate *and* printing the result.

This method has two reasons to change:
(1) a change in how we compute the dropout rate
(2) a change in the UI

See module textbook for discussion of this example.

Violating SRP



The Graphical Application needs to draw and compute area, while the CGA only needs to compute area.

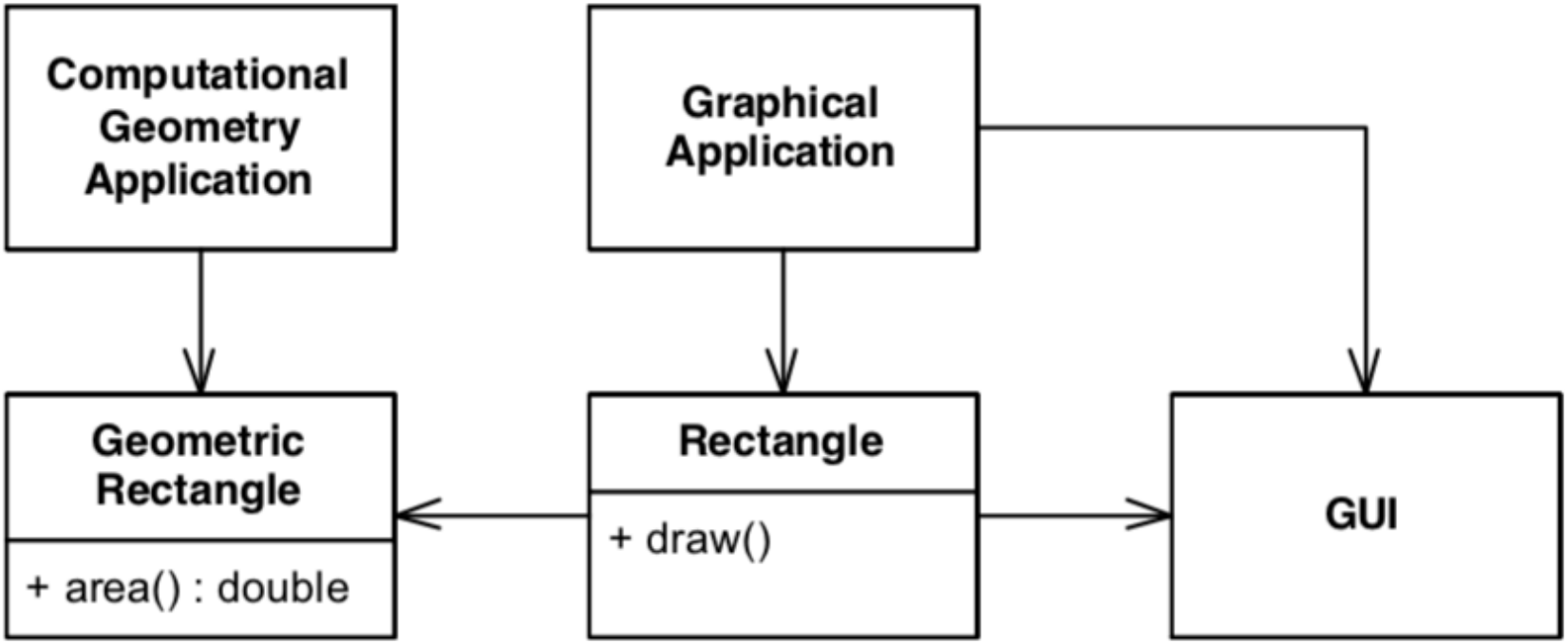
This design violates the SRP.

Why?

Discuss the consequences.

With thanks to Bob Martin for the example.

Redesign to comply with the SRP



The Rectangle class has been split into two, representing the different responsibilities involved (mathematical model of rectangle and graphical rectangle).

Another type of SRP Violation

Don’t forget the ‘**entirely encapsulated**’ clause!

This is violated when part of the natural responsibility of a class is placed in another class.

This reduces comprehension, increases coupling and leads to *shotgun surgery* (we'll return to this).

It can also lead to wholesale code duplication, as clients don't find the required functionality where they expect it, and implement it themselves.

Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Interface Segregation Principle

The Interface Segregation Principle (ISP) states simply that a client should not be forced to depend on methods it does not use.

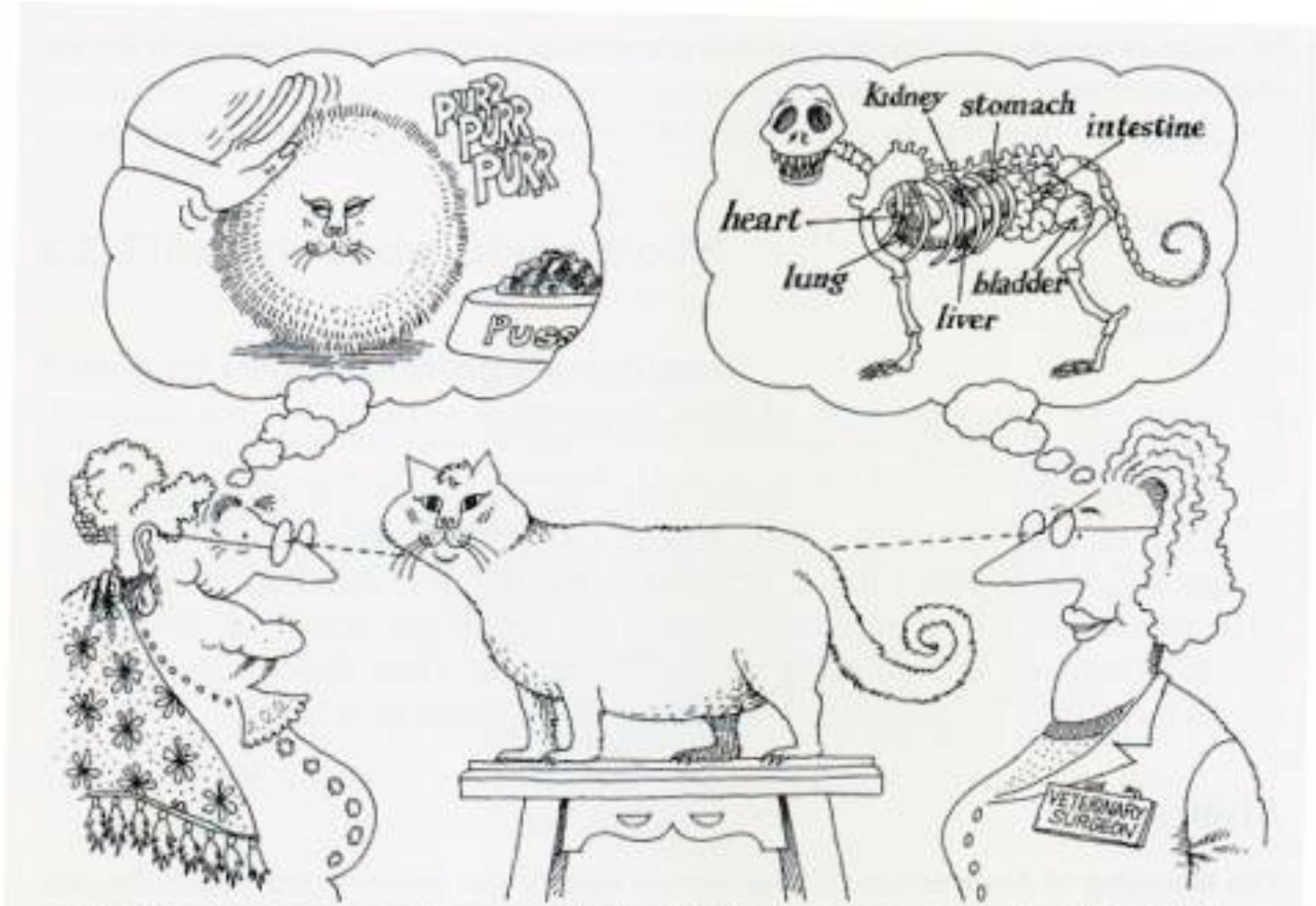
Another way of thinking about this is to say: **interfaces belong to clients**.

We'll look at some examples.

Interface Segregation Principle

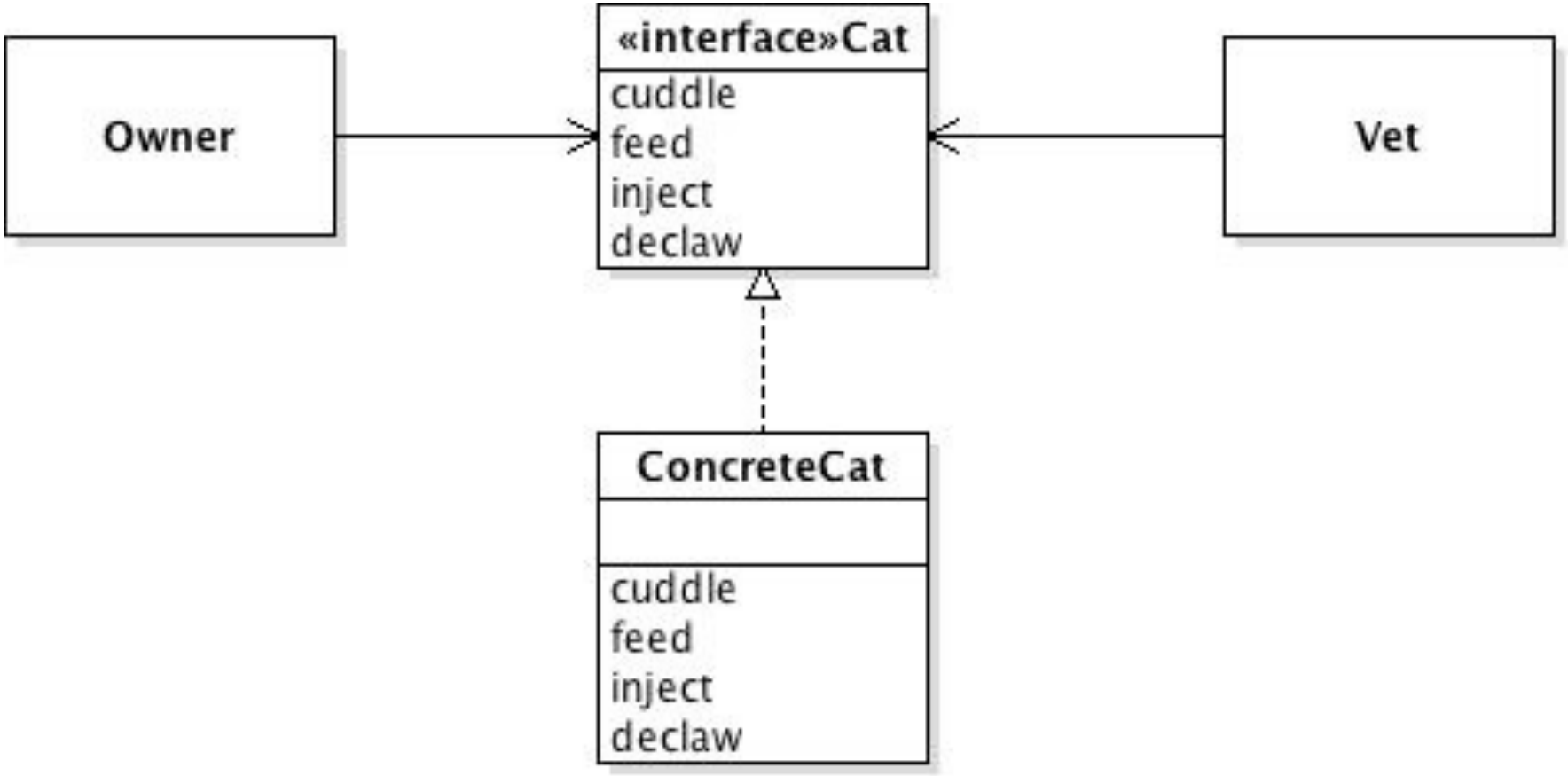
Tailor your Interfaces to the Client's Specific Requirements

Interfaces are in the eye of the Client



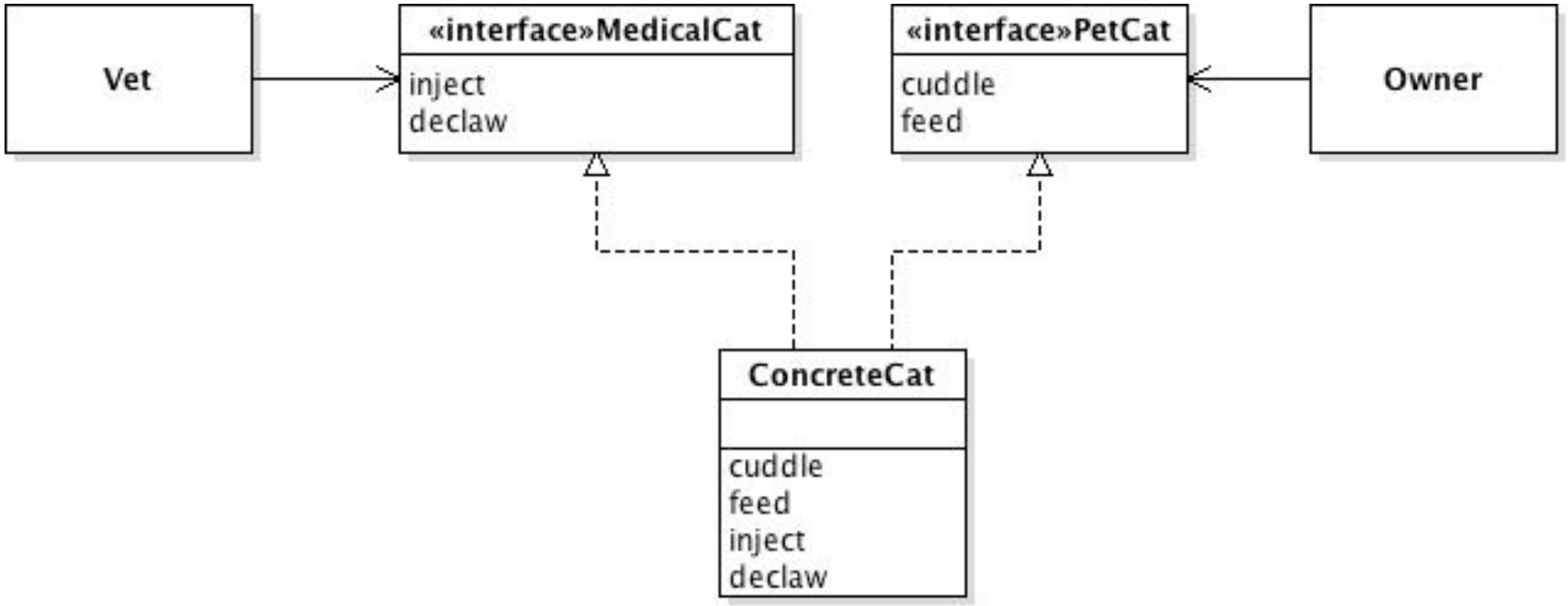
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Cat example without ISP



Owners and Vets can do what they want with the Cat, but each sees an interface that is too wide.

Cat example with ISP



Again, Owners and Vets can do what they want with the Cat, and each sees only the interface they need to use.

Note that the `ConcreteCat` class still looks uncohesive, and could be split to comply better with the SRP.

Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Open-Closed Principle

A module is **closed** if it is not subject to modification. Clients can rely on the services the module provides.

A module is said to be **open** if it is available for extension.

Closed is good -- we can depend on it.

Open is good -- we can extend it.

Is is possible to create modules that are both **open** and **closed**?

i.e. *open for extension* and *closed for modification*?

i.e. can we create modules whose behaviour we can change without actually changing the code of the module?

The Open-Closed Principle

The open-closed principle states that developers should try to produce modules that are both open and closed.

- **open** for extension
- **closed** for modification

This sounds like a paradox. To resolve it we need a way of extending what a module does without changing its code.

We'll see several ways to achieve this in the coming slides.

Achieving Open Modules

Many basic programming techniques contribute towards making modules open:

Method Parameters: enable methods to be used in a variety of contexts.

Higher-Order Functions (e.g. `map`, `reduce` etc.): take functions as arguments, which makes them very tailorable.

Type Parameters (e.g. `Stack[T]`): enable classes work with a variety of types.

Inheritance: a method is open to accepting as argument a subclass of the class of the argument. Also applies to interfaces.

Design Patterns: usually open the program to a new *axis of change*. We'll cover design patterns later in the module.

OCP in Action

A naive implementation of the `List` class in Scala:

```
class List[T]:
  def sortWith(before: (T, T) => Boolean): List =
    ...

  def reduce(f: (T, T) => T): T =
    ...
```

The `List` class is **open** to storing objects of any type `T`.

The `sortWith` method is **open** to sorting in any order, as given in the `before` argument.

The `reduce` hof is **open** to performing any kind of reduction, as given in the `f` argument.

Axes of Change

The notion of *axis of change* of often used in the context of the Open-Closed Principle.

We visualise the program as a point in a vector space, where each dimension represents a type of change.

E.g the List class is **open** to change on these axes:

to create a `List` of a new type of object

to sort the `List` in a new way

to reduce the `List` in a new way

Other the other hand, it is **closed** to changing the data structure employed to store the list.

(we could open it on this axis using the Bridge design pattern.)

Example of using Inheritance for OCP

```
class Driver {
  private FordPuma car;
  ...
  public void drive(){
    car.startEngine();
    car.accelerate();
    ...
  }
}
```

Client

```
class FordPuma {
  private float currentSpeed;
  private boolean doorOpen;
  ...
  public void startEngine(){...}
  public void accelerate(){...}
}
```

Supplier

Problems with this Model?

Firstly, if drivers only ever drive Ford Puma cars, there is nothing wrong with this model!

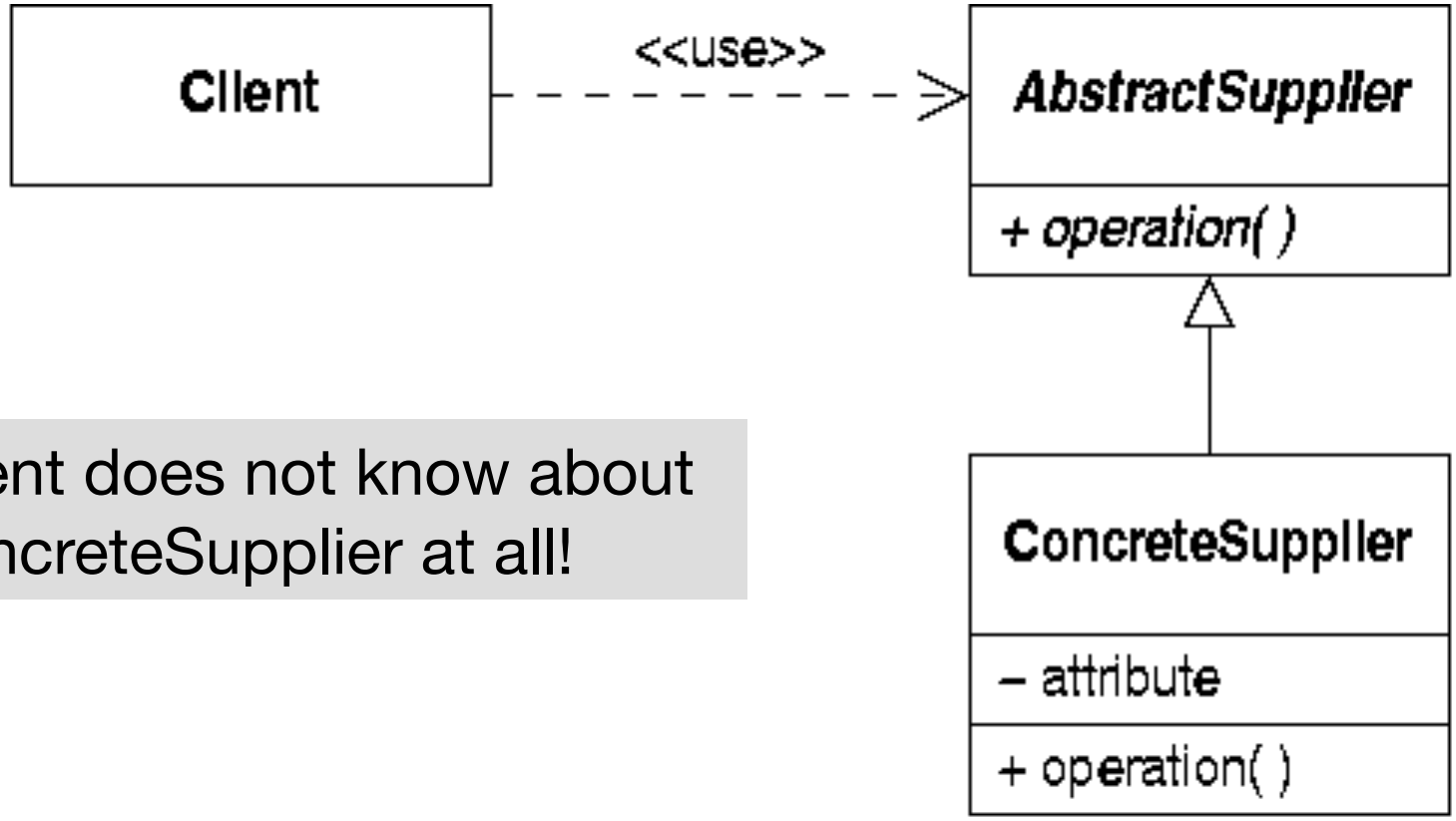
However if we envisage that drivers may drive other car types, this is a poor design.

The client (`Driver`) is dependent on the **actual class** of the supplier (`FordPuma`).

This means that the `Driver` class is **closed** to being evolved to drive a new type of car.

Applying OCP to Reduce Dependency on Supplier

A more flexible approach is one based on an interfaces/ abstract class:



Client does not know about ConcreteSupplier at all!

Applying OCP in Code

```
class Driver {
  private Car car;
  public void drive(){
    car.startEngine();
    car.accelerate();
    ...
  }
}
```

Client

```
interface Car {
  public void startEngine();
  public void accelerate();
  ...
}
```

Abstract Supplier

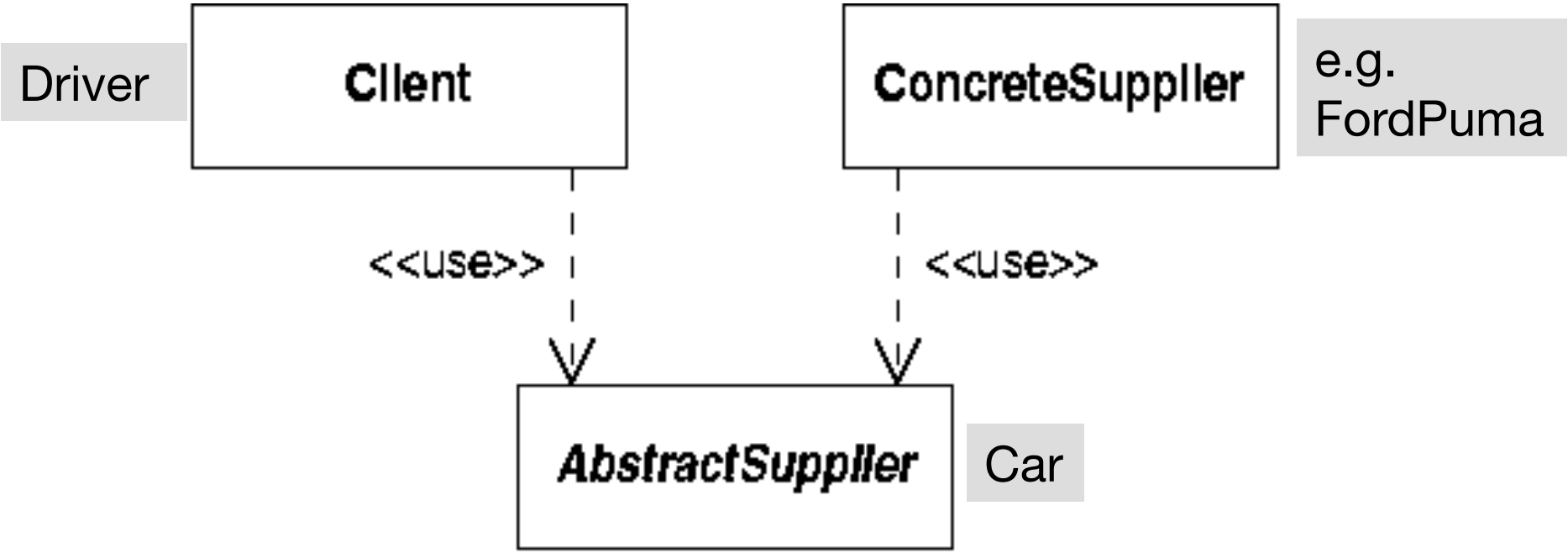
```
class FordPuma implements Car {
  private float currentSpeed;
  private boolean doorOpen;
  ...
  public void startEngine(){...}
  public void accelerate(){...}
}
```

Concrete Supplier

Note that `Driver` is now **open** to being evolved to drive a new type of car, without modification.

Looking at the dependency structure

The **dependency structure** indicates that the client is independent of the concrete suppliers:



This makes `Client` **open** to working with new types of `ConcreteSupplier` without modification.

What to leave open, what to close?

It is usually impossible to predict in what ways a class will be extended.

Makes classes open for extensions that never become necessary is a bad idea.

- waste of time
- violates YAGNI
- it's **overengineering** (makes code harder to work with)

Current Agile practice is to develop software in a closed style, but apply the OCP as soon as it is necessary.

Many **design patterns** are related to this principle. Think about this later on in the module.

An Evolving Shapes Example (in C#)

```
public class Rectangle {
  public double Width { get; set; }
  public double Height { get; set; }
}
```

Does this code look ok so far?

Do you see the likely SRP violation?

From http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle

Area Calculator for Rectangles

```
public class AreaCalculator {
  public double Area(Rectangle[] shapes) {
    double area = 0;
    foreach (var shape in shapes) {
      area += shape.Width * shape.Height;
    }
    return area;
  }
}
```

OK so far?

This code clearly violates the SRP.

Adding Circles

```
public class AreaCalculator {
 public double Area(object[] shapes) {
   double area = 0;
   foreach (var shape in shapes) {
     if (shape is Rectangle) {
       Rectangle rectangle = (Rectangle) shape;
       area += rectangle.Width * rectangle.Height;
     }
     else {
       Circle circle = (Circle)shape;
       area += circle.Radius*circle.Radius*Math.PI;
     }
   }
  return area;
}
```

This code is now looking really bad.

Plus we can't **extend** this class to handle new `Shape` types without **modifying** it.

Let's fix this...

Start applying the OCP

```
public abstract class Shape {
  public abstract double Area();
}
```

This is our new interface for all `Shape` classes.

Rectangle and Circle are Shapes

```
public class Rectangle : Shape
{
  public double Width { get; set; }
  public double Height { get; set; }
  public override double Area() {
    return Width * Height;
  }
}

public class Circle : Shape
{
  public double Radius { get; set; }
  public override double Area() {
    return Radius * Radius * Math.PI;
  }
}
```

Our `Shape` classes now comply much better with the SRP as a by-product.

The Area Calculator Updated

```
public class AreaCalculator {
  public double Area(Shape[] shapes) {
    double area = 0;
    foreach (var shape in shapes) {
      area += shape.Area();
    }
    return area;
  }
}
```

Again, the `AreaCalculator` class now complies with the SRP as well.

And we **can** extend this class to handle new `Shape` types without **modifying** it so it now complies the the OCP.

Object-Oriented Principles: Roadmap

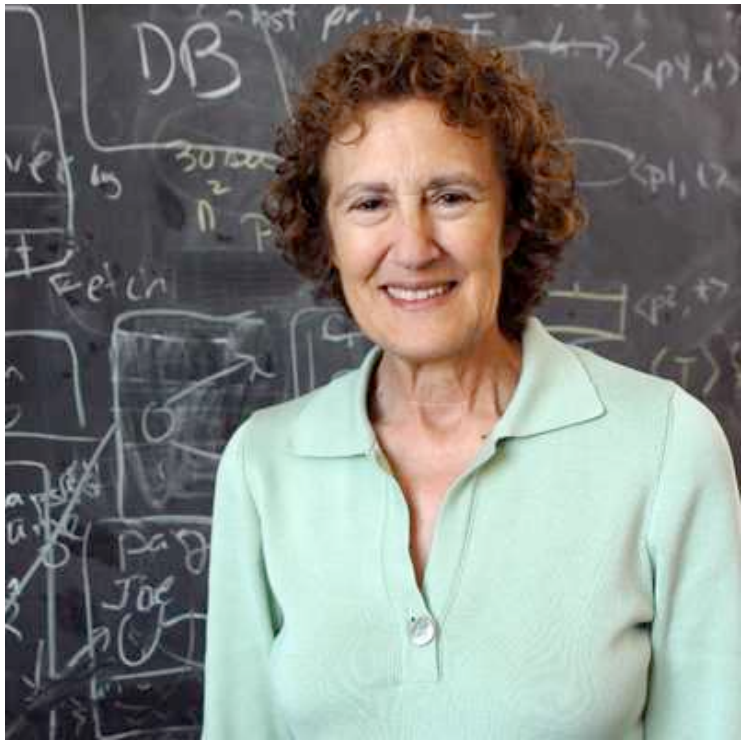
- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Liskov Substitution Principle

The **Liskov Substitution Principle** roughly stated is this:

> "Any context that expects an instance of a Class A should work correctly when provided with an instance of a subclass of Class A."

Named after **Barbara Liskov**, ACM Turing Award winner 2008.



Liskov Substitution Principle

So e.g. a method that expects a **`Vehicle`** should work correctly when provided a **`Car`**.

The Liskov Substitution Principle is essential in understanding how inheritance and polymorphism interplay in object-oriented programming.

It also has a major impact on how we can override, in a subclass, a method defined in a superclass.

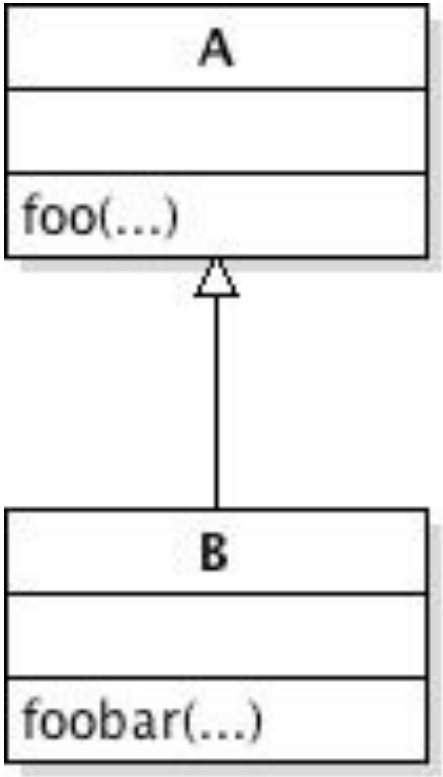
As a slogan (from the perspective of the subclass):

Demand no more, promise no less!

We now explore what the LSP permits and disallows.

Demonstrating Liskov: simple case

Say B is a subclass of A. If B doesn't remove or override any of A's methods, then LSP holds, once any new methods in B preserve any class invariants* of A.



Observing the LSP is trickier when it comes to method overriding, which is what we look at now.

*__class invariant__: a predicate on the fields of a class that should be maintained, e.g., `(1 <= month <= 12)`.

Pre- and Post-Conditions

The **precondition** of method is what the method requires to be true in order for it to run properly.

E.g. the precondition for this method:

```
float sqrt(float n);
```

would be e.g. “**`n`** is a non-negative floating-point value”

Calling `sqrt` with a `String` as argument will lead to a type error, and the result of calling it with a negative value is undefined.

A method's **postcondition** is what is guaranteed to be true once its precondition is true when the method is called (e.g. "return value is square root of **`n`**").

LSP and Method Overriding

The "demand no more, promise no less" slogan applied to method overriding can be expressed this way:

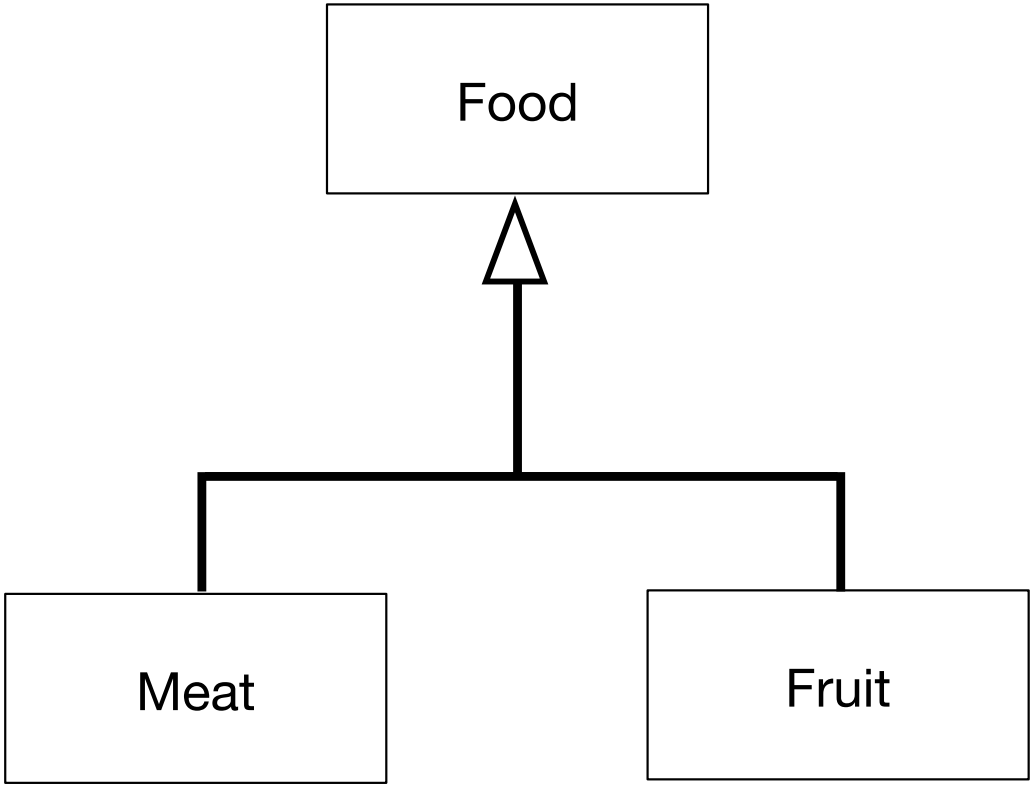
> an overriding method must have a weaker (or equal) precondition than the method it overrides, and
>
> a stronger (or equal) postcondition than the method it overrides.

Is this surprising?

Probably not, but is has some surprising consequences which we now explore.

A concrete example in Java

We assume initially that this simple inheritance hierarchy already exists:



Animals eat food

We want to model that animals eat food:

```
class Animal {
  public void eat(Food f){...};
}
```

Nothing controversial here.

Carnivores eat meat

Carnivores are a type of animal, so we create a class `Carnivore`, a subclass of `Animal`.

Carnivores eat a type of food, namely meat, so we override the `eat` method in `Carnivore` to accept only meat.

```
class Carnivore extends Animal {
  public void eat(Meat m){...};
}
```

All ok? It looks fine…

That violated the Liskov Substitution Principle!

This causes problems when a context that expects an instance of the superclass (`Animal`) is given an instance of the subclass (`Carnivore`).

Here's how we construct such an example:

```
Fruit apple = new Fruit();
Carnivore felix = new Carnivore();
Animal myAnimal = new Animal();

if (coinToss() == heads)
  myAnimal = felix;

myAnimal.eat(apple);
```

This creates a run-time type error whenever **`myAnimal`** refers to an instance of **`Carnivore`**.

What went wrong?

We overrode the **`eat`** method and made its argument more specific, i.e. we “demanded more” in the overriding method.

This violates the LSP.

The code snippet above shows how this permits the new method to run without the new demands being met, hence a run-time type error occurs.

Liskov in Practice

Most languages demand that the argument type in an overriding method be **contravariant**, i.e. become more general in the more specific class.

(Java is even more specific — the arguments in an overriding method must be of the same type.)

The alternative, **covariance**, allows more "natural" modelling (as in the Animal example), but permits run-time type errors to occur.

(Covariance is not mainstream. The Eiffel programming language permits it, but this is no longer a mainstream language.)

Example: the `equals` method in Java

The `Object` class in Java defines an `equals` method as with this signature:

```
public boolean equals(Object o)
```

It would be natural to override this e.g. in a class `Person` as follows:

```
public boolean equals(Person o)
```

This won't work. Why?

The argument type must be *contravariant*, so you can only override this way:

```
public boolean equals(Object o)
```

Example: A Person/Child class hierarchy

Say we have a **`Person`** class as follows:

```
class Person {
  ...
  public void setAge(int newAge) {
    if (newAge >= 0 && newAge <= 125)
      age = newAge
  }
  ...
  private int age;
}
```

Now we want to create a **`Child`** class...

Overriding `setAge` in Person

A **`Child`** is a **`Person`** under the age of 16, so we implement it quickly with:

```
class Child extends Person {
  ...
  public void setAge(int newAge) {
    if (newAge >= 0 && newAge <= 16)
      super.setAge(newAge)
  }
  ...
}
```

Looks ok?

Violating Liskov

`Person::setAge` and `Child::setAge` have the same precondition. No issue there.

Let's consider the postconditions:

Method	Postcondition
`Person::setAge(newAge)`	if newAge in [0, 125] age == newAge
`Child::setAge(newAge)`	if newAge in [0, 16] age == newAge

The postcondition has been made weaker, thus violating the "promise no less" clause of the LSP.

The compiler can't help us here -- we've broken the *behavioural* aspect of the LSP.

This causes real problems (1)

Where **`person`** is of type **`Person`**, what is the effect of this line of code?

```
person.setAge(27)
```

When only the **`Person`** class existed, this line would set the age of **`person`** to 27.

However a colleague has subclassed **`Person`** to create a **`Child`** class as on the earlier slide.

Now this line may have no effect on **`person`** (if it refers to a **`Child`** object).

Very confusing!

This causes real problems (2)

Say we add this method to **`Person`**:

```
public void incrAge{
  if (age < 125)
    age += 1
}
```

What problems may this cause?

Could allow a child's age to exceed 16.

The problem isn't just about the postcondition of the **`setAge`** method, it's that the `Child` class makes the inherited constraint on `age` more specific.

Violating Liskov has made our superclass dependent on its subclasses. Whenever we change it, we'll need to revisit all its subclasses as well. Highly undesirable.

How to model this better?

One possible solution is as follows.

All **Person** objects should have their age limited to a certain range.

Store this range in the **Person** class, and use it in the range checks in **`setAge`** and **`incrAge`**.

In e.g. the **Child** class constructor, set this range to the appropriate one for a child ([0..16]).

This is a safer model, and doesn't violate Liskov.

It would also help to observe the "No Concrete Superclasses" guideline -- see later slides.

This model in code (scala)

```
class Person(
  var age: Int,
  ageRange: Range = Range.inclusive(0, 125)
):
  def setAge(newAge: Int) =
    if ageRange.contains(newAge) then
      age = newAge

  def incrAge =
    if age < ageRange.max then
      age += 1

class Child extends Person(
  0,
  Range.inclusive(0, 16)
)
```

Liskov in Practice

As a programmer you should observe the following:

When adding a new method to a subclass, take care not to break any invariants (documented or implicit) of the superclass.

An overriding method should *extend* the method it overrides, not do something different.

> One the best ways of achieving this is to *invoke* the overridden method in the overriding method.
>
> This is called **method refinement**.

Liskov Summary

The Liskov Substitution Principle explains what substitution means in object-oriented programming.

This principle limits what you can do in a subclass, e.g.,

- inherited methods cannot be rejected
- Overriding methods must have contravariant arguments

Most object-oriented languages try to prevent you from violating the Liskov principle, e.g., in Java:

- you cannot reject an inherited method
- the arguments to an overriding method must be of the the same type as the arguments to the overridden method.

However behavioural subtyping must be observed as well, and the compiler cannot help with this.

Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

Dependency Inversion

The Dependency Inversion principle can be simply stated as:
Abstractions should not depend on details, or
High-level classes should not depend on low-level ones.

How is that **inversion**?

We tend to think of systems in a **top-down** way, where the higher levels depend on the lower ones.

But then changes in the lower levels cause the higher levels to change ("the tail wagging the dog").

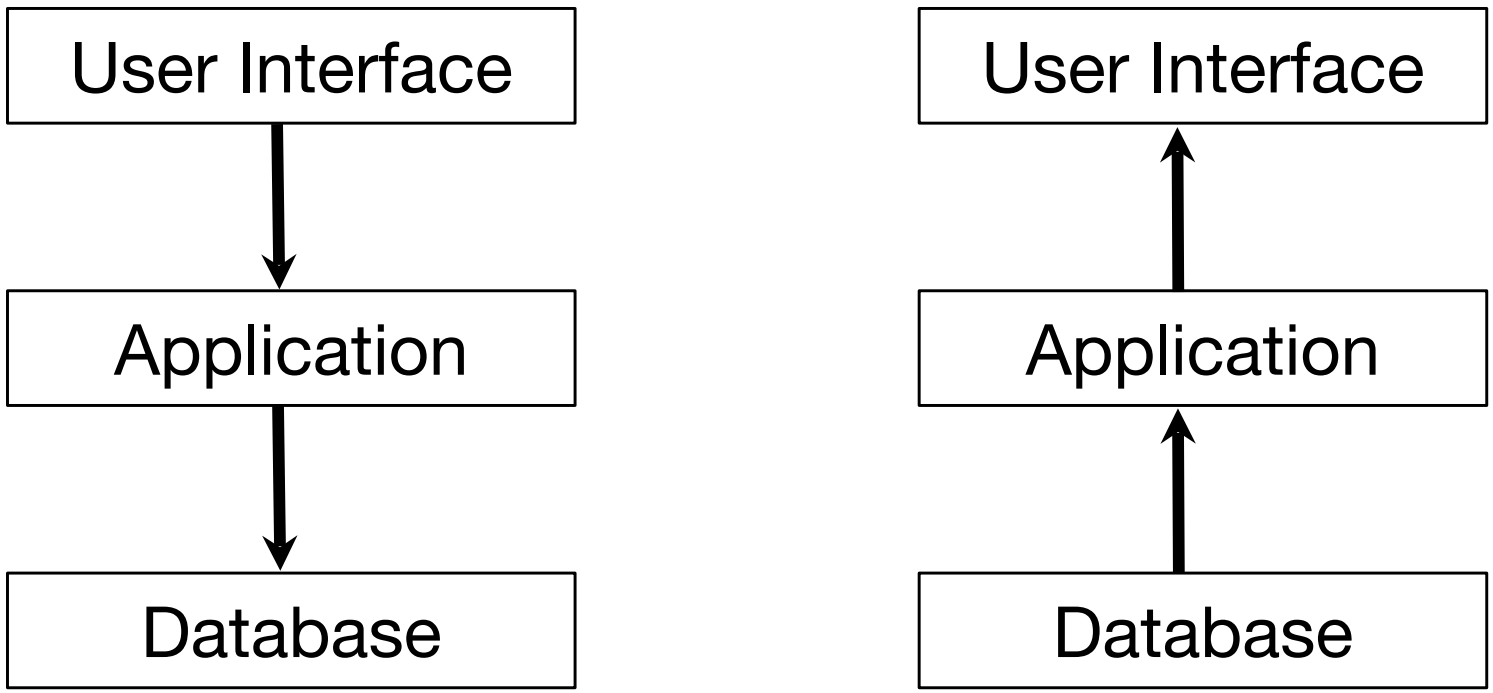
Dependency Inversion means that we invert dependencies to make them **bottom-up**.

Layered Architecture

Systems sometimes employ a **layered architecture**, e.g.
- user interface layer
- application layer
- database layer

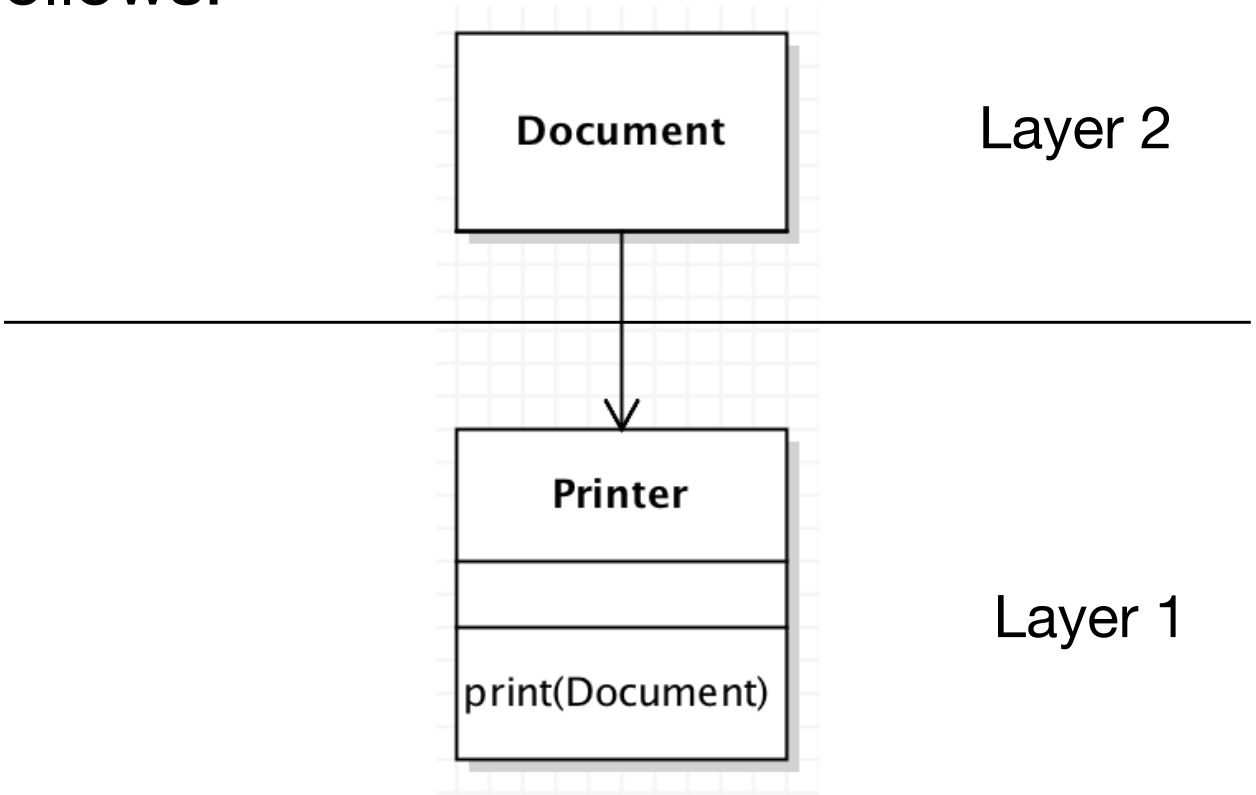
How should the dependencies go in a layered architecture?

Down or **Up**?



How to invert a dependency

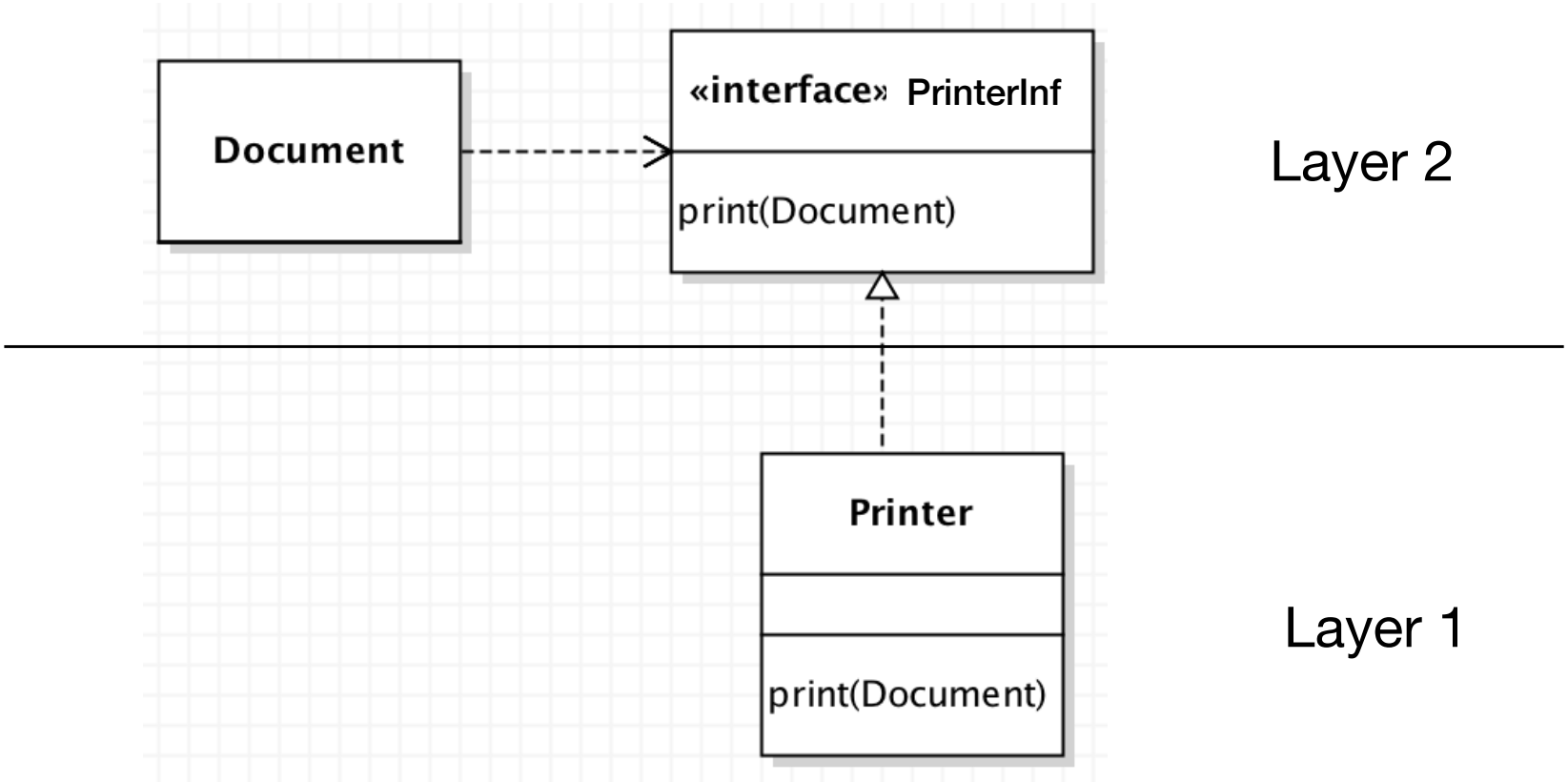
Say a system has a top-down dependency as follows:



How can we invert the dependency from Layer 2 to Layer 1? It may seem impossible...

Inverted to bottom-up dependency

We can however flip the dependency this way:



Note that at run-time, a Document object will still have a reference to a Printer object, but the top-down *static* dependency has disappeared.

Original design in code

```
class Document {
  public Document(Printer printer) {
    this.printer = printer;
  }

  public void foo() {
    printer.print(this);
  }

  private Printer printer;
}

class Printer {
  public void print(Document d) {
    ...
  }
}
```

Refactored to apply DIP

Layer 2

```
class Document {
  public Document(IPrinter printer) {
    this.printer = printer;
  }

  ...

  private IPrinter printer;
}

interface IPrinter {
  public void print(Document d);
}
```

Layer 1

```
class Printer implements IPrinter {
  public void print(Document d) {
    ...
  }
}
```

Comments on this example

Layer 2 has no dependency on Layer 1.

Replacing the `Printer` class in Layer 1 will not affect Layer 2.

Both `Document` and `Printer` depend on `PrinterInf`. This represents the contract between the two classes.

The entire Layer 2 can be reused in another context, once a Layer 1 with a suitable `Printer` is provided.

Reusing Layer 1 is trickier as its interface is buried in Layer 1. This can be remedied by putting `PrinterInf` into its own layer.

Related but Different: *Dependency Injection*

Dependency Injection (DI) is where classes declare their dependencies through their APIs, rather than creating them directly themselves.

Without DI, the dependencies between classes are distributed across the classes themselves.

With DI, the dependencies are centralised and easier to change.

Dependency Injection can be done in a limited way by hand, or it can be done across an entire application using a **dependency injection framework**.

Dependency Injection Example

Say some client code creates a database connection:

```
static void ClientCode() {
  IDbConnection dbConnection = new SqlConnection();
  dbConnection.Open();
  ...
}
```

This hardcodes the `ClientCode -> SqlConnection` dependency. Applying DI avoids this dependency:

```
static void ClientCode(IDbConnection dbConnection) {
   dbConnection.Open();
}
```

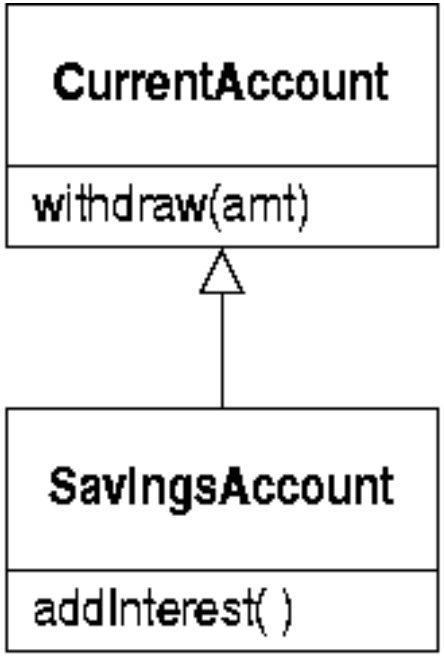
```
static void DICode() {
  IDbConnection dbConnection = new SqlConnection();
  ClientCode(dbConnection);
}
```

Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

No Concrete Superclasses

- It is usually recommended that all superclasses in a system be abstract
- Subclasses are often added to concrete classes as a system evolves e.g.,



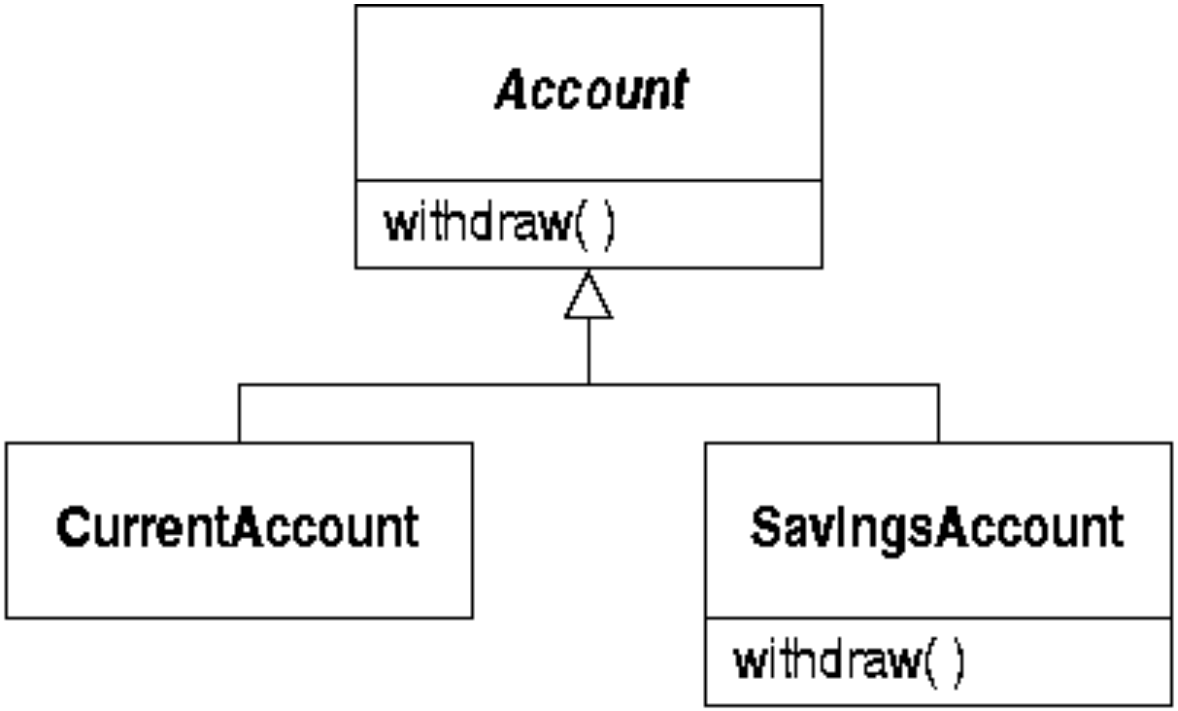
A Dependency Problem

- The savings account class is now dependent on the current account class
  - changes to the current account -- altering functionality or adding and removing operations -- will affect savings accounts

- This leads to problems
  - it may not be desirable to apply these changes savings accounts
  - the code may become cluttered with checks for special cases

Refactoring the Design

- A better approach is to introduce an abstract class



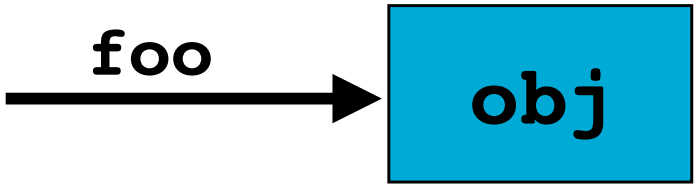
- This is a special case of Dependency Inversion
- Previously SavingAccount depended on CurrentAccount for its implementation; this has been replaced by a shared dependency on the Account abstract class.

Object-Oriented Principles: Roadmap

- Single Responsibility Principle (SRP)
- Interface Segregation Principle (ISP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- No Concrete Superclasses
- The Law of Demeter

The Law of Demeter

What objects should an object **`obj`**, on receipt of a message **`foo`**, be allowed to send messages to?



So the object **`obj`**, receives the message **`foo`**, and executes some method called **`foo`**.

As the method **`foo`** executes, which objects is it able to send messages to?

The Law of Demeter

If we say "any object," we may end up with a very complicated model.

The so-called "Law of Demeter" proposes to limit it to:

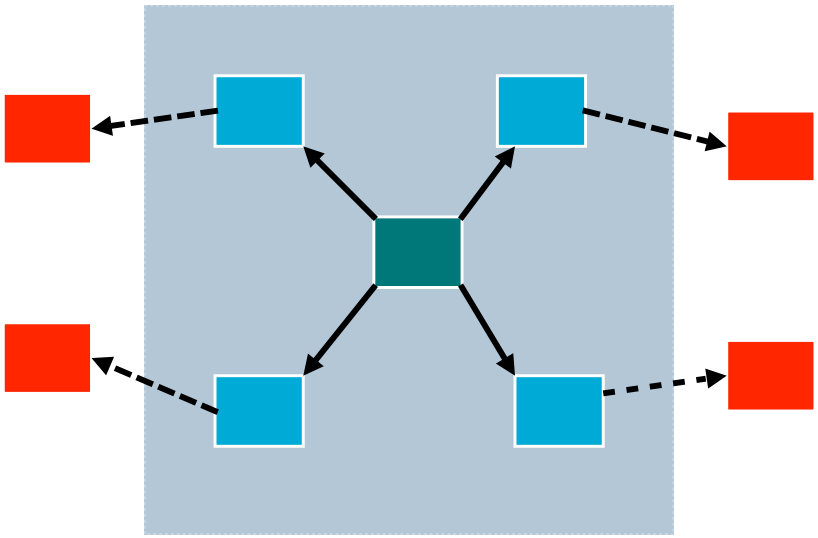
- **`obj`** itself;
- objects that are passed as arguments to **`foo`**;
- objects that are created in **`foo`**;
- objects that are attributes of **`obj`**;
- global objects accessible by **`obj`**.

So what does the Law of Demeter actually disallow?

Who shouldn’t an object talk to?

Demeter explicitly disallows sending a message to a subobject extracted from an accessible object.

This constrains how much an object can know about its environment. i.e. it **reduces dependency**.



Aka "Don't talk to strangers"

The object in the centre should only traverse the links to its immediate neighbours.

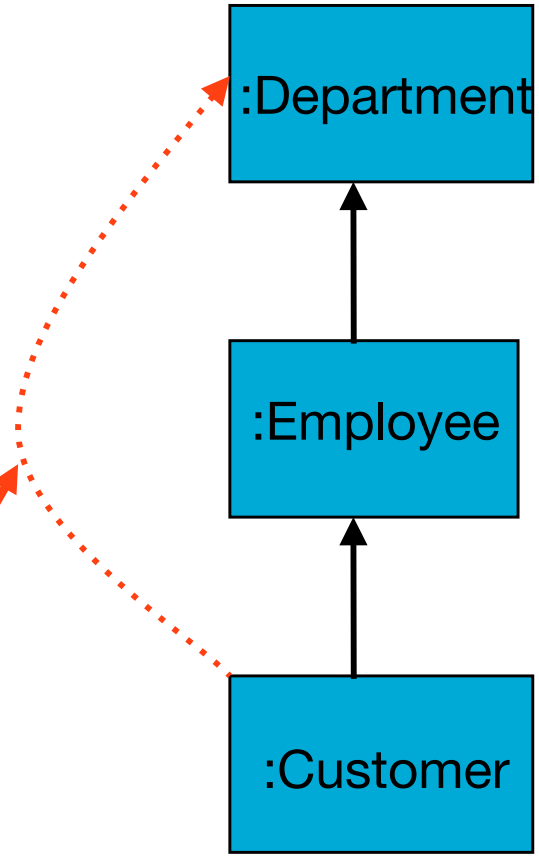
A Violation of Demeter

A violation of the Law of Demeter in program code might look like this:

```
class Department {
   String getName();
}

class Employee {
   Department getDept(){}
   Department dept;
}

class Customer {
   void foo(){
      …
      emp.getDept.getName();
   }
   Employee emp;
}
```



Why is this bad? How would you solve it?

Refactoring to correct

Refactoring this code to resolve this issue produces:

So the **`Customer`** class doesn't even know the **`Department`** class exists.

This may seem like a small matter, but it can be a major simplification in a large system.

```
class Department {
   String getName();
}

class Employee{
   String getDeptName(){
      return dept.getName();
   }
   Department dept;
}

class Customer{
   void foo(){
      …
      emp.getDeptName();
   }
   Employee emp;
}
```

Demeter, strong and weak forms

- The **strong form** of the Law of Demeter prohibits access to objects accessed through inherited instance variables.

- This amounts to providing the same level of data abstraction to subclasses as is given to class clients.

Should I observe Demeter?

Observing Demeter reduces dependency between classes, with all the benefits that brings.

However, it also results in the addition of a lot of small methods and increases the size of public class interfaces.

As with every principle, whether you observe it in any given context is a design decision.

Learner: violates Demeter everywhere
Intermediate: observes Demeter everywhere
Master: observes/violates judiciously

Demeter Summary

- The Law of Demeter essentially avoids encoding the details of system structure in individual methods.

- It is widely regarded as a valuable factor in creating loosely-coupled software that is easier to maintain.

Summary

Object-Oriented Principles are general statements of properties that a good design should have.

By way of contrast, patterns are examples of concrete good design in a certain context, and design heuristics are “rules of thumb” that help in creating a good design.

In this section we looked at several well-known object-oriented design principles.

Don't follow these principles blindly — there are often good reasons to ignore them in a particular context, but you should be aware of the trade-offs in your decision.

Refactoring

Comp 47480: Contemporary Software Development

CODE SMELLS
&
REFACTORING

What is Refactoring?

Refactoring is the process of changing the internal structure of software to make it **easier to understand and modify**, without changing its observable behaviour.

“without changing observable behaviour” -- the program may behave differently internally, but its *observable functionality* remains identical.

Roadmap of this section

This section is structured as follows:

We discuss **refactoring in general**, what it is, where to use it and various issues around its application.

We then examine a **number of specific refactorings**.

Finally we explore **code smells**, hints that refactoring may be beneficial.

Refactoring is central to this module — why?

Agile relies on **emergent design** -- the design of the software evolving over time. This design evolution is enabled, and driven, by refactoring.

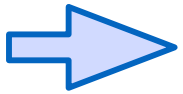
Testing: (1) Unit tests can be used to check that a refactoring hasn't changed behaviour. (2) Refactoring is a core step in Test-Driven Development.

OO Principles: The principles we looked at typically inform the refactoring process — often you refactor to improve adherence to these principles

Design Patterns: The goal of a refactoring sequence may be to introduce a design pattern to a program.

Simple Refactoring Examples

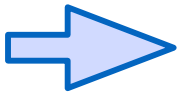
```
class Customer {
   ...
   Date date;
}
```



```
class Customer {
   ...
   Date dateOfLastPayment;
}
```

Rename

```
class A extends B {
   ...
}
```

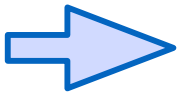


```
class A {
   public A() {
      myB = new B();
   }
   ...
   private B myB;
}
```

Replace Inheritance With Delegation

Another Refactoring Example

```
List<Double> numbers = ...;
double average = 0.0;
for (double num : numbers){
  average += num;
}
average = average / numbers.size();
```



```
List<Double> numbers = ...;
double runningTotal = 0.0;
for (double num : numbers){
  runningTotal += num;
}
double average = runningTotal / numbers.size();
```

Many refactorings we apply don't have formal names.

Refactoring transcends software

"I have rewritten—often several times—every word I have ever published. My pencils outlast their erasers."

— VLADIMIR NABOKOV, SPEAK, MEMORY, 1966

Why Refactor?

Without refactoring program design will **decay** over time.

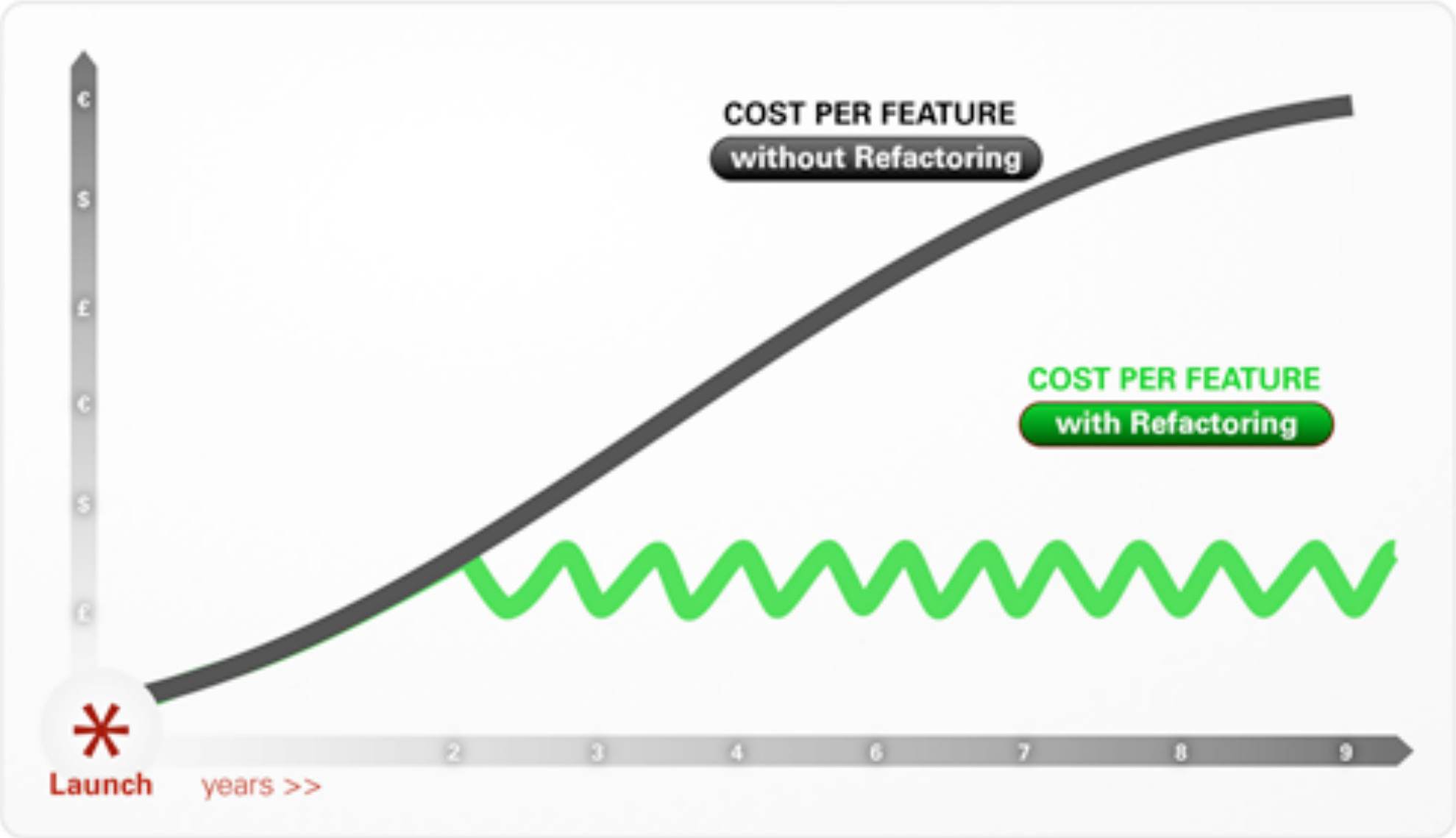
Decay? Really?

- When software is updated in an undisciplined way, its design will match its functionality less and less — it appears to **decay**.

Refactoring makes software easier to understand

- Somebody else will have to read your code, or you will have to return to it in the future
- Makes fixing bugs easier, less chance of creating bugs
- Enables you to program faster

Refactoring aims to reduce maintenance costs



When to Refactor

Refactor whenever you add functionality. If the existing design makes it hard to add the new feature, refactor it.

Refactor as part of TDD. Recall the “red, green, refactor” cycle of Test-Driven Development.

Refactor when you fix a bug. A bug may be a sign that the code was not clear enough in the first place.

Refactor after a code review. A code review is where you learn the team development practices.

Note the so-called “rule of three”. You *might* duplicate code *once*, or hack a single exceptional case *once*. However if it happens again, refactor to solve the design problem properly.

The Role of Test Cases

- If you use a refactoring tool to refactor, you can be confident that program behaviour will not change
  - Not 100% certain -- refactoring tools have bugs too!

- Test cases mean you can refactor radically in confidence
  - Running the test cases assures us that the refactoring has not introduced any new bugs

- The synergy between **automated testing** and **refactoring** is a vital one in current software development practice.

Preconditions

- In order for a refactoring to preserve program behaviour, certain **preconditions** must hold
  - e.g. if you rename a field from **`alpha`** to **`beta`**, there must not already be a field called **`beta`** in the class

- A refactoring tool will check preconditions for you

- For each refactoring we look at, you should have an idea what the key elements of the preconditions are.

COFFEE
REFACTORING

Individual Refactorings

In this section we look at a number of individual refactorings.

There are many other refactorings, but once you understand a number of them the rest seem natural.

Refactorings are mostly language-independent, but we'll assume we're using a statically typed language like Java or C#.

Individual Refactorings

Rename method/field/class

Improve naming

Move class/method/field,
Pull Up method/field,
Push Down method/field

Moving entities to better locations

Extract local variable/constant,
Convert Local Variable to Field

Improving code within a class

Extract Method
Inline Method

Splitting and merging methods

Extract/Collapse superclass
Extract interface

Improving inheritance hierarchy and interfacing

Rename <anything>

- Allows you to rename variables, classes, methods, packages, folders, and almost any Java identifier

```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("{");
    for (Enumeration e= fMonies.elements(); e.hasMoreElem
                                            ent());
    buffer.append("}");
    return buffer.toString();
}
public void appendTo(MoneyBag
    m.appendBag(this);
}
```

Enter new name, press **Enter** to refactor

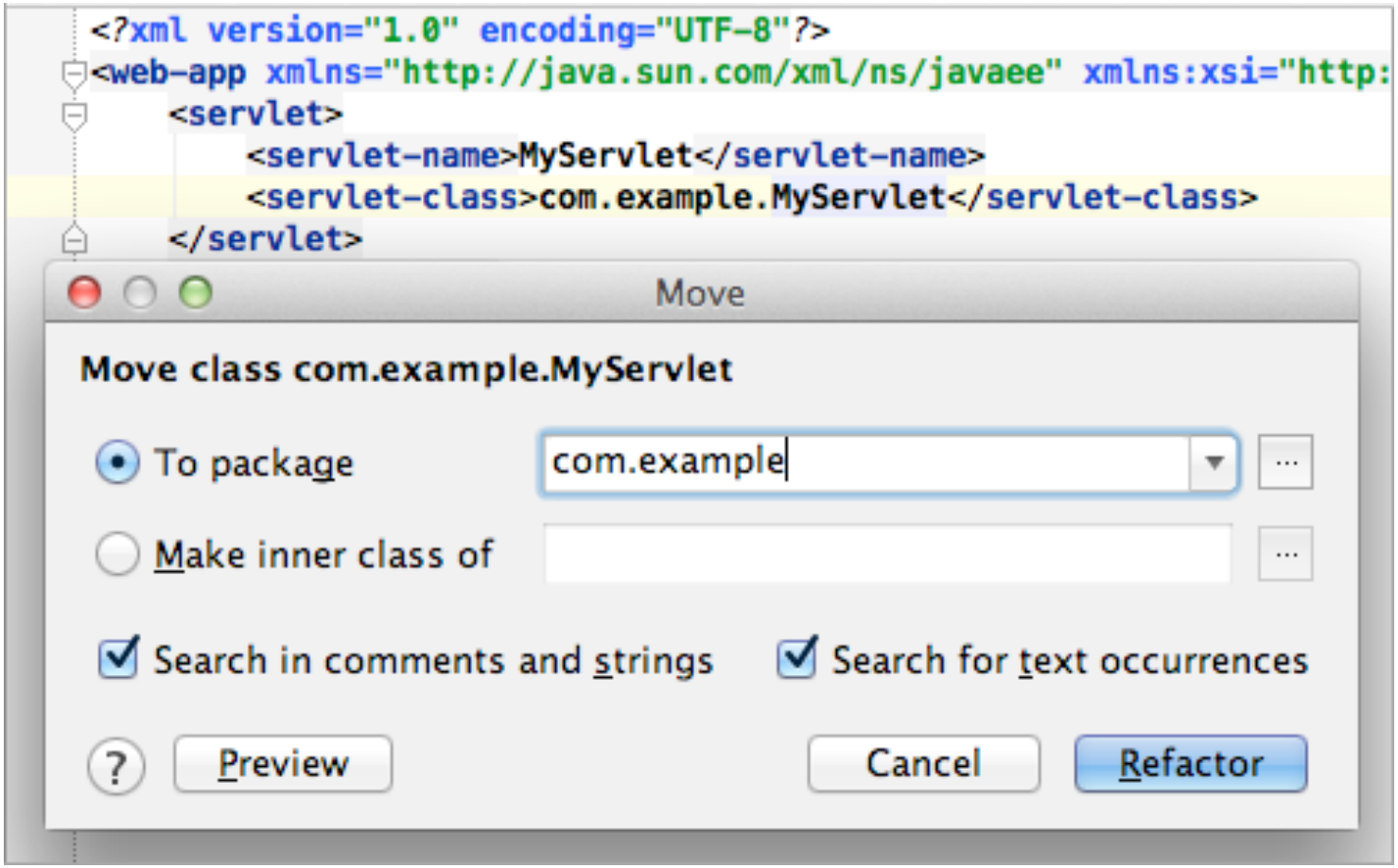
Refactor	Enter
Preview...	Ctrl+Enter
Open Rename Dialog...	Alt+Shift+R
Snap To	▸
Preferences...	

Comments on Rename

- Rename is by far the most commonly used refactoring!
  - although it only clarifies code, not changes its structure
- Useful for making code clearer
  - e.g. don’t comment field declarations; instead Rename the field to a better name that needs no comment
- We’ll see that some other refactorings create new program entities (methods, field etc.)
  - Rename can then be used to give them proper names

Move Class/Method/Field

- Move a class to another package or a method/field to another class.
- For moving a method/field, the source class must have a reference to the target class.



Move Method Example

A project has a number of participants. The **`participatesIn`** method checks if the receiving person is in the given project.

```
class Project {
  public Person[] getParticipants(){return participants;}
  private Person[] participants;
}

class Person {
  boolean participatesIn(Project proj) {
    for(int i=0; i<proj.getParticipants().length; i++) {
      if (proj.getParticipants()[i].getId() == id)
        return(true);
    }
    return(false);
  }
  public int getId(){return id;}
  private int id;
}
```

Issues with this code

- The **`participatesIn`** method uses data of the **`Project`** class rather than its own. This violates the SRP and is an example of the **feature envy** code smell that we'll see later.

- The **`participatesIn`** method also violates the Law of Demeter, or flaunts with it at least.

These issues are fixed nicely by moving the **`participatesIn`** method to its natural home, the **`Project`** class.

Applying Move Method

After moving the **`participatesIn`** method the code appears as below. Think about how it has been improved.

```
class Project {
  public boolean participatesIn(Person p) {
    for(int i=0; i<participants.length; i++) {
      if (participants[i].getId() == p.getId())
        return(true);
    }
    return(false);
  }
  private Person[] participants;
}
```

`getParticipants` method removed

```
class Person {
  public int getId(){return id;}
  private int id;
}
```

`participatesIn` method removed

Will clients of the **`Person`** class have to be changed as well?

Pull Up Method/Field

Moves a method or a field from a class (or set of sibling classes) to a (common) superclass, thus avoiding the duplication of the method/field in the subclasses.

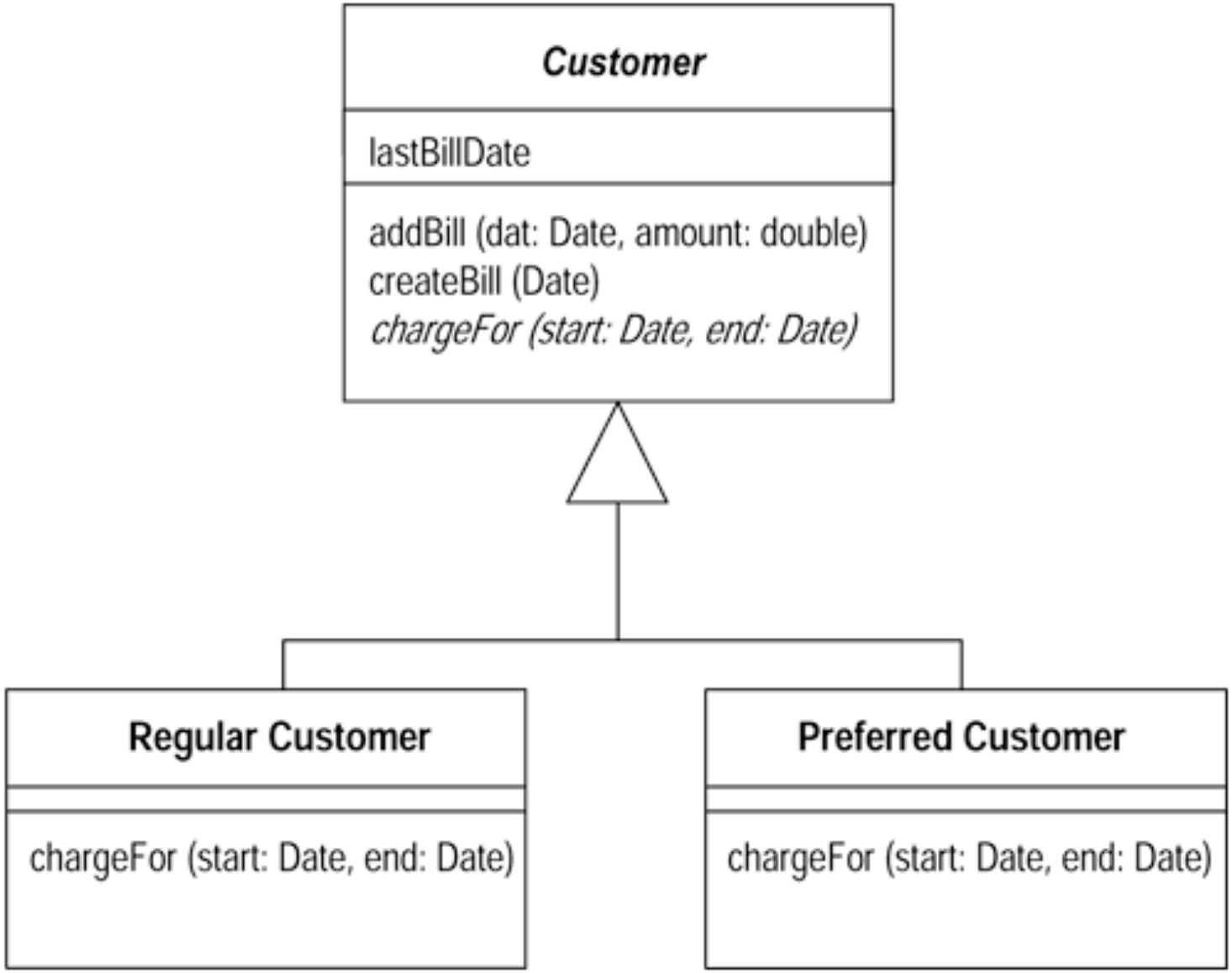
The methods in the sibling classes may be slightly different and need some adjusting prior to being moved.

Pull Up Method may be applied to make a method available in the sibling classes, or simply because it makes more sense in the superclass.

This refactoring is tricky when the method refers to features available in the subclass, but not in the superclass.

How to handle this?

Pull Up Method Example



Push Down Method/Field

Moves a method or a field from a class to one or more immediate subclasses.

Reverse of Pull Up Method/Field.

Apply this refactoring when a method or field is used only in a number of subclasses and it's clearer to move it there.

What principle is being violated?

Extract Local Variable

Assign the result of an expression to a new local variable.

Simplifies a complex expression by dividing it into multiple lines

Useful also if the same expression is evaluated more than once

```
1 package p
2
3 def x = 9
4 def y = 8
5
6 if ((x + y) < (x + y *2)) {
7     print x + y
8 }
```

Rename... ⌥⌘R
Extract Method... ⌥⌘M
Extract Local Variable... ⌥⌘L
Extract Constant...
Pull Up...
Push Down...

```
n = x + y;
if (n < (x + y*2)){
  print n;
}
```

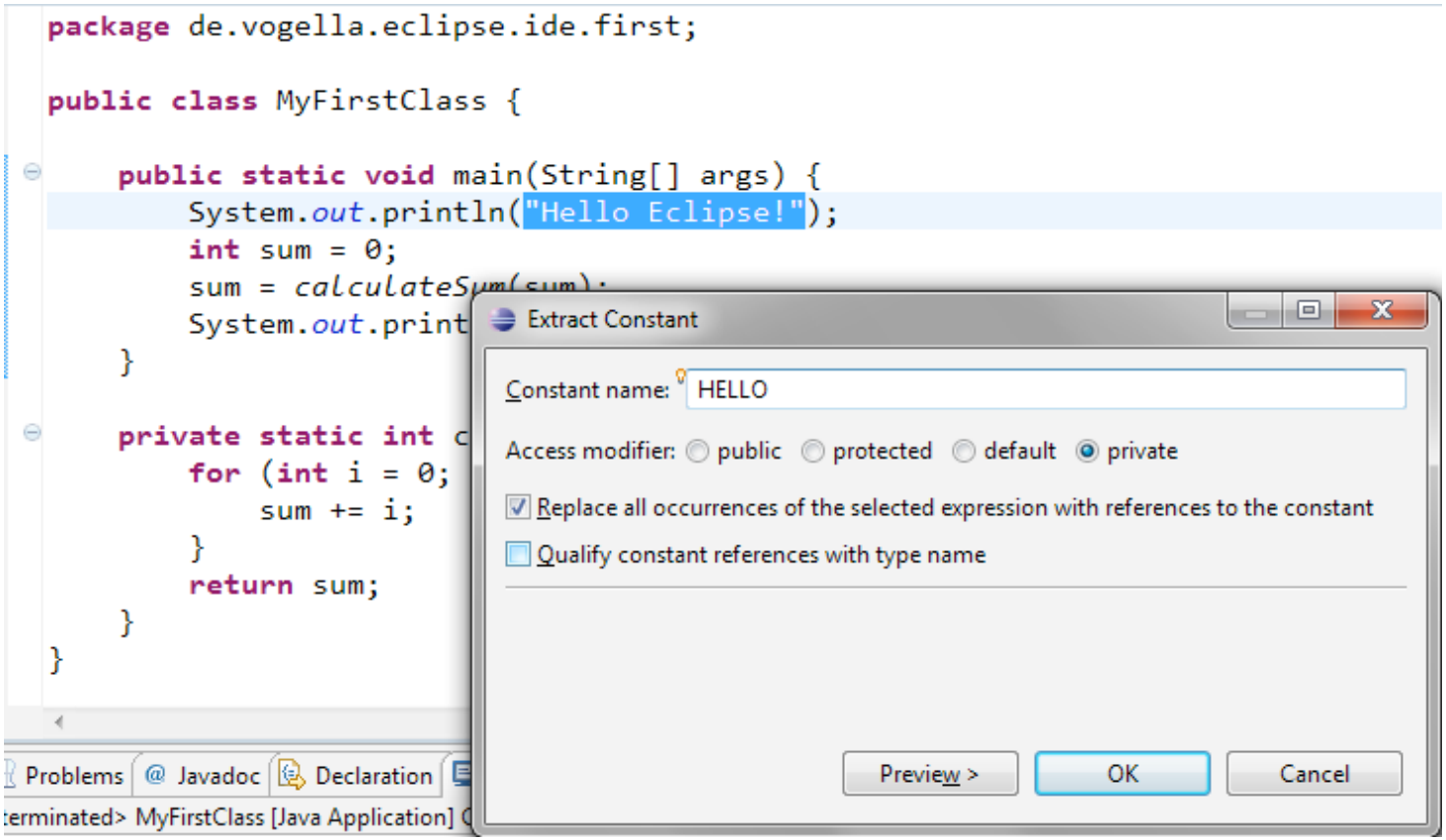
```
sum = x + y;
if (sum < (x + y*2)){
  print sum;
}
```

Extract Constant

Converts a number or string literal to a static final field (Java).

All uses of that number or string literal in the class are updated to use that field, instead of the number or string literal itself

After applying this refactoring, you can modify the constant in just one place.



Convert Local Variable to Field

Converts (promotes) a local variable it to a private field.

(Only use a field where necessary: a field adds complexity to its entire class; a local variable only adds to the complexity of the block it belongs to.)

Apply this refactoring when the need arises for a local variable within a method to be accessed in another method as well.

The inverse, Convert Field to Local Variable, is arguably a more useful refactoring.

Extract Method

Convert a selected block of code into a method.

The block of code is put into its own method and the original block replaced with a call to the newly-created method.

Useful when:

- a method is simply too long
- a piece of code is duplicated across several methods.
- A portion of a method does a coherent task and it's clearer to split this off into its own method.

Extract method example

```
1 package p
2
3 def x = 9
4 def y = 8
5
6 def g = x + y
7 def h = g * 2
8
9 print h
```

Rename... ⌥⌘R
Extract Method... ⌥⌘M
Extract Local Variable... ⌥⌘L
Extract Constant...
Pull Up...
Push Down...

```
def foo(i, j){
  return (i+j)*2;
}

h = foo(x, y)
print h
```

Rename would now be used to give `foo` a better name

Inline Method

Occasionally a method's body is just as clear as its name
- only ever the case for a very short method!

This refactoring replaces calls to the method with the body of the method itself.

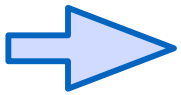
Then the method can be removed.

This refactoring is the reverse of Extract Method

Inline Method Example

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
  return numberOfLateDeliveries > 5;
}
```

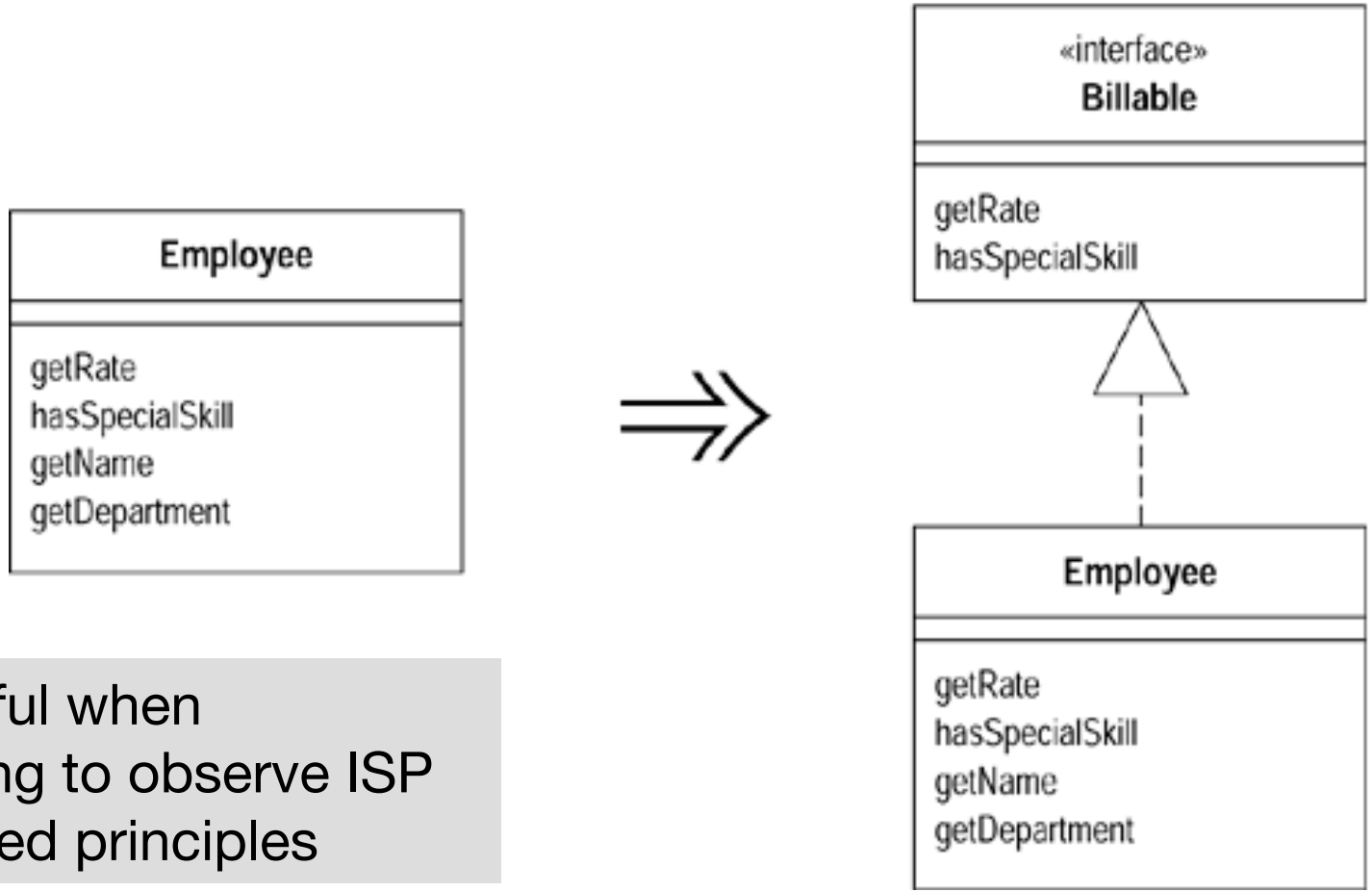


```
int getRating() {
  return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Inlining **`moreThanFiveLateDeliveries`** leads to clearer code.

Extract Interface

- Creates a new interface out of (some of) the methods defined in a class.
  - The class can be updated to **`implement`** this interface
  - References to the class may be updated to references to the interface



Very useful when refactoring to observe ISP and related principles

CODE SMELLS

Code Smells

A code smell is a hint that there's something wrong in your code that maybe should be fixed by refactoring.

Simple example: **Long Method**.
What refactoring comes to mind?

No need to fix all smells (this would be crazy)

* Some "smells" don't indicate problems
* Some problems aren't bothersome, or are not of concern
* Badly designed *stable* code is probably not a concern

Judicious refactoring may be used to resolve the design problem underlying the smell.

Here we look at a number of well-known smells...

Bad Smells in Code

Dozens of code smells have been proposed in the literature.

Here we look at a selection of them.

- **Duplicated Code**
- Long Method
- God Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- **Feature Envy**
- **Primitive Obsession**
- **Speculative Generality**
- Data Class
- **Refused Bequest**

A Really Big Caveat

- Never read smells&refactoring like a medicine bottle
  - “At temperatures over 38.5 degrees, take two spoonfuls every 4 hours.”

- **The smell is only a hint**. They may or may not be anything actually wrong with the code.

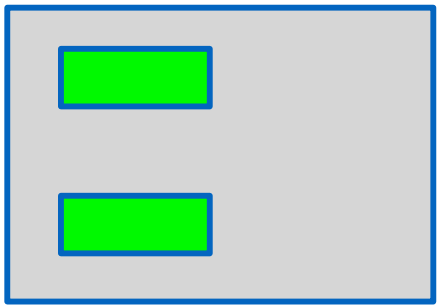
- It may not be worthwhile fixing the problem:
  - e.g. a small, inconsequential piece of code duplication that would require a system overhaul to resolve

- Code smell detection can help in identifying problem areas and in deciding what to do.

Duplicated Code I

If the same code fragment occurs in more than one place, it may be useful refactor to remove the duplication.

The simplest duplicated code problem is when you have the same piece of code in two methods of the same class.

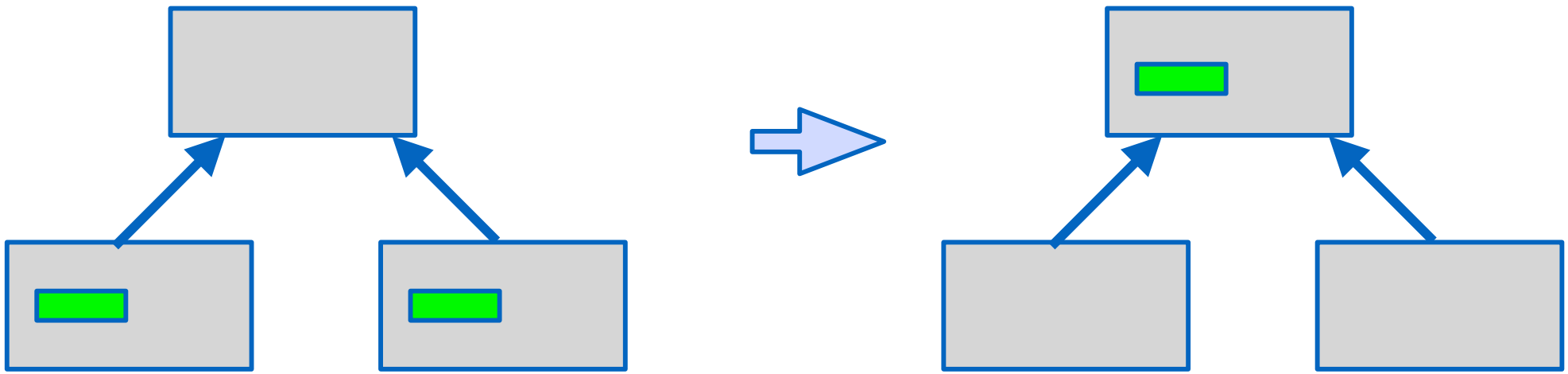


Consider performing **Extract Method** and invoke the new (private) method from both places.

Duplicated Code II

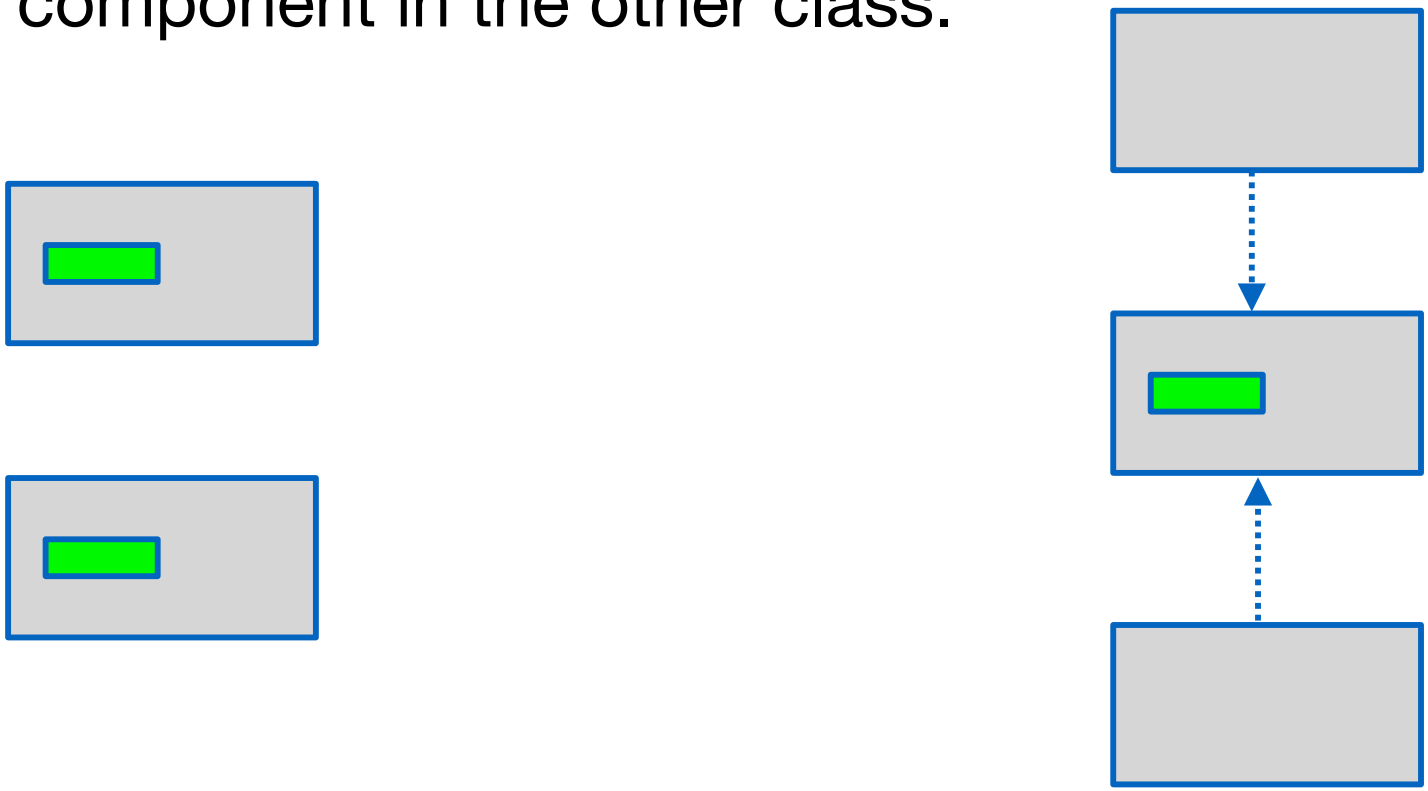
Another common duplication problem is having the same code fragment in two related subclasses.

Consider performing **Extract Method** in both classes, then **Pull Up Method** to the shared superclass



Duplicated Code II

If code is duplicated in two unrelated classes, consider applying **Extract Class**, and then use the new component in the other class.



But ask yourself is it *really* duplicated code? And is it worth fixing?

Don’t overdo it!

One developer’s complaint about being over-zealous about removing duplicated code:

“I have seen duplicate code checkers complain about JUnit test cases being too similar. And yes, you probably could have extracted two lines of code into a private method so the three test cases with those lines could share it, but then to describe those two trivial lines of code you would need to spend a month of Sundays trying to come up with a sensible name, and this is somewhat missing the point of refactoring.”

Long Method

- The longer a method is, the more difficult it is likely to be to understand.

- What's long? Opinions vary! Thresholds from two authors
  - *A dozen lines* -- Martin Fowler
  - *200 lines* -- Steve McConnell

- There is little value in seeking a specific threshold
  - A Long Method is simply one that is too long and should be split

- **Extract Method** is usually key to resolving this smell.

God Class

- A "God Class" is one that knows too much about the rest of the system and/or has too much responsibility
- Is usually a class with too many fields and methods.
- A God Class may also be a source of code duplication
- **Extract Class** and **Extract Subclass** may help

Switch Statements I

A switch or decision statement that affects method behaviour should probably be replaced with polymorphism

```
enum StudentType {UNDERGRAD,POSTGRAD};

class Student {
   ...
   if (studentType == UNDERGRAD){
      ...
   }
   else {
      ...
   }
   ...
   private StudentType studentType;
}
```

A better design is probably two **`Student`** subclasses called **`Undergraduate`** and **`Postgraduate`**.

Switch Statements II

Use Extract Method to extract the relevant code and then Move Method to get code to the appropriate class.

In the example, all code related to how undergraduates are handled should be moved to the **`Undergraduate`** class.

If you only have a few cases in a single method then polymorphism is probably overkill.

If one of your conditional cases is null, consider applying the **Null Object** Design Pattern.

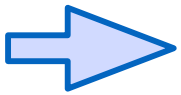
Long Parameter List

Long parameter lists are hard to understand and maintain.

A method needn't be passed everything it needs in the parameter list
* Pass in enough so it can get everything it needs
* (keeping the Law of Demeter in mind)

Consider using **Replace Parameter with Method Call**, where you replace one parameter with a request to an object you already know about, e.g.

```
int basePrice = quantity * itemPrice;
discountLevel = getDiscountLevel();
finalPrice = discountedPrice(basePrice, discountLevel);
```



```
int basePrice = quantity * itemPrice;
finalPrice = discountedPrice(basePrice);
```

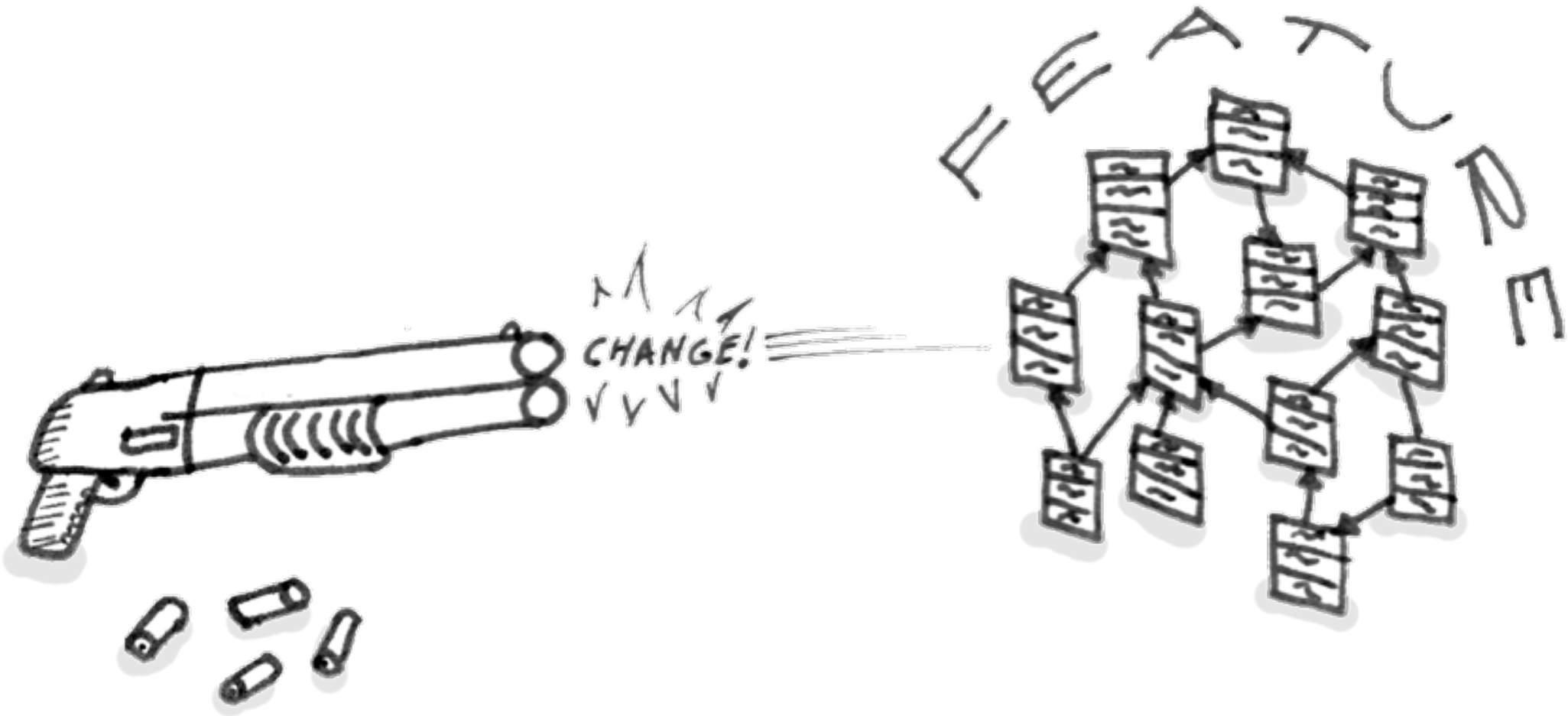
Divergent Change

- Divergent change occurs when one class is commonly changed in different ways for different reasons

- E.g. in a class in an invoicing application you have to change one set of methods every time the invoice format changes, and another set of methods whenever a new user is added...
  - Suggests a violation of the SRP
  - This class is probably better modelled as two classes

- To clean this up, identify everything that changes for a particular reason and use **Extract Class** to put them all together
  - This may or may not be easy!

Shotgun Surgery



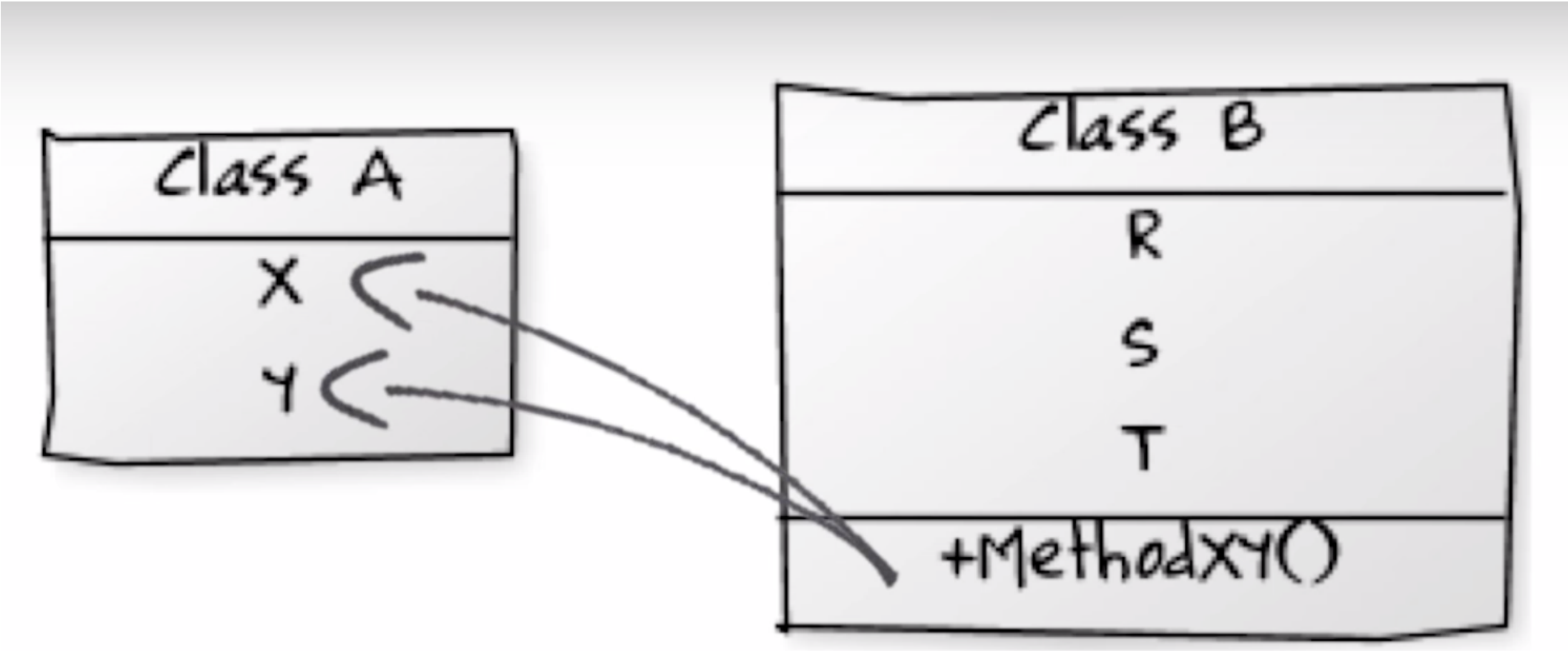
Shotgun Surgery

- If, whenever you change a certain feature, you make changes to different classes **=> shotgun surgery**
  - it's easy to miss an important change, which makes maintenance very challenging.
  - Opposite of Divergent Change in a sense, but also suggests that the SRP is being violated

- Consider using Move Method and Move Field to put all the change-prone elements in a single class
  - Recall 'separate the stable and the unstable' guideline

- If no current class is suitable then it may be best to create a new class to bring the relevant behaviour together

Feature Envy



Feature Envy

- Feature Envy occurs when method is more interested in a class other in the one that it is in
- Usual solution is to use **Move Method** to move it to its appropriate class
- If only part of the method suffers from envy then consider using **Extract Method** on the envious part and **Move Method** to move it to its rightful home.
- There are several **design patterns** that appear envious by design, e.g. **Strategy**
  - this is an example where a code smell does not indicate any underlying problem — quite the opposite.

Primitive Obsession

- Java has primitives for integers, floats, boolean etc.
- Primitive Obsession occurs when a primitive type is used where a class should have been preferred.
  - e.g. using a String for a phone number, instead of creating a proper PhoneNumber class.
- Don’t be reluctant to use “small” classes to model simple abstractions, e.g.
  - an amount in a currency,
  - ranges with an upper and lower bound,
  - special strings such as a student number.
- Such classes can make your code much clearer

Speculative Generality

- This is when a developer implements functionality that they speculate may be needed someday, but isn't currently required.
- One clear sign of Speculative Generality is when the only users of a class or method are the test cases. Here are some of the refactorings you might apply:
  - If you have abstract classes that are not doing enough then use **Collapse Hierarchy**.
  - Unnecessary delegation can be removed with **Inline Class**.
  - Methods with unused parameters can be fixed with **Remove Parameter**.
  - Methods named with odd abstract names can be repaired with **Rename Method**.

Data Class

- A data class has fields, getters and setters, and little or nothing else
  - Any interesting functionality it has is implemented by other classes, and often duplicated

- Use **Remove Setting Method** on any field that should not be changed

- Look where these getters and setters are used by other classes and try to use **Move Method** to move behaviour into the data class

- If you can't move a whole method, use **Extract Method** to create a method that can be moved

Refused Bequest

- A **Refused Bequest** is a method a class inherits from its superclass but doesn't use it
- If this happens a lot, the class hierarchy has been badly designed and needs to be dismantled and redesigned.
- Rejecting a **public** method is normally a serious matter
  - Polymorphic replacement is no longer possible. Also violates the Liskov principle.
- **Push Down Method/Field** may help here, or **Replace Inheritance with Delegation**.

Refactoring Challenges I

- **Databases**
  - An application may be tightly coupled its database schema.
  - Code refactoring may require that the database schema be changed as well, so refactoring enterprise database applications is a challenge.

- **Changing Interfaces**
  - If refactoring involves changing class interfaces, then other classes that use these interfaces must be changed as well.

Refactoring Challenges II

- **Test cases**
  - If interfaces are changed during refactoring, test cases will need to be refactored as well.

- **Radical refactoring is hard**
  - A large code overhaul is challenging, but sometimes necessary

Refactoring and Performance

- How does refactoring affect performance?
- Refactoring will typically make software run more slowly
  - This rarely matters!
  - Refactored software is more amenable to performance tuning
- Profiling ensures that you focus your tuning efforts in the right places
  - e.g. minimising database access is usually far more significant than optimising in-memory processing

Summary

- **Refactoring** is the process of improving the design of a program, without changing its behaviour. It's a vital part of the software development process

- Individual refactorings have been named and automated versions are available in most IDEs
  - Rename, Move, etc.

- **Code Smells** are a hint that something may be wrong in your code.
  - Many smells have been identified and named. Tools exist to detect them (e.g. JDeodorant, Findbugs).
  - Refactoring can be used to resolve code smells, but don't do this blindly

Introduction to Design Patterns

Mel Ó Cinnéide
School of Computer Science
University College Dublin

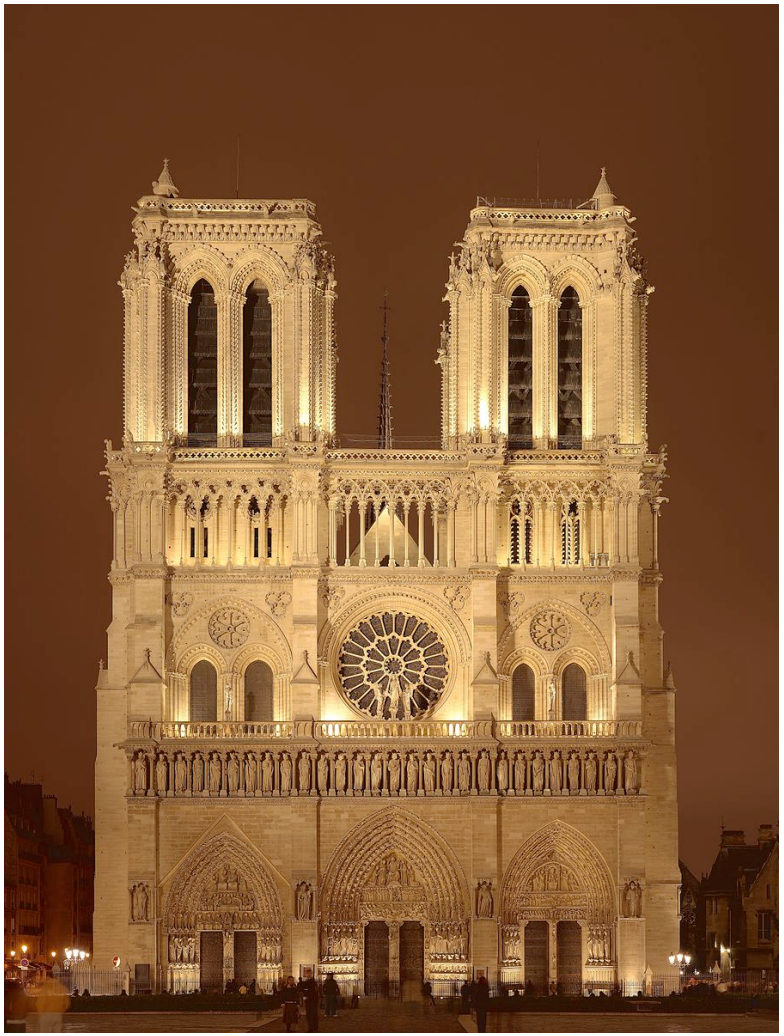
Roadmap of this Section

- Christopher Alexander's Pattern work
- Patterns in Software
- Pattern Form
- Intro to the "Gang of Four" Patterns
- Summary

Where Patterns Started

- In the 1960s, an architect called Christopher Alexander revisited some age-old questions:
  - What is good architecture?
  - Is beauty objective?
  - Or it simply subjective?

Beauty, you decide!



Where Patterns Started

- Alexander studied master solutions to the same architectural problems.
- The similarities he found, he termed **patterns**.
- Examples: Main entrance, alcove, ... corner store, ...

So, what's a pattern?

- A pattern is a **proven solution** to a **recurring problem** in a **certain context**.
- Or, in Alexander's own words:
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

A pattern is a **proven solution**...

- The solution isn't new, it's already been proven to be valid.
  - Usually many examples already exist
- A novel, brilliant idea is not a pattern.
- To someone who has mastered the field, a pattern should be nothing new.

E.g. the **Loop** pattern will contain little new for an experienced programmer

...to a **recurring problem**...

- The problem isn't a one-off.
- It's a problem that you encounter again and again (but a bit differently each time).

E.g. instances of the **Roundabout** pattern are similar, yet never quite the same.

...in a **certain context**.

- A pattern doesn’t solve a problem in a vacuum.
- The pattern has to fit into some **context**.
- The context varies, and this changes how the pattern is applied.

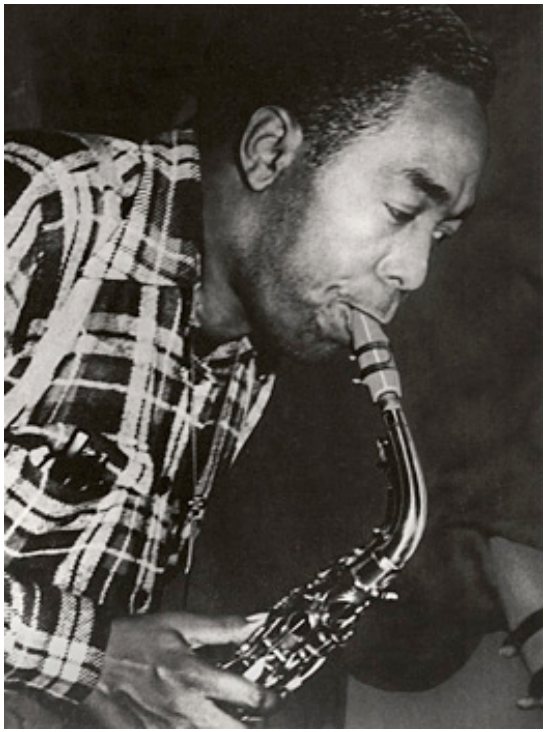
This "tailoring to context" is one of the challenges in using patterns.

Once mastered, patterns are forgotten

“Learn the patterns, then...

forget 'em.”

(“forget 'em” really means to *embody* them.)



Charlie Parker
Jazz musician

Alexander and Software

- Alexander had some following in the architecture community.
- However, his work was lionised by the software community in the 1990s.
- Patterns became, and have remained, an established part of software development.

Key properties of patterns

- Sometimes it is not clear what is and isn't a pattern. Some guidelines:
  - More than just a clever trick.
  - Patterns can't be applied automatically – they need to be tailored to fit their context. They are "**half-baked**".
  - A pattern should be THE solution, and provide you with a warm, fuzzy feeling (what patterns people call the Quality Without A Name, or QWAN)
  - Patterns are not invented: a pattern shouldn't look surprising to an expert in the domain.

Pattern Form

- There are various accepted forms with which to describe a pattern, but five essential parts of any form are:
  - Name
  - Problem
  - Context
  - Solution
  - Consequences
- We examine these in the following slides.

Pattern Form: Name

- Pattern name may seem like a trivial issue. However it has a significant impact:
  - Increases our design vocabulary
  - Enables discussion at higher level of abstraction
- Comment from an experienced programmer on reading the GoF book:
  - “The exciting part of patterns is the names”

Pattern Form: Problem

- Describes when to apply the pattern
  - Explains the type of problem the pattern solves.
- Could be a design problem
  - E.g. how to represent an algorithm as an object?
- Or an inflexible design
  - E.g. how can subclass explosion be ameliorated?

Pattern Form: Context

- No problem exists in a vacuum — there is always a surrounding context
- Context helps you determine where to use the pattern, and provides evidence that it is of general application
  - Also called the constraints or **forces**
- Usually there are some forces in conflict; that is why a pattern is required to resolve them.
  - E.g. optimising for performance and maintainability is usually impossible — these two forces tend to be in conflict.

Pattern Form: Solution

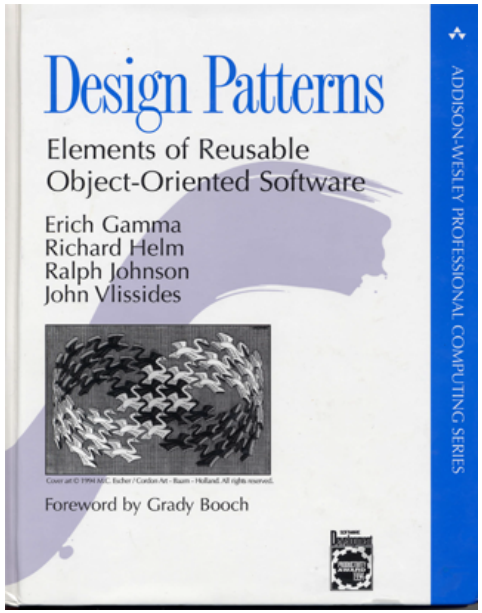
- This is a description of the elements of the solution design, their responsibilities, relationships and collaborations.
- It is not a specific implementation, but a general, tailorable solution.
- The solution should resolve the forces described in the Pattern Context in a “satisfying” way.

Pattern Form: Consequences

- These are the results and trade-offs of applying the pattern.
- Include the pros and cons of applying the pattern.
  - Design patterns usually increase flexibility
  - Complexity usually increases
  - Performance may decrease
- Understanding the consequences of using a design pattern is vital.
  - Comes close to the essence of software design

“Gang of Four” Patterns

- In the early 1990s software developers wondered if the patterns approach could be used in our discipline
  - Are there problems that occur over and over again that could be solved in essentially the same way?
- The so-called “Gang Of Four”, Gamma, Helm, Vlissides and Johnson produced one of the best selling books ever in software.



GoF Patterns

	Purpose		
	Creational	**Structural**	**Behavioral**
	Factory Method	Adapter (class)	Interpreter Template Method
	Abstract Factory Builder Prototype Singleton	Adapter(obj) Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

GoF Patterns

- Although the GoF design patterns are described separately, they are interrelated.
- They are not however intended to be complete – it is not a **pattern language**.

Summary

- We looked at various aspects of patterns and the patterns movement from Alexander to software.
- We will now look further at a few GoF design patterns.
- After excessive hype in the 1990s, patterns have simply become an established part of software development.

The Strategy Pattern

Strategy

Intent

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

So consider using Strategy if a class should have multiple ways of performing some task.

Simple Commuter Example

A commuter needs to travel to the airport.

They might travel by car, bus or taxi.

How best to model this?

Tell the method what to do — an ugly solution

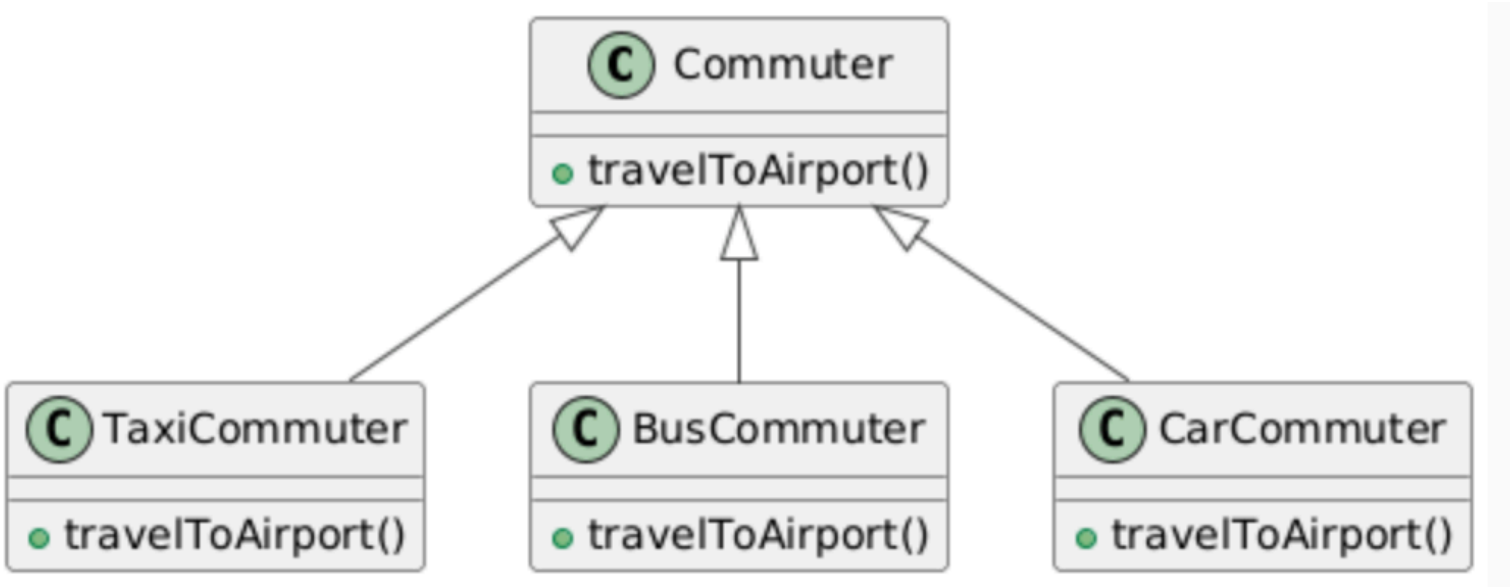
```
public void travelToAirport(String mode) {
  if mode == "car" then
    ... // travel by car
  else if mode == "bus" then
    ... // travel by bus
  else if mode == "taxi"  then
    ... // travel by taxi
}
```

Switching on type is a **code smell**

Method has three unrelated parts.

Gets worse if other methods in the class type check on the `mode` argument as well.

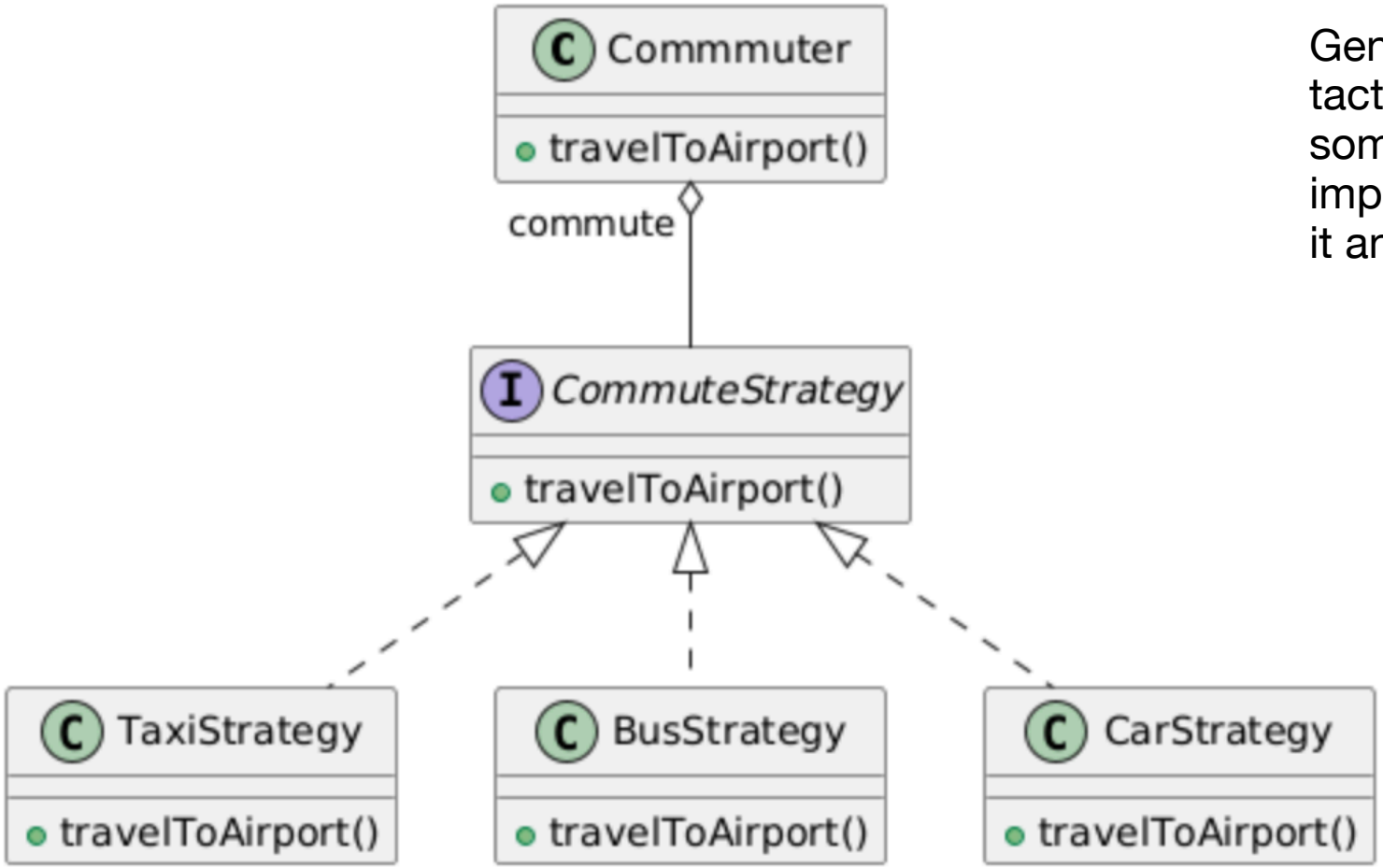
Using inheritance — not a good idea either



Subclassing because just one method is different — 👎.

Also, a commuter may change their mode of transport dynamically, but this model doesn't facilitate this.

Applying the Strategy pattern



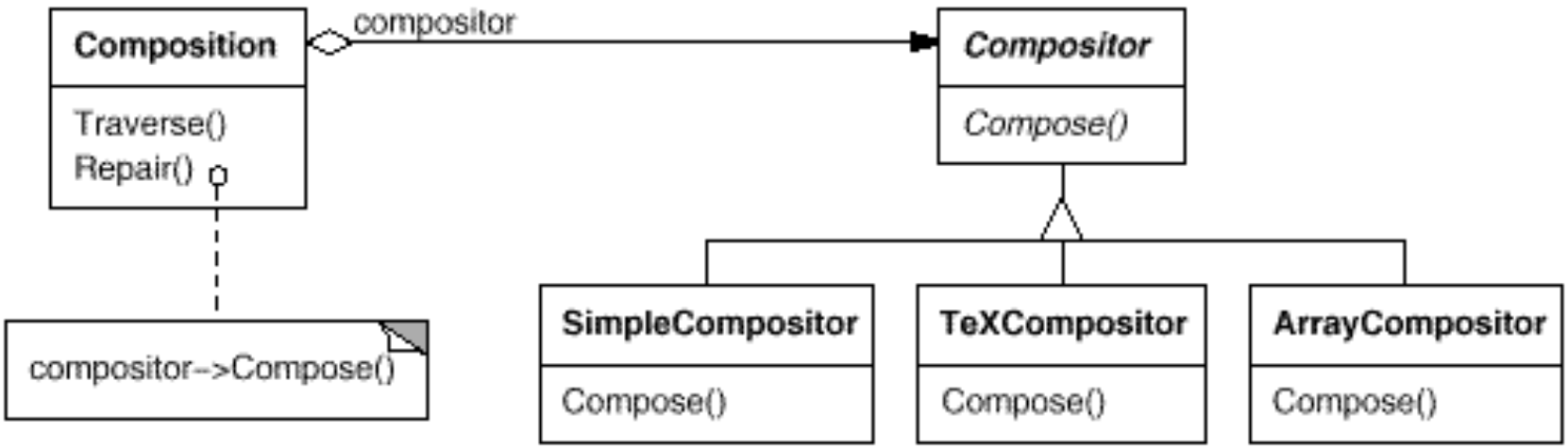
Generalisable tactic: when something is important, make it an object.

`Commuter` delegates to its `CommuteStrategy`:

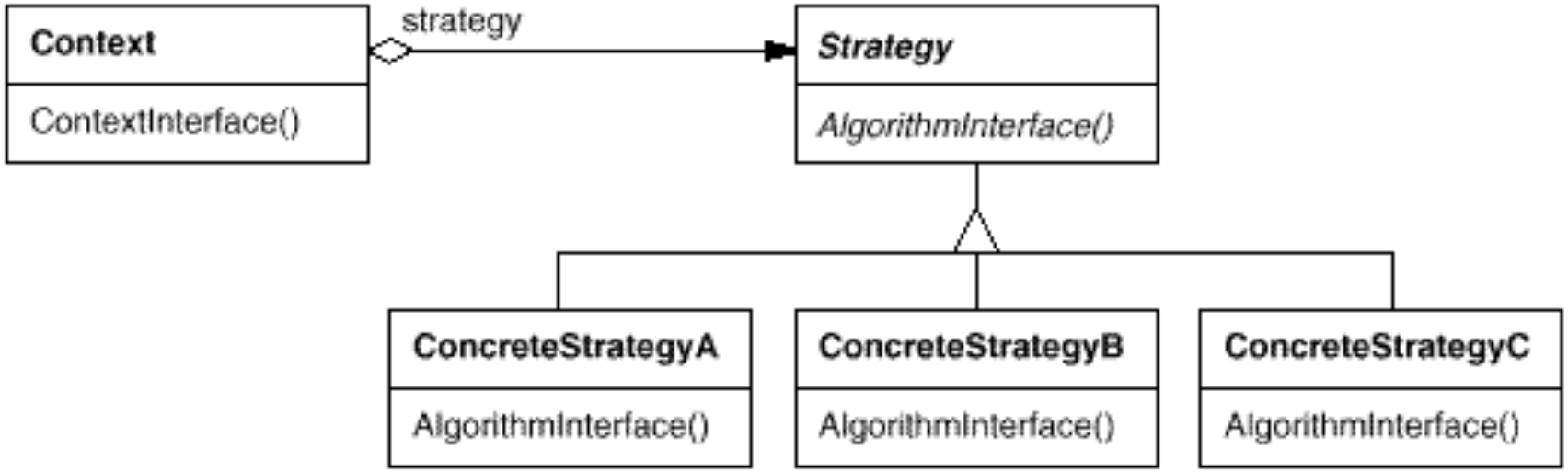
```
public void travelToAirport() {
  commute.travelToAirport();
}
```

Another Example -- GoF Motivating Example

Many algorithms exist for breaking a stream of text into lines. How can we configure an application to dynamically choose which one to use?



Strategy -- Typical Structure



How does the Strategy class access its Context?

```
class Commuter {
  public void travelToAirport(){
    commute.travelToAirport();
  }
  private CommuteStrategy commute;
  private Wallet myWallet;
}

class BusStrategy implements CommuteStrategy {
  public void travelToAirport(){
    ...
  }
  private payDriver(){
    // needs to access myWallet, but how?
  }
}
```

Option (1): Pass it as an argument

```
class Commuter {
  public void travelToAirport(){
    commute.travelToAirport(myWallet);
  }
  private CommuteStrategy commute;
  private Wallet myWallet;
}

class BusStrategy implements CommuteStrategy {
  public void travelToAirport(Wallet wallet){
    ...
  }
  private payDriver(Wallet wallet){
    wallet.pay(...);
  }
}
```

Option (2): Pass reference to the Context object

```
class Commuter {

  public void travelToAirport(){
    commute.travelToAirport(this);
  }

  public void payDriver(){
    ...
  }

  private CommuteStrategy commute;
  private Wallet myWallet;
}
```

An additional public method has to be added

Option (2): Pass reference to the Context object

```
class BusStrategy implements CommuteStrategy {

  public void travelToAirport(Commuter commuter){
    this.commuter = commuter;
    ...
  }

  private payDriver(){
    ...
    commuter.payDriver();
    ...
  }

  private Commuter commuter;
}
```

Strategy -- Applicability

Consider using Strategy whenever:

- Several related classes differ only in their behaviour.
- A class needs several variants of an algorithm.
- An algorithm uses data that clients shouldn't know about. Use Strategy to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviours, and these appear as multiple conditional statements in its methods.
  - Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy -- A Functional Implementation

Strategy can also be implemented using an **anonymous function**.

In the example above, `Commuter` would simply take a `commute` function as argument, and invoke this function where needed.

This is very easy, and leads some to claim that Strategy is no longer a valid pattern, more a programming idiom.

However, if the strategy is more complicated, using the class-based approach makes more sense.

Strategy -- Consequences

Provides an alternative to subclassing the Context class to create a variety of algorithms or behaviours.

Eliminates repeated conditional statements.

Provides a choice of implementations for the same behaviour.

Clients must be aware of different Strategies.

May increase the number of objects in the system, but Strategies tend to be stateless and so a single instance can be shared (the **Flyweight** pattern).

All algorithms must provide the same Strategy interface.

Strategy -- Comments

Related algorithms are grouped, and **Template Method** (covered in COMP30950) can be used to capture their commonality.

Strategy can be implemented as a stateless object, so it can be shared by several contexts (**Flyweight**).

Strategy is a very common pattern, e.g. Java.util.zip uses Strategy to enable two algorithms be used for performing a checksum on a stream: Adler32 and CRC32.

The State Pattern

Mel Ó Cinnéide
School of Computer Science
University College Dublin

The State Pattern

Intent

Allow an object to alter its behaviour when its internal state changes. The object will *appear* to change class.

In most cases, objects of a class respond to the same message in a similar way, e.g. `calculateNetPrice` will do a similar calculation for any product it is applied to.

Sometimes the same message will elicit a fundamentally different response, depending on the **state** of the receiver.

e.g. how a car responds to `pressAccelerator` depends on whether the engine is running or not.

State — Motivating Example

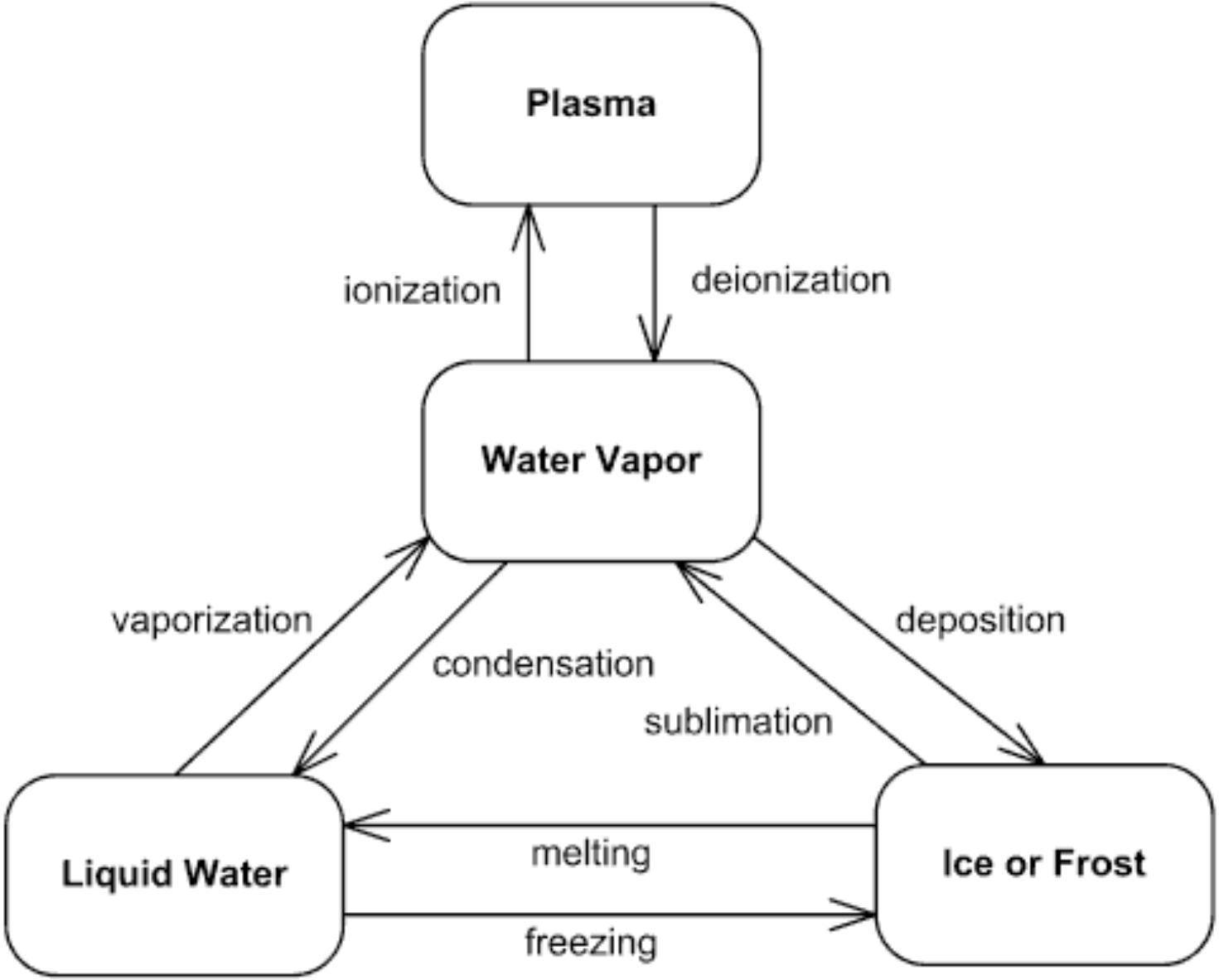
Consider modelling H_20 as a class, and how these methods might be implemented:

```
pour
mix(other: Chemical)
freeze
melt
```

We'll consider just two states: water and ice.

Water can be poured, but not ice. Ice can be melted, but water can't, etc.

States of Water



Coding the H2O class

```
public void pour(){
  if (state == WATER)
    // pour it
  else
    // do nothing
}

public void freeze(){
  if (state == ICE)
    // do nothing
  else
    state = ICE;
}

private H2OState state;
```

Each method checks and may change the state

Hard to understand each state individually

Hard to understand state transitions

Adding/removing states will be hard work!

Applying the State Pattern

Each state is represented by a class, so we have classes for **`Water`** and **`Ice`**.

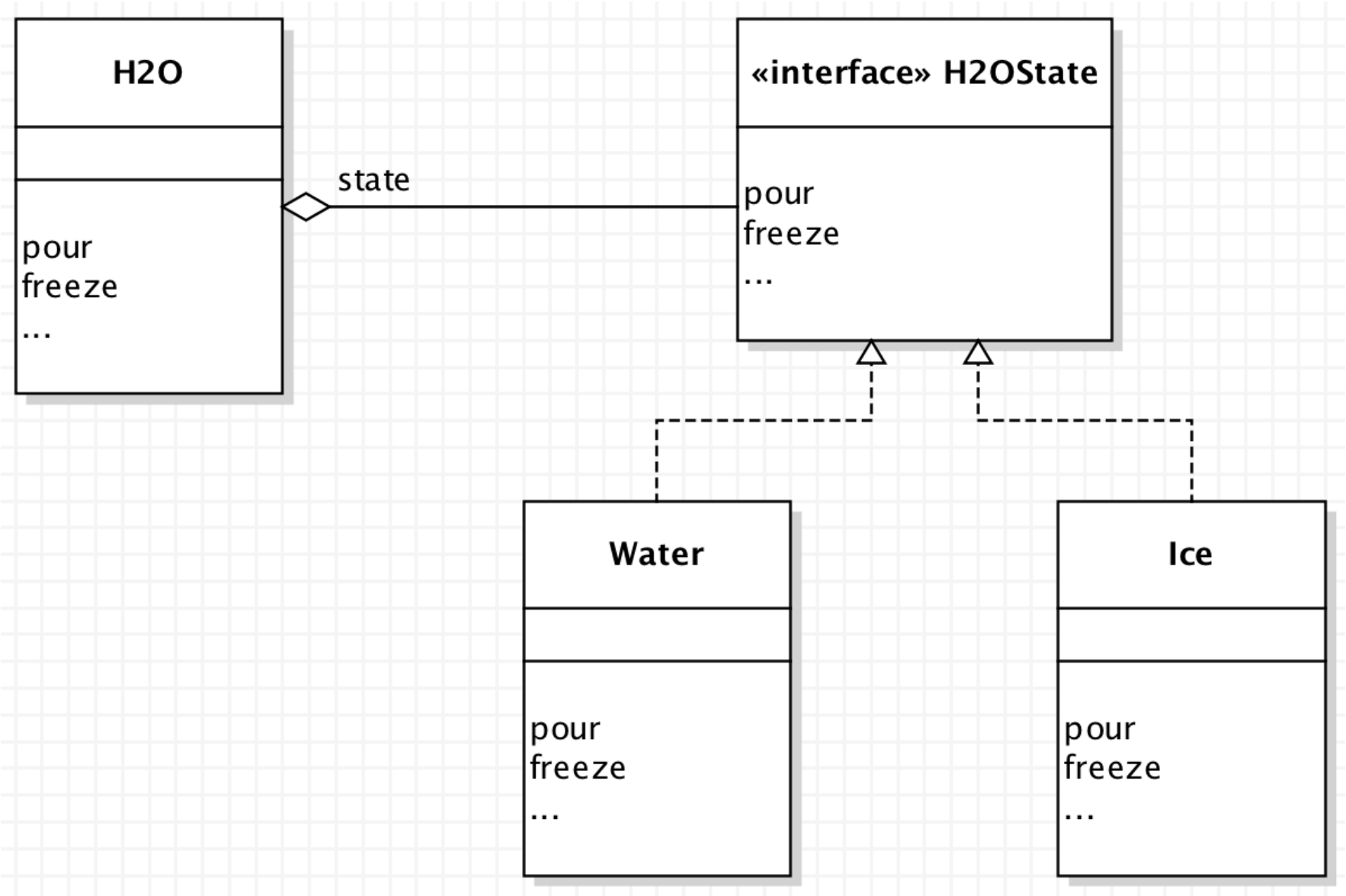
In these classes, the methods (pour, freeze etc.) are implemented but only for the state the class represents.

`Water` and **`Ice`** have the same interface, which is placed in the **`H2OState`** interface.

The **`H2O`** class is given a reference to a **`H2OState`** object, and delegates any messages to this object.

State transitions are achieved by assigning the **`H2OState`** field a different state object.

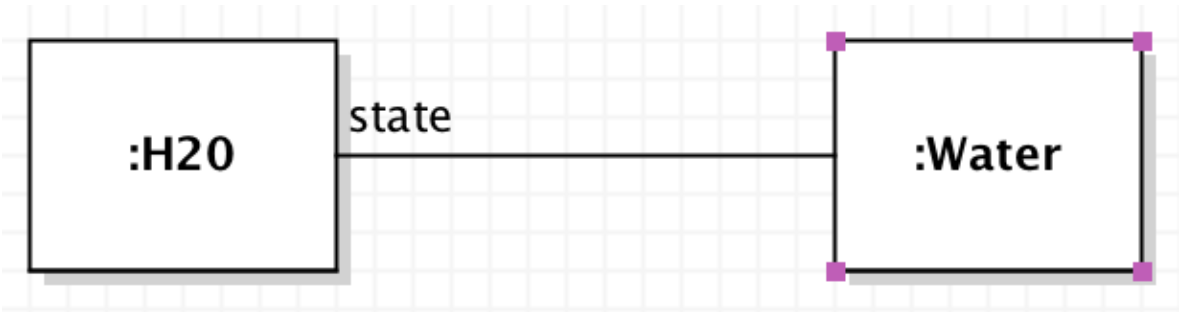
Structure of the State Pattern Solution



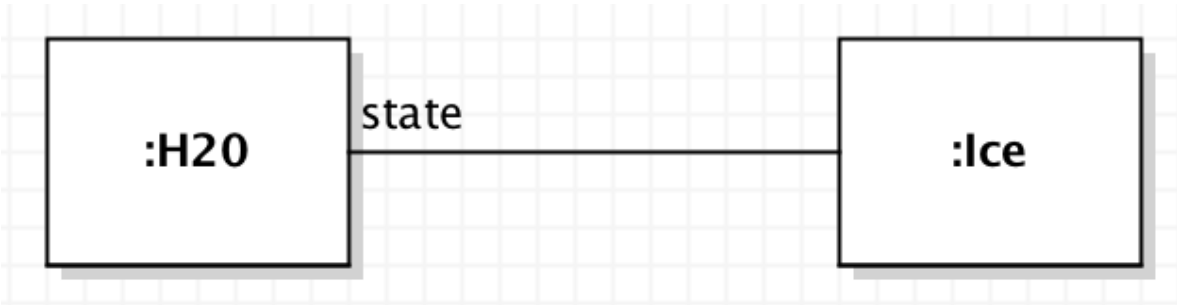
State Object Structure

Each **`H2O`** object now has a reference to its state object.

Water



Ice



Sample Java Code from this solution

The **`H2O`** class delegates everything to its **`H2OState`** object.

```
class H2O {
  public void H2O(){
    state = new Water();
  }

  public void pour(){
    state.pour();
  }

  H2OState state;
}
```

Methods whose implementations depend on the object state are delegated to the `H2OState` object.

Sample Code from this solution

Each state class implements the methods appropriately for its state, e.g.

```
class Water {

  public void pour(){
    // perform pouring
  }

  public void freeze(){
    // change state to ice
  }

  public void thaw(){
  }

}
```

See next slide on **state transitions**

Water is already thawed.

Where are state transitions handled?

There are three possibilities:

1. In the Context class

2. In the State class.

3. In a separate state transition object

We look at these in the following slides.

(1) State transitions in the Context class

Here the Context explicitly encodes the state transitions.

For example, when water freezes, the `H2O` object changes its state reference to refer to an `Ice` object.

Pros:

+ easy to code

Cons:

- context is cluttered with state transitions, and knows about all the states
- state transition code is decentralised

This approach works best if the states and state transitions are fixed.

(2) State transitions in the State classes

Here each state specifies its successor state, and when to make the transition.

For example, if a `Water` object is asked to freeze, it informs the `H2O` object to set its state object to `Ice`.

Pros:

+ context is decoupled from the states
+ adding a new state is easy (no need to change the context)

Cons:

- each state knows about at least one other state (unless a terminal state)
- Context interface must allow state object to be changed
- state transition code is still decentralised

(3) State transitions in a separate object

The state transitions are stored in a separate object, which typically contains a table that maps each (state, event) pair to the next state, e.g.

```
(water, freeze) => ice
```

Suitable for complex state transitions, or when changing state transition rules at runtime is necessary.

Pros:

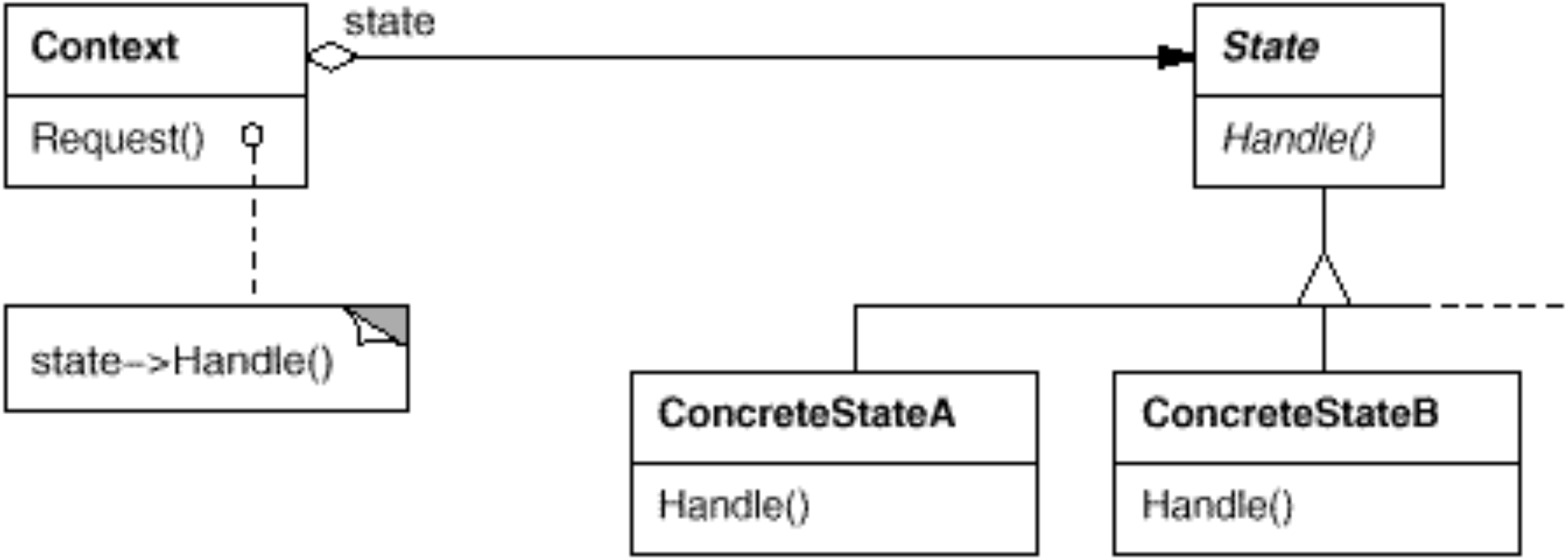
+ state transitions are centralised
+ state transitions can be changed dynamically

Cons:

- complex to code
- transition criteria are encoded in the table => less explicit

State -- Structure

Typical class diagram for State:



State -- Comments

State-transition code can be placed either in Context or in State. Most flexible approach is to make this table-driven in a separate object.

States can be created upfront, or created as needed.

A state object is usually stateless (may sound odd, but think about it).

- Many contexts can share the one state object, i.e. the state object is also a **Flyweight.**
- **Singleton** may be useful here too.

State -- Comments

State and Strategy may have a similar structure, but their **intent**s are different.

As with Strategy, the states will likely need to access their Context object. Same solutions are possible.

Helps with SRP and OCP. Tends to create additional coupling between State classes and Context.

Using the state pattern for a simple case like a regular light switch would be over-engineering.

The Observer Pattern

Mel Ó Cinnéide
School of Computer Science
University College Dublin

Introduction to the Observer Pattern

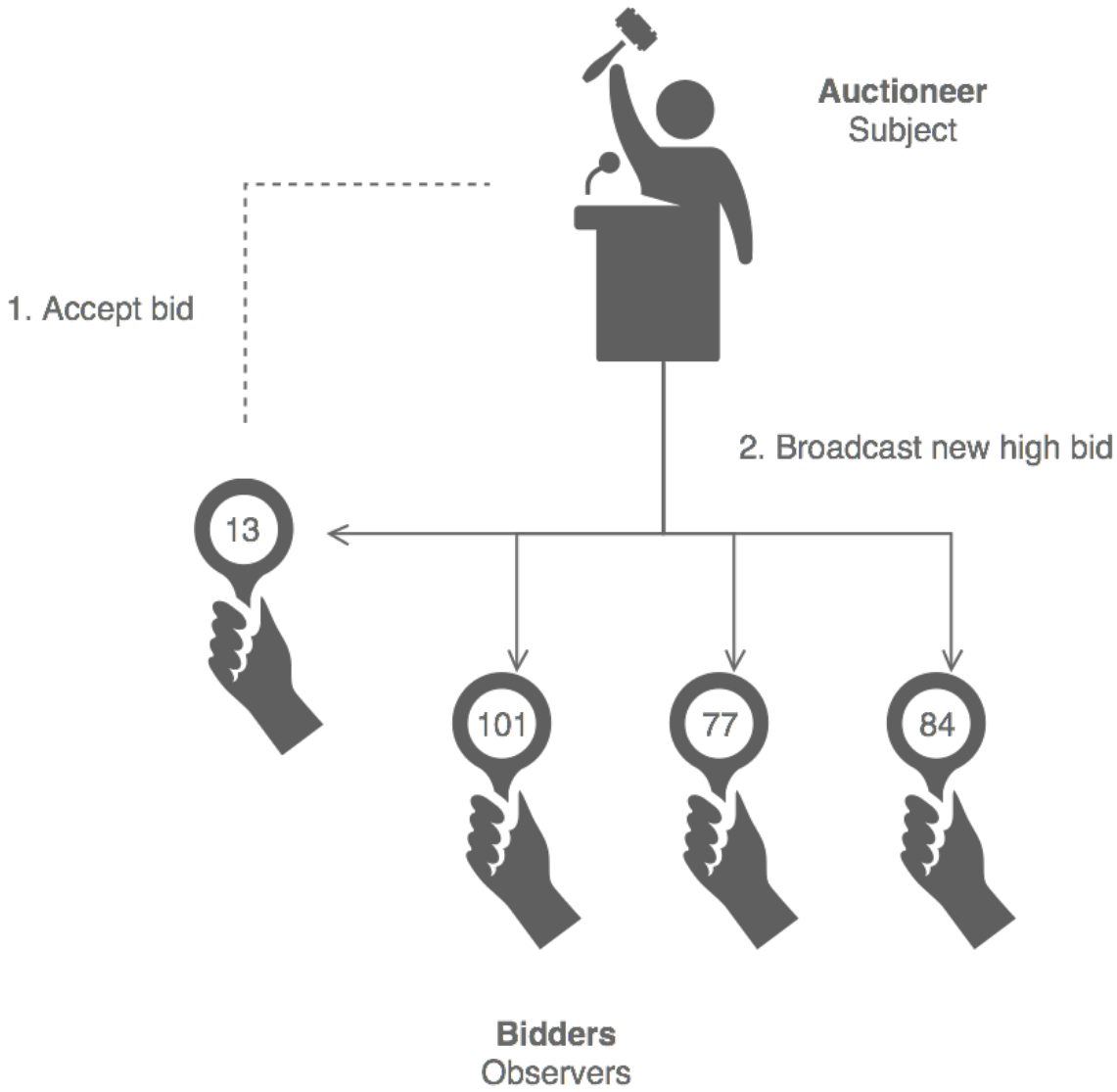
Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependants are notified and updated accordingly.

Examples

- Examples of this pattern abound, both in the real world and in the software domain.
- To motivate this pattern, we consider a real-world example.

Real World Example

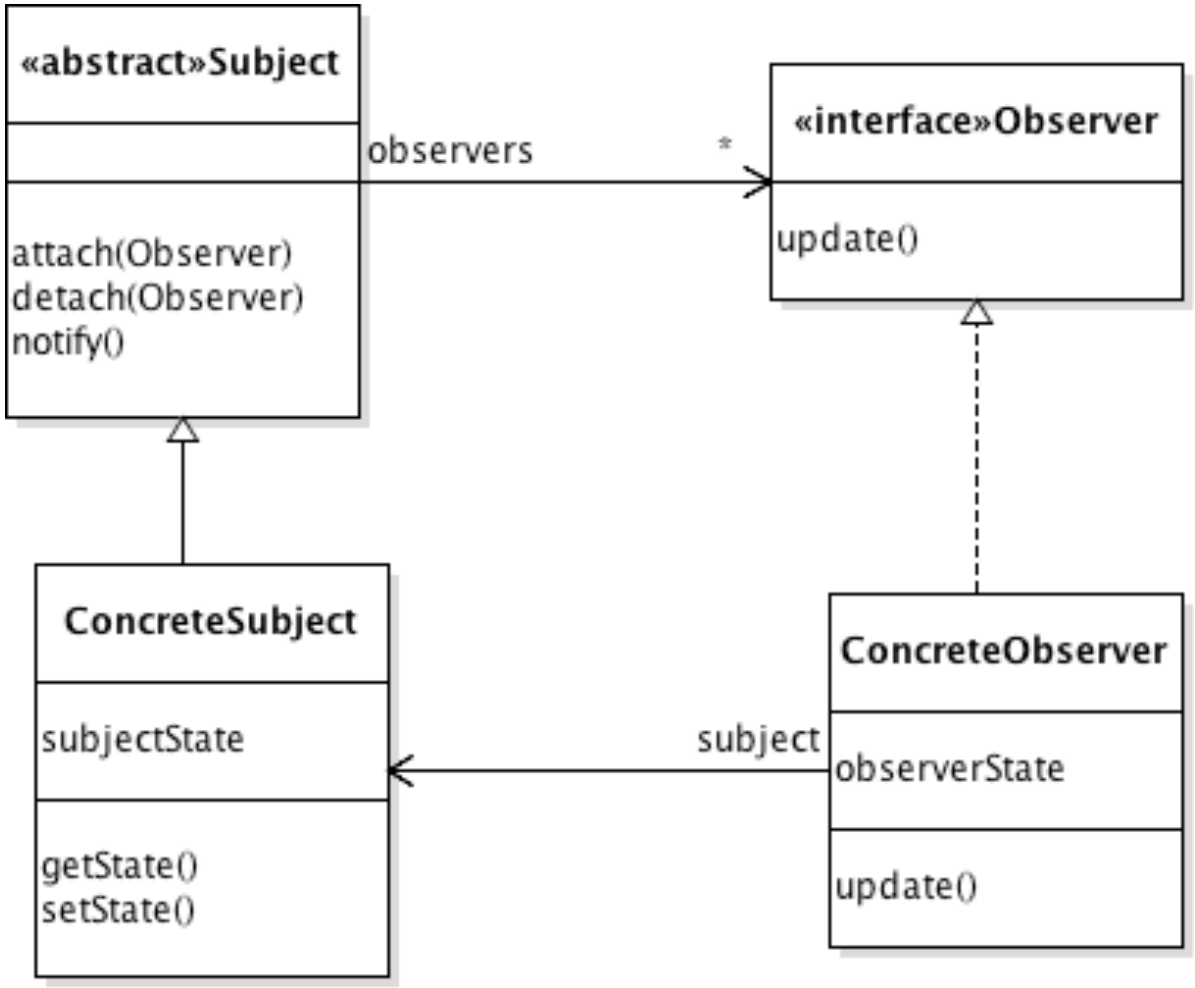


Applicability

Use the Observer pattern when:

- An abstraction has two aspects, one dependent on the other, and it is necessary to model them as separate objects;
- A change to one object requires changing others, and you don't know how many objects need to be changed;
- An object should be able to notify other objects, without making assumptions about who those objects are (loose coupling).

Observer *typical* class structure

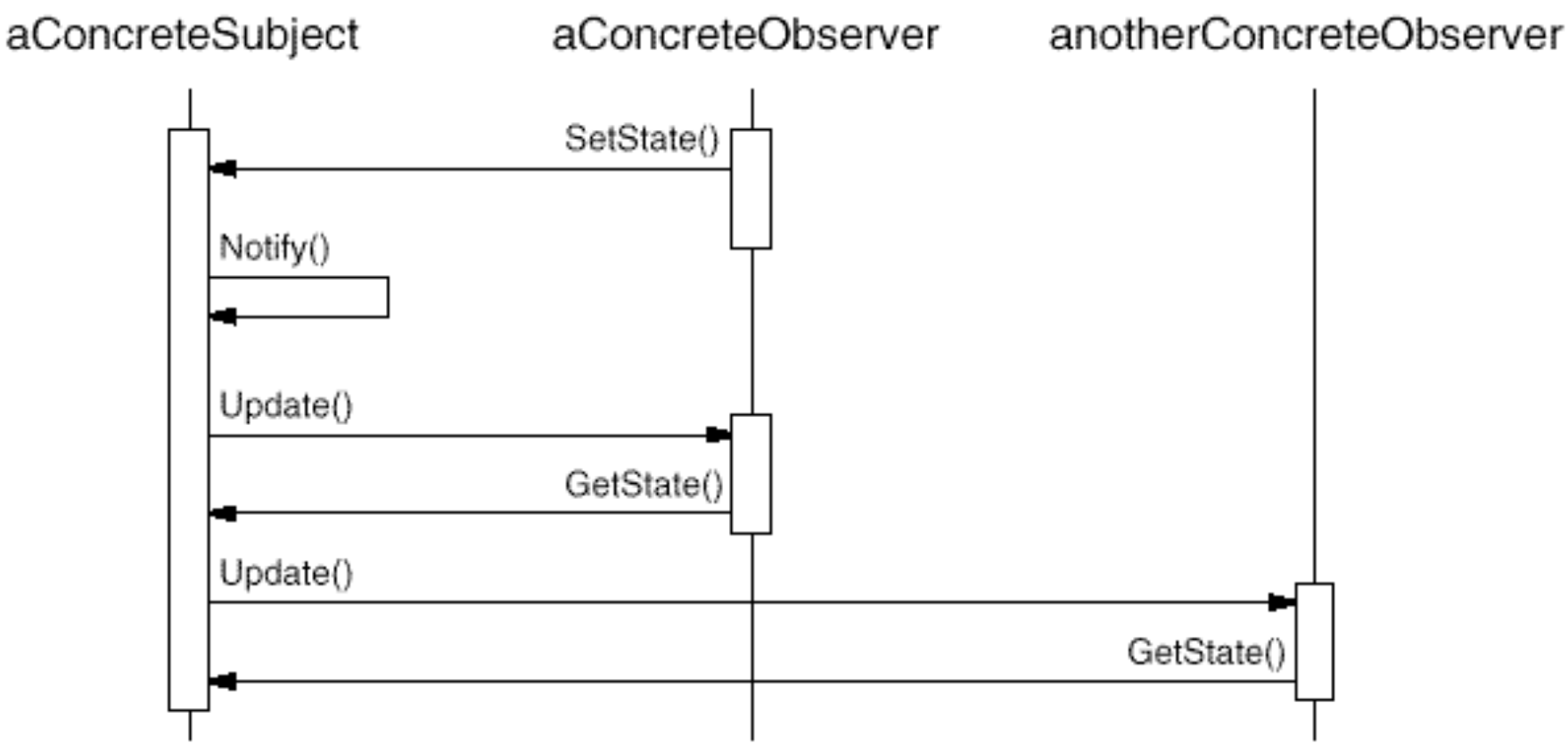


`notify` invokes **`update`** on all observers

`getState` returns the contents of **`subjectState`**

`update` invokes **`getState`** on **`subject`** and stores result in **`observerState`**.

Collaborations



Consequences of applying Observer

Benefits and liabilities of Observer include:

- *Abstract coupling from Subject to Observer*. (The coupling is tighter in the opposite direction… why couldn't it be weakened?)
- *Support for broadcast communication*. Sending a single **`notify`** request results in multiple **`update`** messages being sent (known as *multicasting*).
- *Unexpected updates*. A single change to the subject may cause a cascade of updates to observers and their dependants.
  - can be a serious challenge in practice

Triggering the Updates

Who triggers the update? When the subject gets updated, **`notify`** must be invoked in order for the **`update`** messages to be sent to the observers.

Who invokes **`notify`** on the subject? Two possibilities:

- All state-setting operations on the subject call **`notify`** after they change the subject state (less efficient but safe).
- Clients that update the subject state call **`notify`** at the appropriate time (more efficient, but also more error-prone).
- Clients can switch mode using a `setNotificationsOn/Off` protocol, but this remains error-prone.

Push and Pull Models

Pull Model: the observer is notified that a change has occurred and must find out itself what the changes are.

Push Model: the subject sends observers detailed information about the change that has occurred (in the simplest case, the entire new state itself).

The Pull model is simple, but leads to further requests from the observer to the subject.

The dumb ‘push everything’ model is simplest, but may be inefficient.

Extending the Subject registration interface to enable smarter pushing is possible, but increases subject→observer coupling.

More on the Push and Pull Models

These models make more sense with a slightly more complex example, so let's extend the Auctioneer example to an *Auction House*, where multiple items are auctioned simultaneously. The AuctionHouse is the Subject, and let's assume the Observers comprise online bidders.

Pull Model: A bidder registers with the AuctionHouse and receives a message `update(AuctionHouse)` whenever a new bid is made. This AuctionHouse is easy to implement, but now the bidder has to repeatedly interrogate the AuctionHouse to see if the specific item they are interested in has a new bid.

Push Model: A bidder registers their interest in a certain item and only receives a message `update(Item, NewBid)` whenever there is a new bid on the item in which they are interested. This is sweet for the bidder, but the Subject will have to manage the Observer -> Item mapping.

The discriminating factor is where is the specific interest of the Observer is stored: in the Observer (pull model) or in the Subject (push model).

A "push/pull" model is possible, where e.g. the AuctionHouse sends an update message to say that Item 37 has a new bid, but leaves it to the bidder to interrogate the AuctionHouse to find out what the actual bid was.

The *Lapsed Listener* Problem

Say at some point the observer no longer needs to receive updates and just ignores them.

The observer should detach, but let's say it doesn't, either due to programmer oversight or a bug.

This leads to a negligible performance degradation, and possibly a more serious **memory leak**.

- the reference in the subject to the observer prevents the observer from being garbage collected.

A **weak reference** may be an effective solution.

- A weak reference does not prevent the object from being garbage collected. Supported in many languages.

Other Implementation Issues

Mapping Subjects to Observers. Simplest is to store a list of references to observers in the subject.

- A central look-up table to store the Subject->Observer mapping is another possibility.

O*bserving multiple subjects*. Here the observer can receive updates from several subjects, so it needs to know the source of any **`update`** message.

- Simplest solution is for the subject to pass a reference to itself with the **`update`** message.

Observer Implementation in Java

Java explicitly supports the Observer pattern through `java.util.Observer` and `java.util.Observable`.

The **`Observer`** interface contains the `update` method

```
public void update(Observable o, Object arg)
```

The **`Observable`** class provides a full implementation of Subject behaviour including:

```
addObserver(Observer o)
deleteObserver(Observer o)
notifyObservers()
```

Aside: In Scala we can use a *self-type* to tighten the type of the argument to `update` to be the actual type of what's being observed.

Uses of Observer

First used as part of the MVC (Model/View/Controller) framework in Smalltalk.

Observer is a very commonly-occurring pattern:

- **Java Listeners** are essentially a specialisation of the Observer pattern.
- Closely related to the **Publish/Subscribe** pattern commonly used in message-oriented middleware
- Language support in Java, C#, Ruby and others.

All social media applications embody the notion of Observer (e.g. 'follow' on X/LinkedIn etc.).

Related Patterns

Related Patterns include:

- **Template Method**
- **Strategy** can be used where the `notify` or `update` operations are complex and added flexibility is required.
- **Mediator** may be used to coordinate updating of multiple interdependent observers
- **Adapter** can be applied if an observer wanna-be doesn't implement the required `update` interface.

- How does Observer relate to the SOLID principles?

Observer – Summary

- Observer is a very pervasive design pattern, and is a central part most UI architectures.
- We looked that the pattern in a general way. You'll develop a concrete implementation in practicals.