Introduction to Software Quality

Comp 47480: Contemporary Software Development

Is Programming Art, Science or Engineering?

The 1st International Conference on the
Art, Science, and Engineering
of Programming

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.

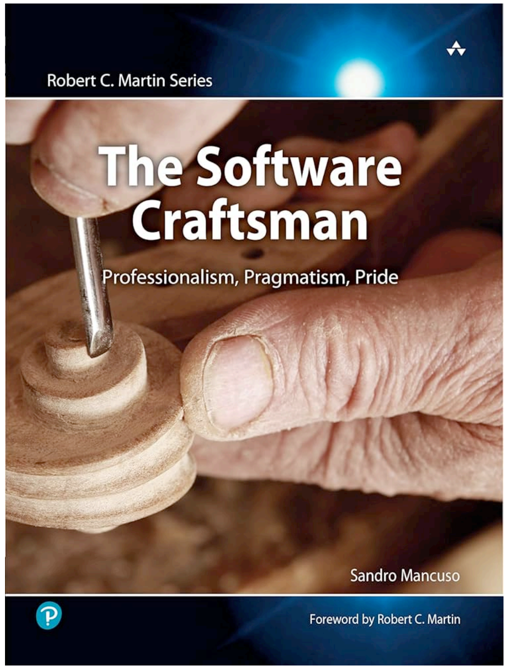
Source: Collins Dictionary

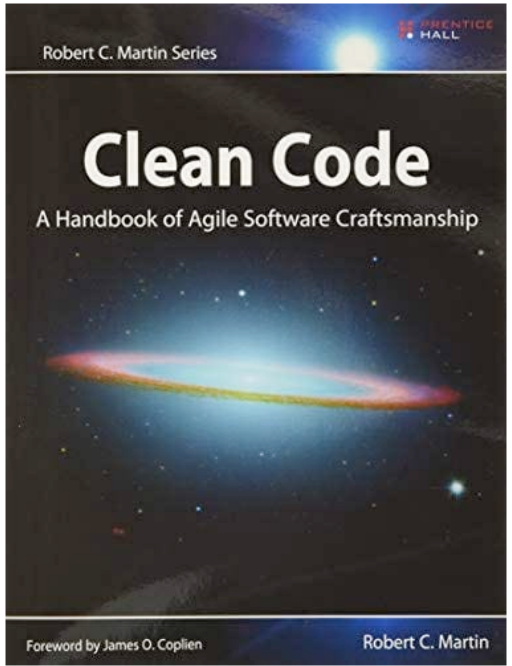
Software Craftsmanship

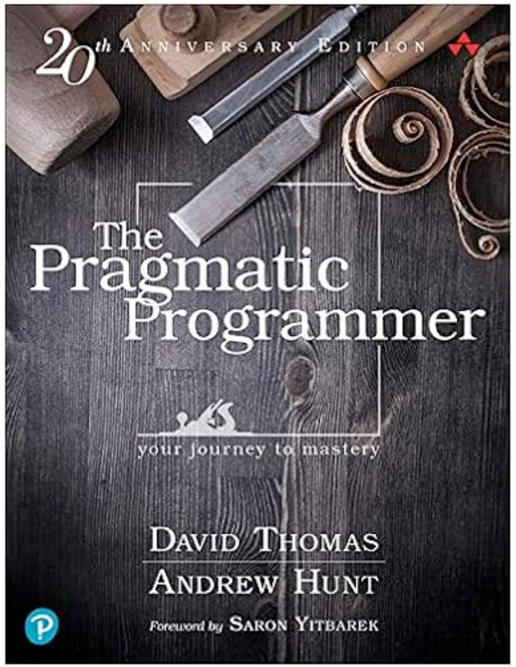
Extends Agile with a focus on the production of high-quality software that can be evolved easily.

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.


Robert C. Martin Series
The Software Craftsman
Professionalism, Pragmatism, Pride
Sandro Mancuso
Foreword by Robert C. Martin


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin



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

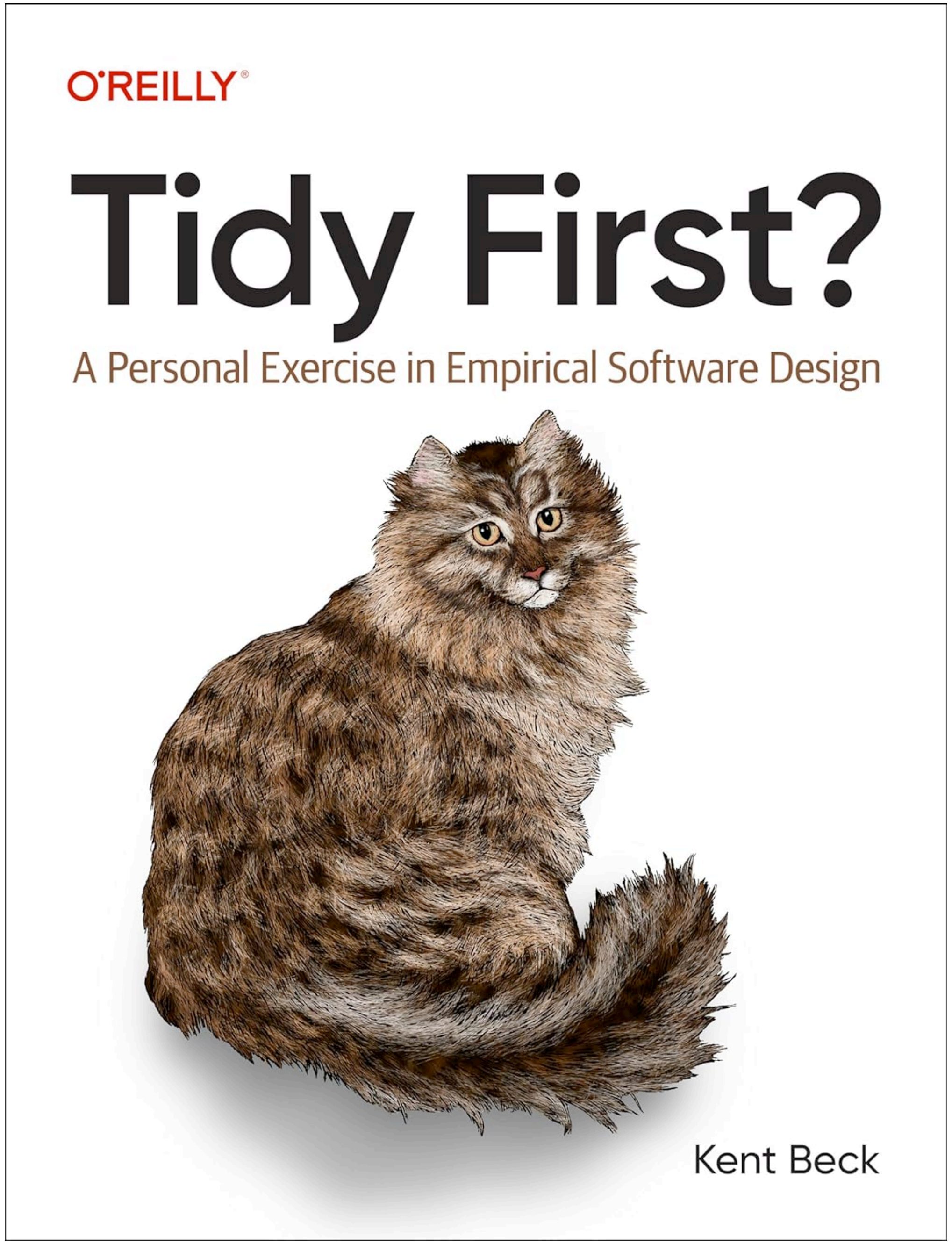
Two foundational principles can be said to capture the essence of software quality:

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.

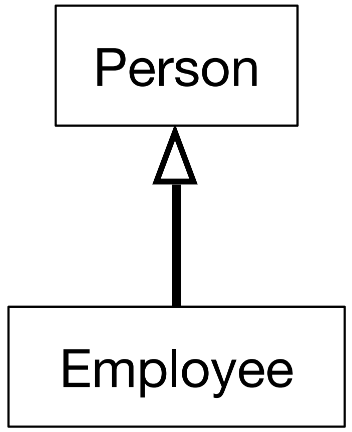
```
class Mailbox{
  public void addMessage(Message){...}
  public void removeMessage(int){...}
  public String getCommand(){...}
  public void printMessage(int){...}
   ...
}
```

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?

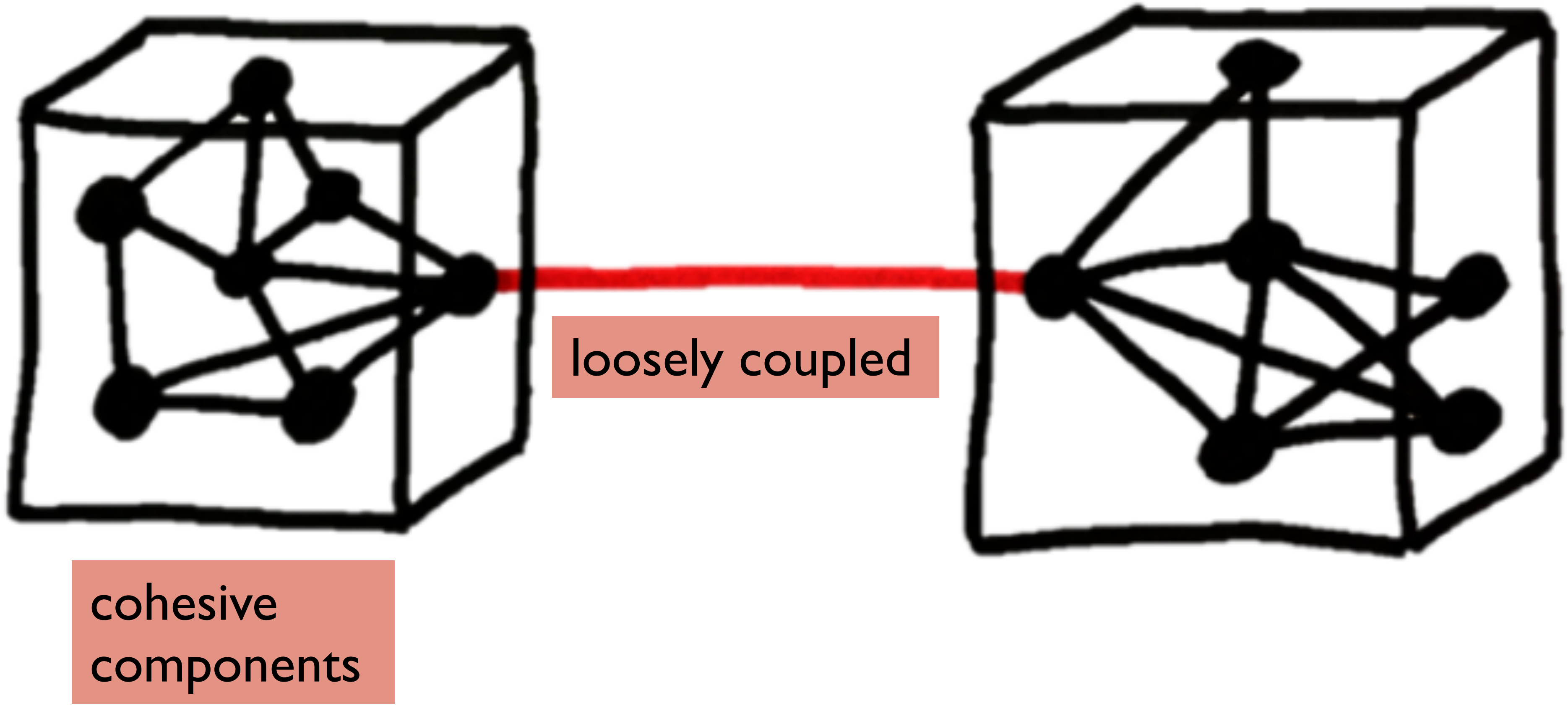


These two classes are *explicitly* coupled.

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

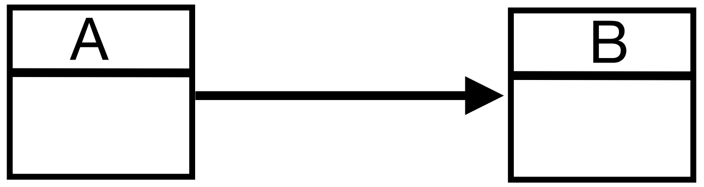
Classes are easier to **understand**. They are focussed on ‘one thing’ (high cohesion) and interact little with the rest of the system (low coupling).

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.



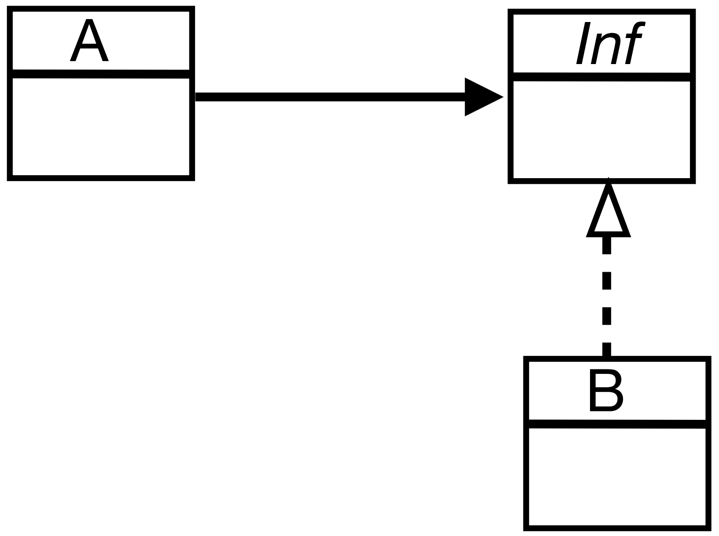
Here class A and B are coupled.

A depends on B.

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

Static Coupling, Dynamic Coupling

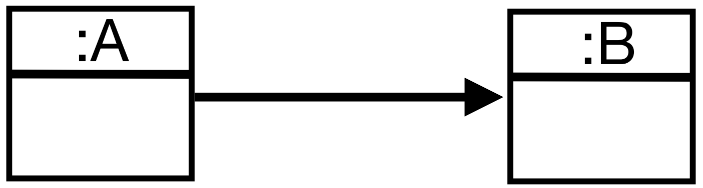
We're concerned with **static (compile-time)** dependency, not dynamic (run-time) dependency.



class diagram

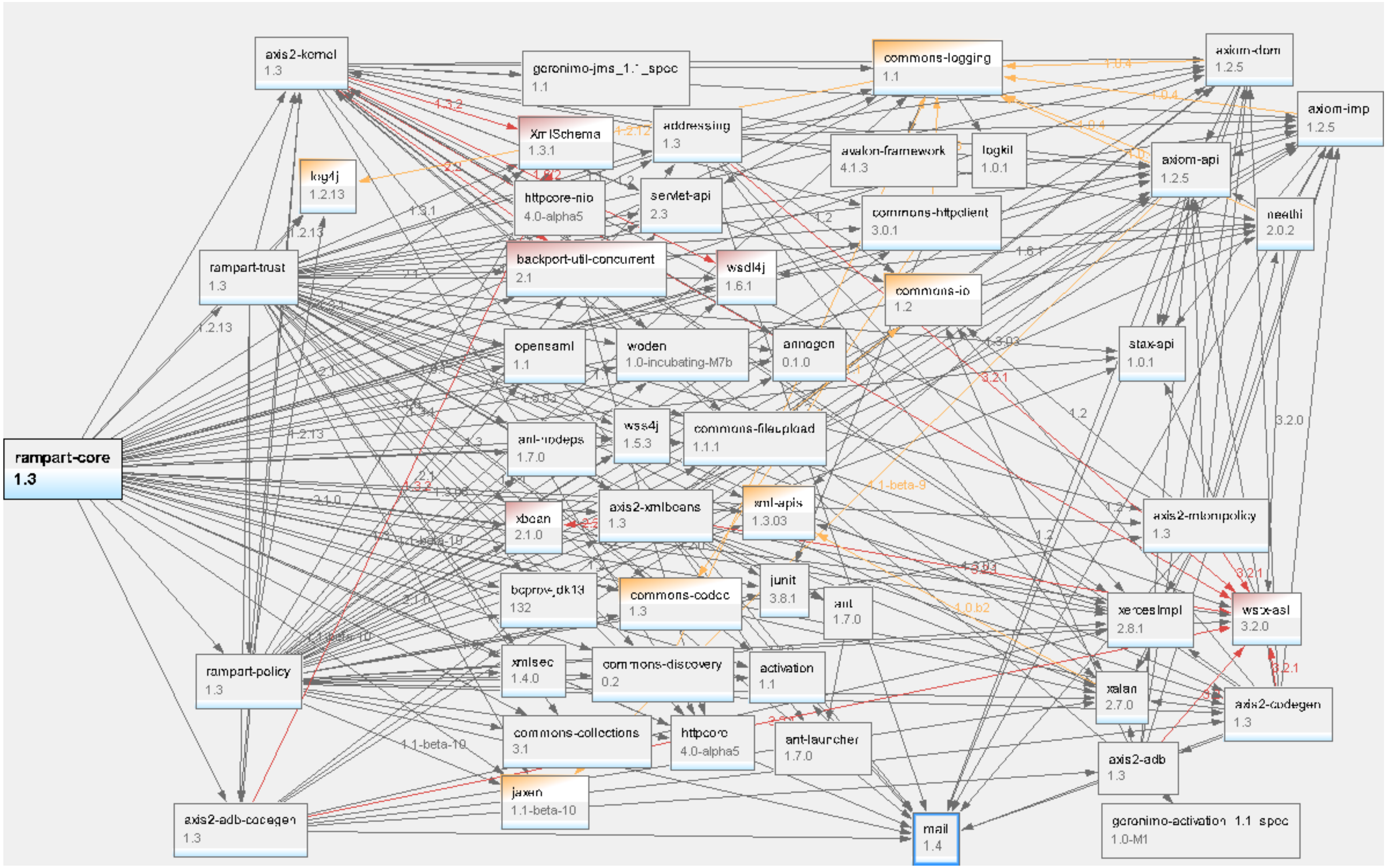
Class A does not depend on Class B directly.

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.

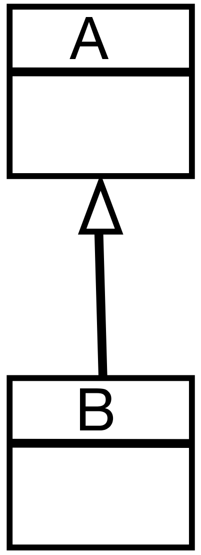
2. What classes does A depend on? To understand A, it may be necessary to understand them first.

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?

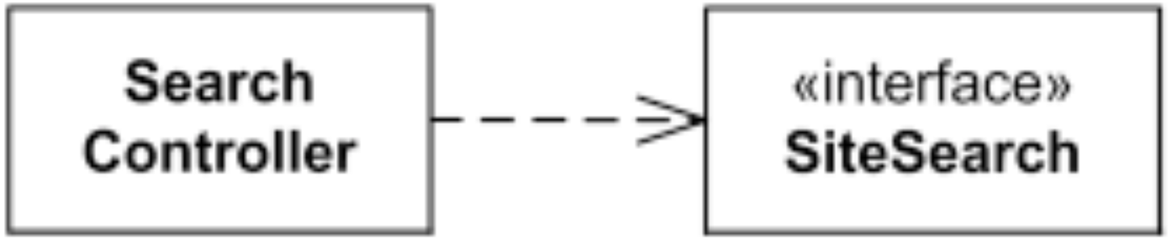
```
class Foo {
  public Foo {
    ... = new FooBar();

  }
  ...
}
```

`Foo` depends on `FooBar`.

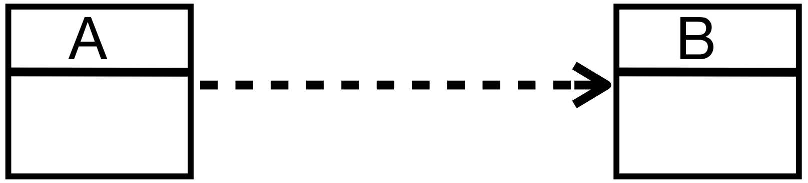
UML and Dependency

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.

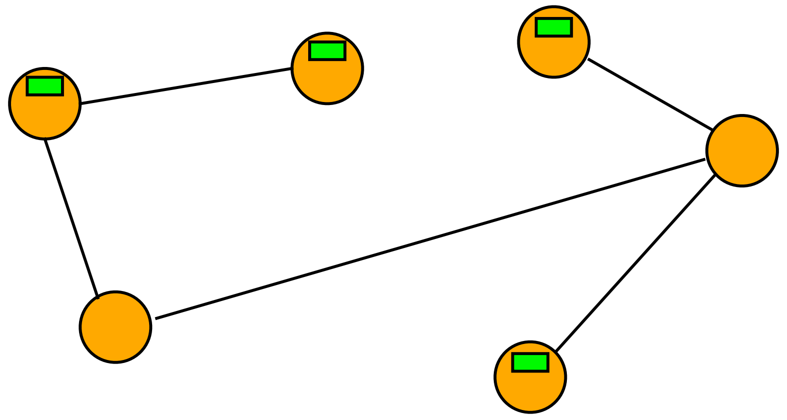
Key finding: More than 14% of all methods were copied from another package in another project.

(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.