1

Handout for lecture 4 (SE):

Principles and Patterns

1. Recap

2

Lecture overview

- agile development
- writing maintainable code
- principles and patterns (writing maintainable code part 2)
- software testing

3

Recap: writing maintainable code

- How to write code that supports changing requirements?
  - low coupling
  - high cohesion
- Basic advice to achieve low coupling / high cohesion
  - Split long / complex functions
  - Do not copy code!
  - Avoid circular dependencies in the architecture
  - Write automatic unit tests
- Incremental design
  - avoid premature generalisation
  - when a possible improvement presents itself, refactor
  - avoid duplication of ideas

4

Today: principles and patterns

- Principles: rules you adhere to in your code
  - setting rules can be important in team work
  - tried and tested principles that help keep coupling low and cohesion high

- Patterns: standardised solutions
  - often recurring problems have standard solutions
  - note: some modern languages implement patterns as language features
- Testability
  - applying principles for high-quality code also makes your code more testable!
- Inheritance
  - inheritance coupling
  - how to properly use inheritance
- Code smells and Anti-patterns
  - code smell: indication that there is a problem with your code
  - anti-pattern: bad solutions for often recurring problems

2. Principles

5

Recall: simplicity in agile design

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

– Antoine de Saint-Exupéry

Any intelligent fool can make things bigger, more complex and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction.

– Albert Einstein

Keep It Simple, Stupid

– U.S. Navy

2.1 Principles to maintain simplicity

6

Design and workflow principles to maintain simplicity

- **Principle of Least Astonishment**

  If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature.

  – Mike Colishlaw, *The design of the REXX language*

  This is a rule both for user interfaces, and for code design. Essentially: a system / framework / piece of code should behave in the way that most users will expect it to behave. If you put functions in an unexpected place, or have side effects that would surprise people, you should rethink your design.

- **Fail Fast**

  This principle encourages agile developers to try things out, experiment, and accept that these experiments will often fail and require discarding or drastic changes. This is

the opposite of the idea of designing and building a large system and only at the end discovering that it is not well-suited, or the design ideas were problematic.

The ten minute rule is a good example of applying this rule: when two team members cannot agree on how to do something after 10 minutes of discussion, they just both start implementing their own favourite approach, to quickly have a working prototype. One of those will likely be worse than the other; a failure.

The important thing of the "fail fast" mindset is to remove the stigma from failure; accept that an idea was not so good and pivot, rather than falling into the sunk cost fallacy.

See also: `https://www.techtarget.com/whatis/definition/fail-fast`

- **Limit Published Interfaces**

  When you write a library or class definition for a project and publish it to the outside world, it becomes harder to change, since people rely on the existing definition. Changing it later will then require you to build in some support for backwadrs compatibility.

  Hence, if you do not *have* to publish an interface for a project that might still be subject, don't. (Or: only publish a part of the interface, and leave less important parts out of it.)

- **You Aren't Gonna Need It**

  (See next slide)

- **Once and Only Once**

  (See following sldies)

7

You Aren't Gonna Need It

~~The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly.~~ – Gang of Four

YAGNI

Avoid premature generalisation!

(And don't keep outdated code because "I might need it in the future")

8

Incremental design

- The first time you create a design element, be completely specific.
- The second time you work with an element, make it general enough to solve both problems.
- The third time, generalise it further.
- By the fourth or fifth time, it's probably perfect!

9

Once and Only Once (aka: Don't Repeat Yoursef)

An idea should be expressed only at one place in the code.

- Don't copy code! (And as explained in the last lecture, if you do have a really good reason to copy code, you should actually be okay with doing the push-ups or similar penalty because it's worth it!)

- Non-obvious knowledge should probably be wrapped in an abstraction.
  - **Example:** int money
  - Regular code occurrence: printf("$%.2f", money/100.0)
  - Better: class Money { int pennies; void print() { ... } }

10

A conflict: YAGNI/KISS versus OOO

- Scenario: creating game objects with weights stored in units of 1 gram.
- YAGNI: implement only
  - void set_weight(int grams)
  - int query_weight()

- OOO: implement
  - void set_weight(Weight value)
  - Weight query_weight()

  From the perspective of "keep it simple, stupid", this seems like it introduces an unnecessary complication, and kind of a generalisation! Doing this for everything will certainly mean a lot of tiny classes in your code, most of which you probably won't need.

  Fortunately, we do not actually have to choose here! We could alternatively design the code as follows:

  - void set_gram_weight(int number)
  - int query_gram_weight()
- Why does this minor change (explicitly indicating the unit in the function name) make such a difference?
  - In Discworld: there are 18163 occurrences of `set_weight` in the codebase. When we decided at one point that we wanted to change the minimal weight to support items of 1/100 gram, it was quite a pain to go over everything! Of course, for backward compatibility we could simply let set_weight and query_weight *default* to grams, and introduce a new function for smaller weights, but unfortunately:
  - There were also quite a few calls such as: xxx→set_weight(yyy→query_weight())

    When the function is named *set_gram_weight* instead of just *set_weight*, the person writing this might feel a bit uncomfortable, and recognise that they are making an assumption (that all weights are expressed in grams). While the assumption is *currently* true, it is still an assumption, and duplication of an idea in the codebase. Hence, with the clearer naming, it would likely not take too long before someone adds the following function:

    * void copy_weight(object other)

    Which would avoid the problem! (And would have saved us a *lot* of time when making that change...)
- **Best practices:**
  - Do avoid duplicating non-trivial ideas!
  - However, don't introduce abstractions just for the sake of it.
  - And definitely do not generalise beyond removing duplication!

11

Recall: Risk-driven design

When it seems likely that code is going to be changed in the future, this is of course exactly when you do prioritise OOO over YAGNI, and introduce abstractions so as to avoid duplication around a concept. Recall from the previous lecture:

But I already have a strong suspicion of what I will want to do in future iterations and I can see that this is going to be a really big problem...

- Remove duplication around the risky code.
- Schedule risky features early on!

Example: I only need RED/GREEN/BLUE/YELLOW now, but eventually will want colours that depend on user settings.
Bad: class Colour with unused functions setRGB(), setDisplay(), ...
Good: class Colour with options for RED/GREEN/BLUE/YELLOW.

12

Recall: Incremental design

During/after implementing, ask questions:

- Is this code similar to other code in the system?
- Are class responsibilities clearly defined?
- Are concepts clearly represented?
- How well does this class interact with other classes?

If there is a problem:

- Jot it down, and finish what you're doing.
- Discuss with teammates (if needed).
- Follow the ten-minute rule.

2.2 Principles to maintain low coupling and high cohesion

13

Some principles to obtain a decoupled, cohesive design

- Single Responsibility Principle
- Dependency Inversion Principle
- Isolate Third-party Components
- Self-Documenting Code

14

Single Responsibility Principle

Every module/class should have responsibility over a* single *part of the functionality, and that responsibility should be entirely encapsulated by the class.

(Or put differently: maintain *functional cohesion* in every module.)

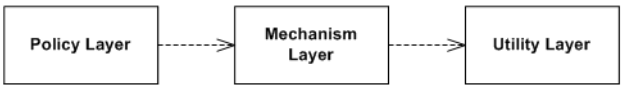
- A module/class should have only one reason to change.
- Bad example: a module that compiles and prints a report.
- Good example: a module that compiles a report.
- Good example: a module that is responsible for arithmetic reasoning.
  - large responsibility, but only one responsibility
  - may contain sub-modules for specific sub-responsibilities
- Note: a responsibility should not contain "and".

15

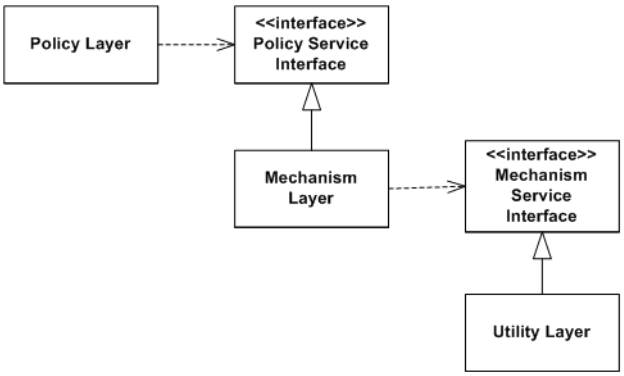
Dependency Inversion Principle

(A) High-level components should not depend on low-level components. Both should depend on abstractions.

(B) Abstractions should not depend on details. Details should depend on abstractions.



- Suppose high-level class A depends (via interaction coupling) on low-level class B. (For example, the policy layer depends on the mechanism layer, and the mechanism layer depends on the utility layer.)
- If a mechanism in B changes, we should not have to adapt A.
- Instead, we should have made an abstract interface B′ on which A depends and which B implements. (In the example, we make an interface for both the mechanism layer, and the policy layer only takes this interface as an argument; not the actual classes of the mechanism layer. Similarly, we abstract the utility layer into an interface.)

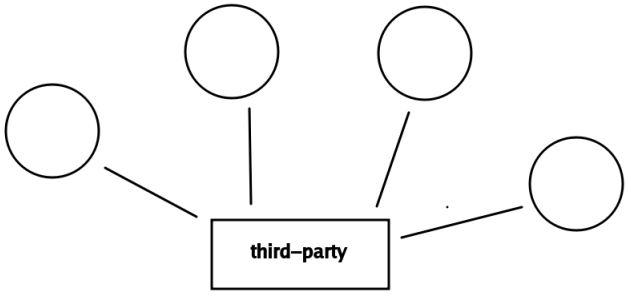


- In essence, it becomes the role of B′ to capture the interaction aspect between A and B.

16

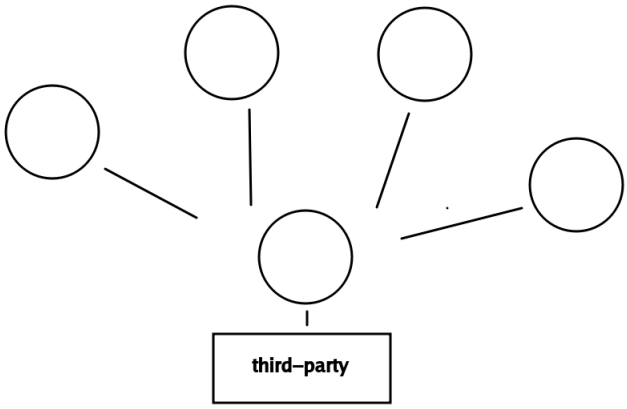
Isolate Third-party Components

A hidden source of duplication lies in calls to third-party components. When you have these calls spread throughout your code, replacing or augmenting that component becomes difficult.



Here, multiple components of our code interact with for instance a third-party library. But what if we ever decide that we want to use a library from a different party? Or indeed, what if the third party decides to change its libraries in a later version? We would have to change all our components to match.

It would be much better if, instead of 4 components relying on this third-party library, only a single component interacted with it; a wrapper interface, that provides access to the library. The original 4 components simply do calls on this interface; and if the library is ever replaced, we only have to update the one wrapper component.



As an example of this, suppose that the third party component is a database API. Specific SQL queries vary slightly between different databases. If you follow this principle, you would *not* put specific queries throughout your code, but let the database interface build the queries, following specific calls. Note that, depending on how this is done (and how pervasive the database use in your code is) this may also allow you to redesign your database with relative ease.

17

Self-Documenting Code

You might have learned to document your code properly. Indicate properly what parts are for. And this is a good habit in general. However, it is *not* good enough if your code ends up looking like this:

```
PConstant InputReaderAFSM ::  read_constant(string description) {
  // start:  do we have a colon, to separate name and type?
  int colon = description.find(':');
  if (colon == string::npos) {
    last_warning = "missing colon.";
    return NULL;
  }
  // is the thing before the colon a single word and legal name?
  string name = description.substr(0,colon);
  while (name.length() > 0 && name[0] == ' ') name = name.substr(1);
  while (name.length() > 0 && name[name.length()-1] == ' ')
    name = name.substr(0,name.length()-1);
  if (name.length() == 0) {
    last_warning= "missing constant name.";
    return NULL;
  }
  for (int i = 0; i < name.length(); i++) {
    if (!generic_character(name[i])) {
      last_warning= "illegal characters in " + name + ".";
      return NULL;
    }
```

```
  }
  // is the thing after it a legal type?
  string typetxt = description.substr(colon+1);
  PType type = TYPE(typetxt);
  if (type == NULL) {
    last_warning = "could not read type:  " + last_warning;
    return NULL;
  }
  return new Constant(name, type);
}
```

This function is not functionally cohesive (though it is sequential cohesion). But the code smell (aka red flag) that we could see here is that we *need* the comments to understand what the code does. Could this not be improved? Since we also have the code smell that this function is well over 15 lines, we surely can...

18

Self-Documenting Code

```
PConstant InputReaderAFSM ::  read_constant(string description) {
  int separator_position = find_separator(description, ':');

  string name = readLegalName(description.substr(0, separator_position));
  if (name == "") return NULL;

  PType type = readValidType(description.substr(separator_position+1));
  if (type == NULL) return NULL;

  return new Constant(name, type);
}
```

This code doesn't *need* comments. We can still add some, for instance to indicate that the function may alter `last_warning` as a side effect and return NULL if something goes poorly, but the functionality has been split up into sbu-functions whose name already indicates what they do. Since the variables also have clear names, this code is effectively self-documenting.

3. Patterns

19

Software design pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

- not code, but rather a kind of template, a standard way of doing things
- arguably: a missing programming language feature

20

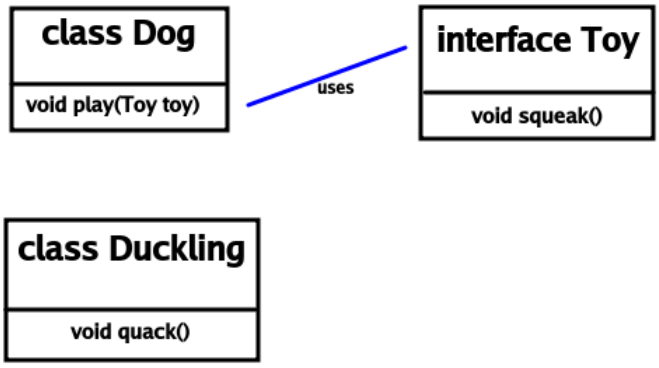
Some design patterns

- Adapter pattern: use a wrapper to convert the interface of a class without modifying its source code
- Facade pattern: tidy up the interfaces to a number of related objects that have often been developed incrementally
- Observer pattern: tell several objects that the state of some other object has changed
- Decorator pattern: allow for the possibility of extending the functionality of an existing class at runtime

21

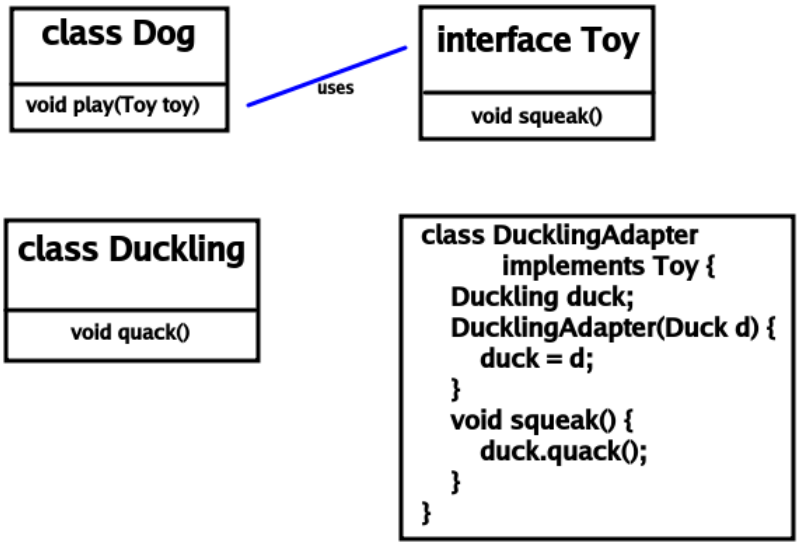
Adapter pattern

The *adapter pattern* creates a wrapper class that follows an interface used by object A, to allow it to interact with object B. (Hence, it *adapts* object B.)



Here, the Dog class has a functionality to play with a toy. Since the dog only uses the interface, we know that the only behaviour necessary for the toy is that it can squeak.

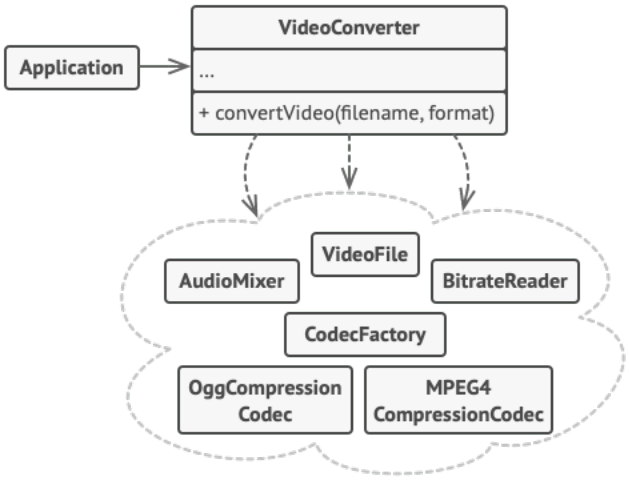
But now suppose that we have a class for rubber duckies. We want to let the dog play with the duckie as well, but the duckie is not designed as a toy; it does not squeak, and extneding its code would clutter up the interface for other places where the duck is used. Instead, we create an adapter wrapper class, which has a duck as an internal argument, and simply passes on the calls from the Toy interface to the appropriate functions in this duck.



22

Facade pattern

Somewhat related, the facade pattern presents an interface to more complex code. Like we can view the facade of a building and not see the whole infrastructure inside, we use a facade object to hide the details of one or more interrelated classes.



Note that this pattern effectively implements the *isolate third-party components* principle. However, it can also be used within your own code, for example to build a facade for a different layer of code.

23

Observer Pattern

The *observer* pattern decreases coupling by replacing direct calls from one class to another by an event handling pattern. Consider for instance the following code.

```
class A {
  void update() {
    ...
    otherClass1.do_thing_one();
    otherClass2.do_thing_two();
    otherClass3.do_thing_three();
  }
}
```

Here, an update for class A induces various updates to other classes. Think for instance of UI updates, or something being sent on the network. The downside is that there is direct coupling here between class A and these other classes. Not only does class A need to be aware of all these classes, but when one of them is updated to behave differently, then class A should probably be updated alongside it.

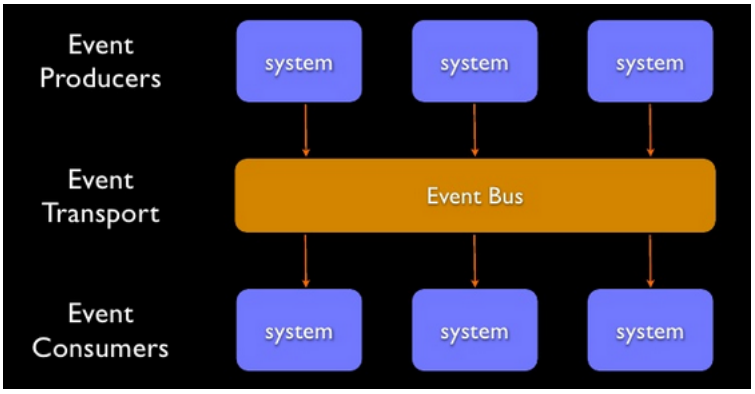
As an alternative, suppose that each of those classes that needs to react when A is updated implements a specific interface `AListener`, and at the start of the program, class A is informed of all the classes which listen to it. Then the code could look something like this.

```
class A {
  void notify_observers() {
    for (int i = 0; i < observers; i++) {
      observers[i].eventChangeToA();
    }
```

```
  }
  void update() {
    ...
    notify_observers();
  }
}
class B implements AListener {
  void eventChangeToA() { do_thing_one(); }
}
```

Class A does not need to be aware of the specific classes that are notified, since it only needs to keep track of a number of `AListener` objects. The listening classes can react to the update event appropriately.

Note that this is a small version of the *event-driven architecture* which we saw before. There, events are used as the primary mode of communication between classes. The observer pattern can be used also for individual classes whose events invoke reactions in a number of other classes.



24

Decorator pattern

The decorator pattern uses wrapper objects and *composition* to modify the functionality of an object at runtime, without affecting other instances of the same class (or module).

The decorator pattern works by defining an interface, a base object that implements a minimal definition of the interface, a decorator class which is a wrapper implementing the interface that simply forwards all calls, and a number of extension classes.

For example:

- KeyComponentInterface
- KeyComponent implements KeyComponentInterface (This is the basic class, providing a minimal interpretation for the KeyComponentInterface.)
- Decorator implements KeyComponentInterface

  - internally keeps an object `component` of type KeyComponentInterface
  - implements all interface functions by forwarding them to `component`

- Extension1(c) inherits Decorator, but overrides method x
- Extension2(c) inherits Decorator, but overrides methods y, z
- ConcreteDecorator inherits Extension1(Extension2(KeyComponent))
- A concrete example where this is useful could be a windowing system with configurable borders, scrollbars and perhaps other settings. When a user creates a window with certain properties, an appropriate object of class WindowInterface can be created on the fly.

25

Example: a simple animation

Let us say we have an animation of flying objects. There are different kinds of objects, some of them move while others stand still, and when they run into each other they bounce off. The base object class could look something like this.

```
class AnimatedObject {
  protected:
    Position position;
    Model *model;
  public:
    virtual void update() { ⟦update location⟧ }
    virtual void draw() { ⟦draw model⟧ }
    virtual void meet(AnimatedObject other) {
      ⟦handle collision⟧
    }
}
```

Different options! **(12 combinations)**

- square, circle or triangle (difference in `draw`)
- moving or static blocks (difference in `update`)
- solid or immaterial (difference in `meet`)

Let's say we don't want to implement 12 different classes, and we definitely do not want to copy functionality between them. This can perfectly be handled using decorators!

26

Example: a simple animation

We start with the interface, and the basic object. In this case, we arguably have *three* basic objects: one for each of the kinds, as there is no basic drawing behaviour that works for all of them.

```
interface AnimatedObject {
  void update();
  void draw();
  void meet(AnimatedObject other);
}

class Circle implements AnimatedObject {
  public:
    void update() { }
    void draw() { ⟦draw circle⟧ }
    void meet(AnimatedObject other) { }
}
class Square implements AnimatedObject {
  ...
}
```

27

Example: a simple animation

The decorator is just a wrapper class, which passes calls directly to the AnimatedObject that it wraps. Quite boring, but the power comes from inheriting it.

```
class DecorAnimObject :  AnimatedObject {
  private:
    AnimatedObject core;
  protected:
    Position position;
  public:
    DecorAnimObject(AnimatedObject c) {
      core = c;
    }
    virtual void update() { core.update(); }
    virtual void draw() { core.draw(); }
    virtual void meet(AnimatedObject other) {
      core.meet(other);
    }
}
```

Of course, this can be tedious if your interface has many functions. Some programming languages directly support composition so there you do not have to do this manually.

28

Example: a simple animation

The extensions are defined by inheriting the decorator, and overriding specific functions.

```
class SolidObject :  public DecorAnimObject {
  public:
    SolidObject(AnimatedObject c) {super(c);}
    void meet(AnimatedObject other) {
      ⟦handle collision⟧
    }
}
class MovableObject :  public DecorAnimObject{
  private:
    Direction direction;
  public:
    MovableObject(AnimatedObject c)  { super(c); }
    void update() { ⟦update position⟧ }
}
```

29

Example: a simple animation

Challenge: how to get a solid movable square?

Answer: new SolidAnimObject(new MovableAnimObject(new Square()));

30

Exercise: a Pacman game

As a very similar, but more realistic example, here is an idea you can try on your own.

```
abstract class GameObject {
  private:
    Position position;
    Model model;
  public:
    virtual void update() { }
    virtual void draw() { ⟦draw model⟧ }
    virtual void meet(GameObject other) { }
}
```

Needed objects: `Pacman`, `MeanGhost`, `EdibleGhost`, `Obstacle`, `Food`.

Your task: design a class technology to do this!

Note that you may need additional functions, and that it is possible for one extension to override multiple functions.

31

A recurring theme

A powerful technique for maintainable code is using **abstractions**:

- use abstractions to avoid duplicating ideas (e.g., class Weight)
- use an abstraction to isolate third-party functionality
- when interacting with distant code in your own codebase – that is, code that you do not really want coupling with, use an abstraction instead
- abstractions play a critical role in several patterns

Modern languages typically have interfaces or abstract classes which accommodate abstractions easily.

Even if you are using a language that does not have these features: most of this can be done through functions.

4. Testability

32

Principles, Patterns and Testability

Observation: code with loose coupling and high cohesion is easier to unit test!

- Recall the Single Responsibility Principle: *Every module/class should have responsibility over a* single *part of the functionality, and that responsibility should be entirely encapsulated by the class.*

  Following this principle exactly gives you loose coupling, high cohesion... and ease of testing.

33

The Single Responsibility Principle and Testability

Consider some code that violates this principle.

```
string lookmap_text(string text, int lookmap_type) {
  string ret = text;
  string map = lookmap(this_player()->map_setting());
  send_room_info(this_player(), map);
  swwitch(lookmap_type) {
    case NONE: return text;
    case TOP: return map + text;
    case LEFT: return combine(map, text);
  }
}
```

Here, the function has multiple responsibilities: it both generates the combination of the "lookmap" with the given text, and it causes room information to be sent to the player as a side effect. The side effect makes this function quite annoying to test.

On a positive note, however, this function *does* separate the trickiest part of the behaviour into a separate function: the "combine" function which takes two strings and returns a string that has the map on the left and the text on the right (with appropriately inserted newlines) has a very clearly defined responsibility, and should not have any side effects. Hence, this subfunction, at least, is highly testable.

```
PConstant InputReaderAFSM ::  read_constant(string description) {
  // start:  do we have a colon, to separate name and type?
  int colon = description.find(':');
  if (colon == string::npos) {
    last_warning = "missing colon.";
    return NULL;
  }
```

```
  // is the thing before the colon a single word and legal name?
  string name = description.substr(0,colon);
  while (name.length() > 0 && name[0] == ' ') name = name.substr(1);
  while (name.length() > 0 && name[name.length()-1] == ' ')
    name = name.substr(0,name.length()-1);
  if (name.length() == 0) {
    last_warning= "missing constant name.";
    return NULL;
  }
  for (int i = 0; i < name.length(); i++) {
    if (!generic_character(name[i])) {
      last_warning= "illegal characters in " + name + ".";
      return NULL;
    }
  }
  // is the thing after it a legal type?
  string typetxt = description.substr(colon+1);
  PType type = TYPE(typetxt);
  if (type == NULL) {
    last_warning = "could not read type:  " + last_warning;
    return NULL;
  }
  return new Constant(name, type);
}
```

Although the multiple functionalities in this function all work together towards a single call (reading the constant), there are several subfunctionalities which can be seen as separate responsibilities, and which we might want to reuse in other functions, and test separately.

That being said, many languages do not allow you to unit test *private* functions. Hence, even when splitting this up, you may need to test the whole function in one go – though perhaps with inputs specifically designed to test the specific subfunctionalities. If you have one sub-behaviour that you really think should be unit-tested on its own, it might be necessary to for instance make it "default", (which in Java allows you to unit test it, but not for any class in another package to call the function), or provide a clearly named test access function.

34

Principles, Patterns and Testability

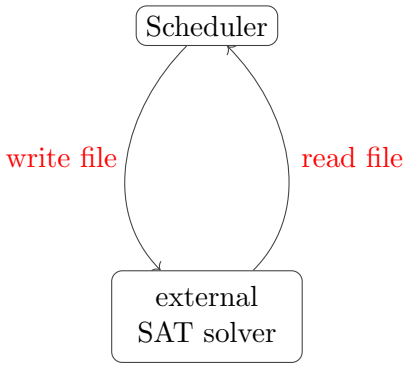
Observation: code with loose coupling and high cohesion is easier to unit test!

- Single Responsibility Principle
- Isolate Third-party Components

35

Isolating third-party components for testability

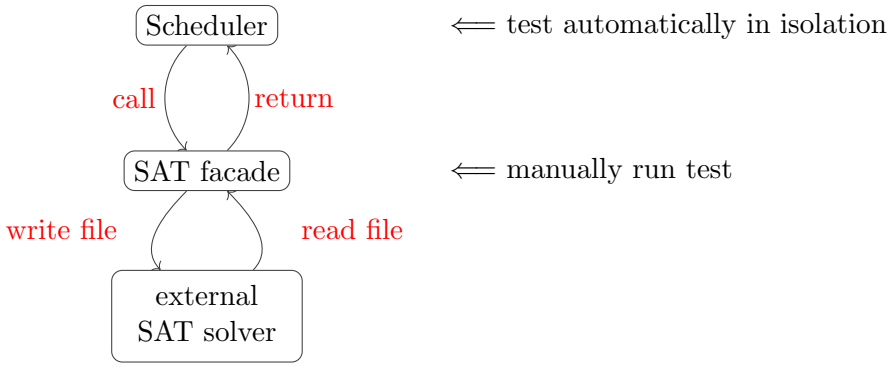
Hard to test:



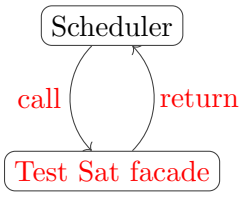
The scheduler is a tool which, given a group of GiPHouse teams and their available times as input, determines what would be the best allocation of rooms for each moment in the week, satisfying that each team gets a room for at least 4 hours a week in which they are available. It does so by making a file with a problem in proposition logic, invoking an external tool to do all kinds of clever tricks on this file (follow the *automated reasoning* course in the master if you want to learn more about this!), then reading the output of this file and translating it back into either a good schedule, or a note that no good schedule exists.

Due to the reliance on file reading/writing and the use of an outside tool, this code is rather hard to test – especially if the tool can take a relatively long time to supply an answer. If each test takes a second, it is not feasible to do an adequate number of tests for all kinds of edge cases on every commit.

Easier to test:



If you isolate the third-party component, you can avoid the side effects and the potentially long waiting time by (1) testing the Facade ↔ external solver interaction manually – this takes time, but the Facade is small and unlikely to change a lot, so when you have done the tests and everything works it is likely that you won't have to redo them very often – and (2) testing the scheduler *without* its reliance on the external solver. After all, the facade is just an interface... you could provide the scheduler with something else that inherits the same interface, and that provides simplified behavious (a *fake object*):



36

Testing using fake objects

The "fake solver" that is used for testing could be designed specifically for the test. for example, it could check that it is given an expected formula (and set warning flags if not), or give a particular output for wich you want to test how the scheduler deals with it. For an example of the latter, see the following "test solver" which simply claims that whatever problem it is given is unsolvable.

```
class TestNonSolver implements SatSolver {
  ArrayList<String> reqs;
  TestNonSolver() { reqs = new ArrayList<String>(); }
  public void addClause(Clause clause) {
    reqs.add(clause.toString());
  }
  public Solution solve() {
    return new Solution(false);
  }
}

@Test
public void testScheduleWithImpossibleRequirements() {
  TestSatSolver solver = new TestNonSolver();
  Scheduler scheduler = new Scheduler(solver);
  ...
}
```

37

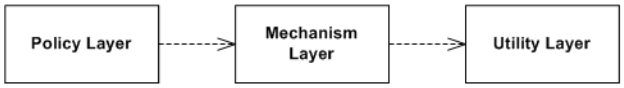
Principles, Patterns and Testability

Observation: code with loose coupling and high cohesion is easier to unit test!

- Single Responsibility Principle
- Isolate Third-party Components
- Dependency Inversion Principle

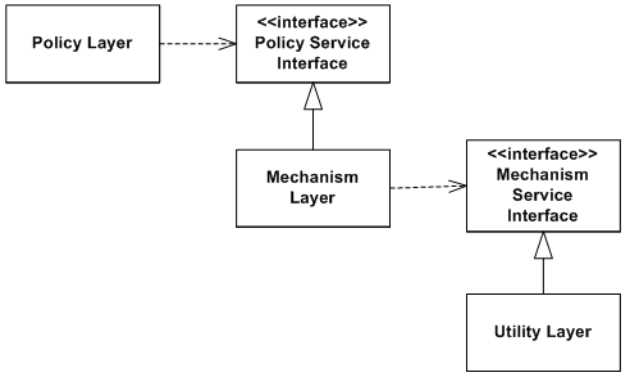
38

Testability by depending on abstractions



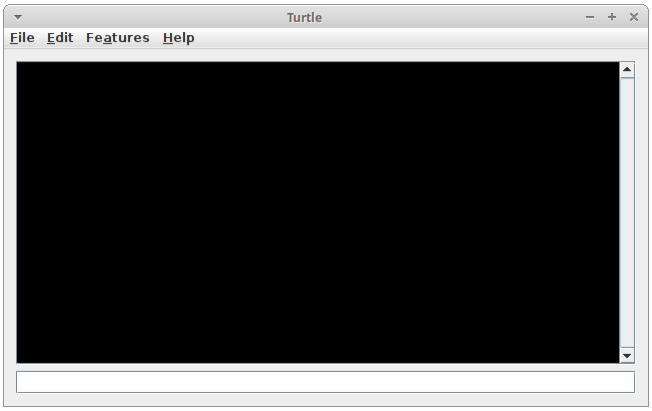
If the policy layer directly does calls on the mechanism layer, it is hard to test in isolation. Automatic tests that effectively test the whole program together are still valuable (as *integration tests*), but to be able to comprehensively test all behaviours, you will usually want to test behaviours on their own, without relying on what the rest of the program does. This allows you to test all cases, and especially edge cases, for specific functions.

Hence, if we use the dependency inversion principle, we get an architecture where the higher-level layers interact with interfaces, rather than directly with the lower-level layers. These interfaces can be replaced by fake objects for testing!



Class exercise: designing testable code

Let us say that we are designing a client for a textbased game. In this game, the user types commands, which are sent over a network. Our focus right now is the input window. In this game, users often want to send the same command twice, or repeat earlier commands. We want it to be very easy for users to recover their previous commands, edit them, and send the result.



Class exercise: designing testable code

The input window:

- It should mostly act like a normal GUI component (left and right arrow keys, entering commands, selecting, etc.).
- When return is pressed, the current text should be sent, and selected (for easy deleting).
- The up and down arrow keys browse through the "history" of previously sent lines.
- When new text is sent, it is put at the bottom of the history, and this position is selected.
- When browsing history, any change also causes the history position to go to the bottom.
- (Perhaps some more requirements.)

Before reading on, think about how you would design this! Take into account that user interactions are typically hard to test, and that the GUI is a third-party library.

41

Class exercise: designing testable code

Naive implementation:

- Put it all in one class "input window".

Better implementation:

- Separate the history! This is a component with its own responsibility.

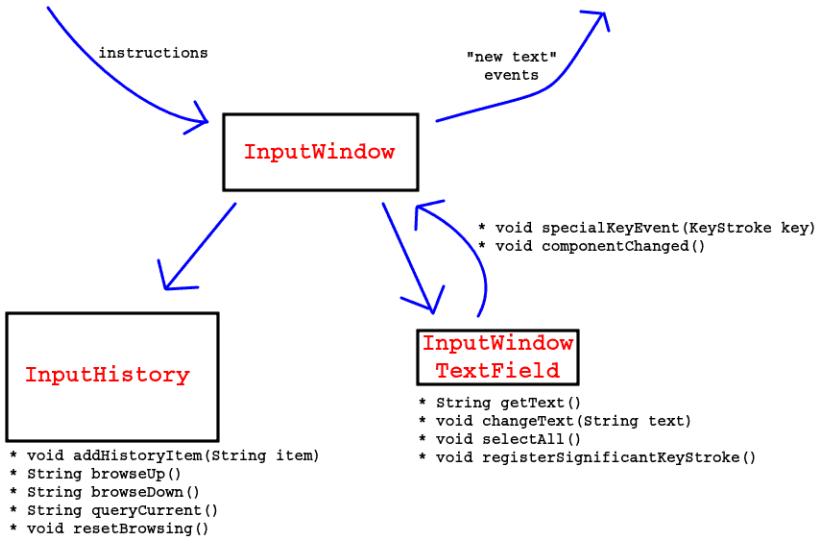
Best implementation:

- Also separate the actual GUI component!

42

Class exercise: designing testable code

This idea would yield a design like the following:



In this design:

- The InputHistory is a basic class with a single responsibility and no dependencies. It is easy to test automatically.
- The InputWindow is a manager class. It is the only one of the three classes that interacts with the outside world (e.g., giving events when return is pressed).
- The InputWindow can be tested *without* the InputHistory and InputWindowTextField by replacing these two by fake objects. For example, to test the result of pressing a return:
  - have a fake TextField where getText() returns "abc 113708a"
  - have a fake InputHistory which simply keeps track of every item added to it by addHistory
  - call InputWindow.specialKeyEvent(Enter) and test that the fake InputHistory was extended with the string "abc 113708a" and nothing else
- The InputWindowTextField is still hard to test. Depending on your test framework, this may be supported automatically, or it may require (or be much easier with) *manual testing*. This can still be done systematically: define manual tests in much the same way as you would define automatic test (considering all the edge cases and possible things that could go wrong as well as things that should be supported), write them down in your manual testing document, and agree that they are executed whenever someone changes the component.
- The InputWindowTextField is a *very small* class, which inherits the relevant GUI component, can be questioned for active text, and passes on requested key events. Because it is so small, it will rarely need changing, and the manual testing will not need to be so elaborate.

- The InputWindowTextField does not know about the InputHistory: it is simply given an object to which it must pass special key events and componentChanged() notifications. This makes it easier to systematically test it in isolation. (You can still manually test it using a fake object, which for instance prints which events are sent to it.)

5. Inheritance

5.1 Problems with inheritance.

43

Inheritance. . . awesome?

Inheritance is a key feature of object-oriented languages. It offers the ability of subtyping and code reuse. Yet. . . is it really all that great? As it turns out, inheritance can cause quite a few issues as well, and is often the source of bad coupling / cohesion.

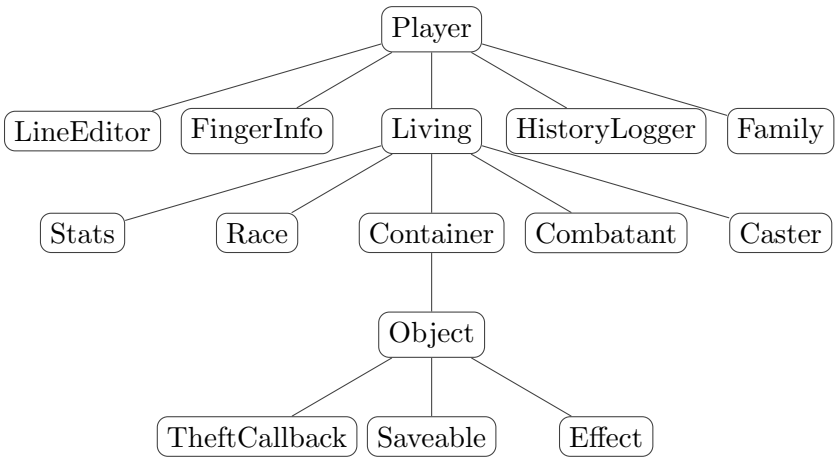
- hierarchy depth
- the diamond problem
- fragile base classes

44

Hierarchy depth

Code reuse: I loved that class from my other project! I want to use it in my new project.

As an example of a class, let us consider a *significantly trimmed down* visual representation of the Player object in the online game I often use for examples. (The one in the live game consists of 70 classes. Something like this is quite common in large codebases, especially when the language supports multiple inheritance, as is the case here.)



45

Inheritance. . . awesome?

The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

– Joe Armstrong (creator of Erlang)

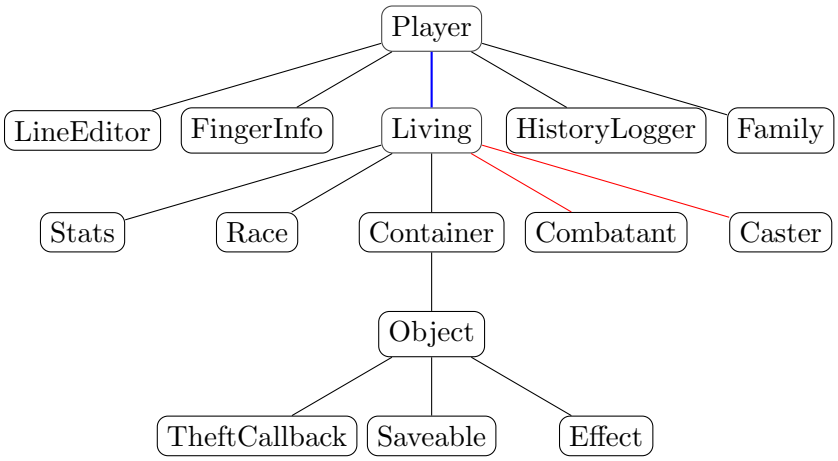
46

A deep hierarchy?

Because inheritance is great for code reuse, it often ends up being used for situations where arguably *containment* would be better. Ideally, if A inherits from B, we want to say that A is a special case of B. In the inheritance tree for the player object, there is really only *one* place where this is satisfied: a player is a form of living creature.

In addition, the multiple inheritance is used here to isolate specific sub-functionalities of "living" into separate files. Hence, the functions that define spell casting and fighting, which every living can do, are separated into different files, which increases cohesion. This use of inheritance is also good, though it does not satisfy the rule above.

For the other cases, however, it seems like inheritance is being somewhat abused. A player *has a* family, rather than *being* a family; a living *has a* race. The reason that inheritance was used here is because, for instance, the Family class defines a function query_family_name(), and using inheritance allows other code to call `player->query_family_name();`.



If, instead, we replace these forms of inheritance by containment – that is, a Player object has a *variable* `family` of class `Family`, then we need a lot of delegation functions, e.g.,

```
string query_family_name() {
  return family->query_family_name();
}
```

(Though some languages have functionality to forward calls without having to write a forwarding function every time.)

This doesn't solve the problem that we need the whole jungle to get the monkey. However, it does keep the interface cleaner. By isolating the "family" functionality to a variable, it becomes more clear where this functionality is used, and we only have to write forwarding functions for the methods of Family that we are actually interested in, rather than automatically inheriting all the public functions. If we want to reuse the Player object in another project and do not care about families there, this design makes it easier to remove the Family class altogether.

47

Inheritance. . . awesome?

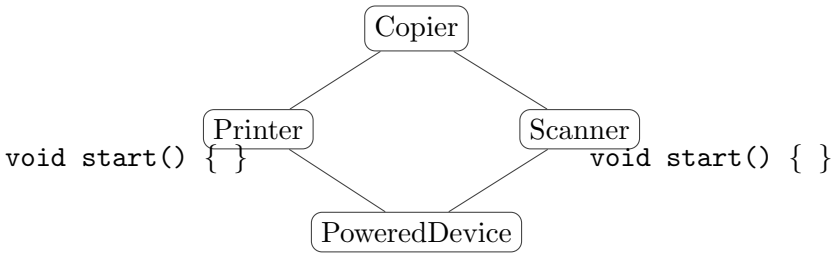
- hierarchy depth
- the diamond problem
- fragile base classes

48

The diamond problem

Another problem arises around the question: do we want to allow multiple inheritance at all? It does make sense, since often, one class is a special case of multiple more general classes. For example, a copying machine is arguably a special case both of a printer and a scanner.

However, problems arise when both inherited classes define the same function – for example because they inherit from the same object (say, an electrically powered device). What should happen when `start();` is called on the copier in the figure below?



Different languages have different answers to this question. One solution that is often taken is to disallow multiple inheritance altogether, no matter how much sense it would make. Another is to require the programmer to deal with any conflicts, for instance by supplying a function in Copier:

```
void start() {
  Printer.start();
  Scanner.start();
}
```

Arguably the latter solution is better, as it gives more freedom to the programmer and at an intuitive level it does make sense to be able to inherit from multiple higher-level classes. However, it is worth noting that we do not *need* multiple inheritance here. In fact, we have seen the perfect solution for *combining* functionalities earlier in this lecture!

49

The diamond solution with Decorator Pattern

- `interface PoweredDevice { void start(); }`
- `class BasePoweredDevice { void start() { ... } }`
- ```
  class Scanner implements PoweredDevice {
    PoweredDevice device;
    Scanner(PoweredDevice pd) { device = pd; }
    void start() {
      doScannerStuff();
      device.start();
    }
  }
  ```
- `class Printer implements PoweredDevice { ... }`
- ```
  void main() {
    PoweredDevice copier = new Scanner(
      new Printer(new BasePoweredDevice));
  }
  ```

50

Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

51

Fragile base classes

This issue is best explained with an example. Suppose we have a simple Array class in java, which defines some functions. Internally it is defined using the ArrayList from Java's standard library, but this might change at some point.

```
public class Array {
  private ArrayList<Object> a =
    new ArrayList<Object>();
  public void add(Object element) {
    a.add(element);
  }
  public void addAll(Object elements[]) {
    for (int i = 0; i < elements.length; ++i)
      a.add(elements[i]);
  }
}
```

Now, someone using this code may write the following class, which is basically an Array that keeps track of the number of elements in it!

```
public class ArrayCount extends Array {
  private int count;
  @override;
  public void add(Object element) {
    super.add(element);
    count++;
  }
  @override;
  public void addAll(Object elements[]) {
    super.addAll(elements);
    count += elements.length;
  }
  public int queryCount() {
    return count;
  }
}
```

This seems perfectly reasonable, right until someone makes a simple modification to the lower-level Array class, replacing `a.add(elements[i]);` by `add(elements[i]);`:

```
public class Array {
  private ArrayList<Object> a =
    new ArrayList<Object>();
  public void add(Object element) {
    a.add(element);
  }
  public void addAll(Object elements[]) {
    for (int i = 0; i < elements.length; ++i)
      add(elements[i]);
  }
}
```

With this change, the ArrayCount code stops working correctly, as the elements added using addAll are now counted twice!

The person who made this change was perfectly reasonable: the Array class still works exactly as one might expect. And they might not even know about the existence of the ArrayCount class – they could be making their change inside some published library. Even if they are working in the same codebase, they would likely be surprised to suddenly see the unit tests of a completely different part of the code failing.

The problem here is that the inheritance created an unexpected dependency. Of course class ArrayCount is dependent on class Array: when you change ArrayCount you should know about Array. This is reasonable, and expected since "`extends Array`" is right there in the class definition. However, we have seen that now class Array is *also* dependent on class ArrayCount, since we can no longer change it (in a reasonable way, without affecting its publishes signature) without potentially breaking the behaviour of things that inherit it!

This is unexpected and not reasonable, since a class may be inherited by any number of other classes without it being immediately obvious.

One solution to this problem is to never call public / non-final methods in the same class unless specifically indicated. This becomes the responsibility of any (library) class that may be inherited.

Another, and arguably better, solution is to use *contain and delegate* again. Instead of inheriting Array, the ArrayCount class should keep track of a variable `Array myarr`, and call the add and addAll functions in myarr instead of inherited functions. This will also protect ArrayCount from being extended in unexpected (and potentially incorrect) ways if class Array adds new public functions.

52

Inheritance. . . awesome?

- Emerging trend: use containment and delegation over inheritance.
- Inherit from interfaces and abstract classes.
- Inherit only for `is-a-kind-of` relations.

5.2 Composition over inheritance.

53

Composition over inheritance

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation".

– Gang of Four

- Inheritance is white-box reuse, composition is black-box reuse.
- Interfaces offer all the advantages of polymorphism.
  (And great flexibility!)

  Polymorphism means that for instance a function may expect an argument of type A, and instead get an argument of type B, which inherits from A. Using interfaces or abstract classes gives this advantage as well, since (a) if a class ImplA that implements the interface IntA is inherited by a class B, then B still implements IntA, and can be used by any code that expects an IntA; and (b) you can inherit interfaces, and code more specialisd classes that implement the more specialised interface (while still also having a type IntA).

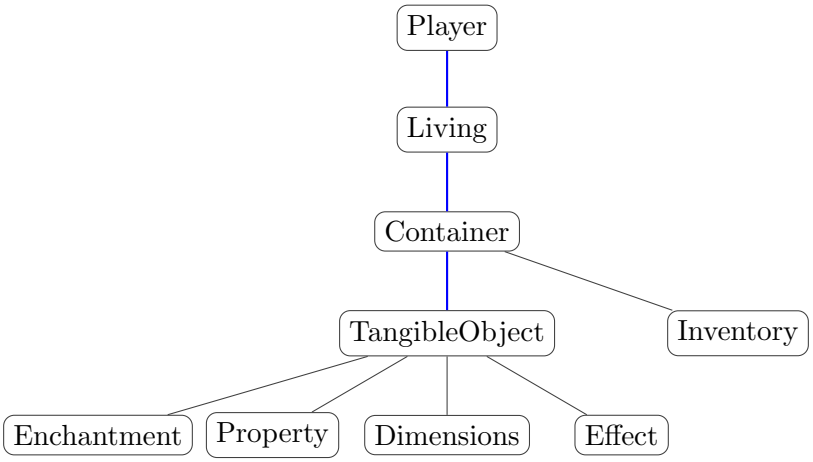
  An addition advantage of interfaces (which is unfortunately not shared by abstract classes) is that multiple inheritance of interfaces does not share the problems that multiple inheritance of classes has. This often makes it possible to have a class that implements both interfaces A and B, even when neither is a subtype of the other.
- Delegation is a powerful method, which can often replace inheritance. (That is, by doing something like `int a() { return myvar.a(); }`)

- Annoying in some popular languages. Do it regardless. (Implementation details should not influence the overall design.)
- **Note:** not a blanket "inheritance is bad"!
  - Inheritance is important, particularly for specialisation.
  - Inheritance is, however, overused. It should typically not be the first thing you think about (as it used to be, and which unfortunately was the case when the standard libraries of languages like Java were developed).

54

Exercise: the player object

Let us consider a simpler player object than the one we saw before. The blue lines indicate inheritance; the black lines containment. Hence, a player *is a* living; a living *is a* container; a container *is a* tangible object and *has an* inventory; and an tangible object *has* enchantment (that is, it can be made more durable by magic), *has* properties (that is, any code can add temporary "variables" to it, like "heat-damage": 15), *has* dimensions (length, width and height), and *has* effects (that is, any code can add a special class inheriting the Effect interface to it, for example an effect OnFire that damages the object every 5 seconds). At first glance, this inheritance tree might look reasonable.



Yet... this is not ideal. A *player* does not need enchantment, for instance (as this is only about making items, not livings, more durable). And as it turns out, in this particular game, dimensions are only used for non-living objects.

Take a moment to consider whether all the inheritance is a true is-a relationship – can this inheritance chain be improved? Feel free to make assumptions; this is not an object in a real game, so you can decide for yourself how you would expect a container to work.

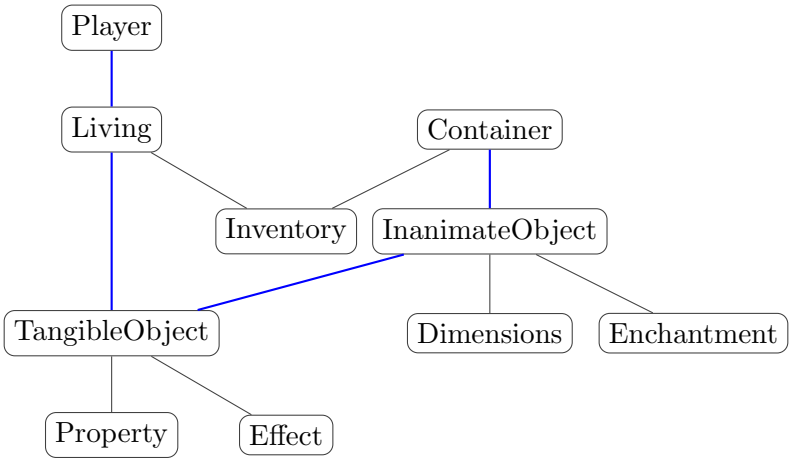
55

Exercise: the player object

The solution is given below. We recognised two problems:

- The *TangibleObject* class was being abused as a kind of *inanimate* objects. Only inanimate objects can be enchanted and need dimensions, after all. Hence, this was not functionally cohesive. The InanimateObject class was split off.
- A *container* is typically something that people can put stuff into and take it out, which probably would not work so well on a living being. Like a container, a living *has* an inventory, and *is* a tangible object, but it does not seem fair to say that a living being *is* a container.

Hence, the new inheritance tree becomes the following:



(It is possible that there is additional functionality shared between livings and containers. Even if that is the case, that is still no reason for inheritance when there is no a true is-a relationship! If there is more shared functionality, the best course depends on which functionality and how much there is; for example, some of the functionality may fit better in the Inventory class, and other functionality could perhaps be split off into a new class that both Living and Container use.)

56

Liskov's Substitution Principle

The lessons learned are well-captured by a principle formulated by Barbara Liskov in 1987:

(Objects of) sub-classes must be substitutable for (suitable objects of) their base classes without change in behaviour of the overall program.

57

Liskov's Substitution Principle

Given:

```
public class Rectangle {
  ...
```

```
  public int getHeight() { ...  }
  public int getWidth() { ...  }
  public void setHeight(int height) { ...  }
  public void setWidth(int width) { ...  }
}
```

We might want to have another, more restricted class of *squares*.

Liskov's Substitution Principle

How about:

```
public class Square extends Rectangle {
  ...
  public int getHeight() { ...  }
  public int getWidth() { ...  }
  public void setHeight(int height) { ⟦enforce⟧ }
  public void setWidth(int width) { ⟦enforce⟧ }
}
```

Seems very reasonable relationship, since squares are rectangles.

But this violates the principle! Not each `Square` *is-a* `Rectangle`.

6. Anti-patterns

6.1 Things to watch out for

59

Code smells and anti-patterns

- **Code / design smell:** a surface indication that usually corresponds to a deeper problem in the system
- **Anti-pattern:** a common response to a recurring problem that is usually ineffective and potentially counterproductive
  (Also applicable outside software itself.)
- Contributes to technical debt.

60

Design / code smells

- **cyclical dependencies**

  Usually, modules depend on *lower-level* modules of some kind. Hence, when there is a cyclical dependency, such as class A depending on class B, B depending on C, and C depending on A, this is suggestive of a problem. If the modules are clearly closely connected to each other and should be seen as forming one larger structure, then it's probably fine, but otherwise, it seems likely that either some of these modules are combining functionalities that do not truly belong together (e.g., class A consists of A.1, which depends on B, and A.2, which is what C depends on), or that you are missing an abstraction (e.g., instead of directly depending on B, A could also use an interface which is *implemented* by B, and which does not have a dependency on C).

- **inappropriate use of inheritance**

  This is when inheritance is used for relations other than *is-a*.

- **data clumps (missing abstraction)**

  This is when the same group of parameters is passed to multiple functions, especially when it's a large group; e.g.,

```
int *parry_modifier(object defender,
        object attacker,
        object defense_weapon,
        object attack_weapon,
        int parry_defense_bonus,
```

```
        int distance,
        int give_feedback) {
  ...
}
```

This suggests that the parameters together are contributing to a shared concept, so you may want to put them together into a class such as "attack data".

- **duplicate code**

  Since you shouldn't duplicate code, this is obviously bad. Generally, copied code (especially large chunks, and especially if it is copied more than once) means you are missing a necessary generalisation or abstraction. For example, if there is a need to copy *part* of a function, then that part clearly does something that has value of its own, so should go into a different function instead. If the same few functions appear in multiple classes, then most likely there is a concept that captures these functions, which could be abstracted into its own module and used by the original classes using inheritance or (more likely) containment.

- **unclear naming**

  When the name for a class, function or even variable does not clearly indicate what it is for, then it is unlikely that this class/function/variable as a single responsibility.

- **contrived complexity**

  This is when someone uses a design pattern where simpler code may suffice. The software design patterns that we discussed in this lecture (and others from the original book, or available on the internet) are meant to help you solve problems in a way that provides high cohesion and low coupling. However, you should still consider carefully if they are the best solution for the situation at hand. The KISS principle takes priority over using a pattern!

- **God object**

  A God object is an object or module that is essentially too large. Such an object contains information about (and often runs) most of the program, and is coupled with most other program components. Such an object violates the single responsibility principle, and should typically be split into multiple components.

61

Anti-patterns

- **Yak shaving**

  Yak shaving describes any seemingly pointless activity which is actually necessary to solve a problem which solves a problem which, several levels of recursion later, solves the real problem you're working on. (MIT Media Lab) It is best exemplified by this story from Seth Godin:

"I want to wax the car today."

"Oops, the hose is still broken from the winter. I'll need to buy a new one at Home Depot."

"But Home Depot is on the other side of the Tappan Zee bridge and getting there without my EZPass is miserable because of the tolls."

"But, wait! I could borrow my neighbor's EZPass. . . "

"Bob won't lend me his EZPass until I return the mooshi pillow my son borrowed, though."

"And we haven't returned it because some of the stuffing fell out and we need to get some yak hair to restuff it."

And the next thing you know, you're at the zoo, shaving a yak, all so you can wax your car.

This can also happen when programming. You want to develop something, but you find a dependency on something else, and to do that properly you need to rewrite some other code, and. . . before you know it, you're at the zoo shaving a yak. :)

The solution to this is to recognise when you are finding yourself going down a rabbit hole, and seeing if you cannot cut it off earlier. In the car waxing story, the car would have gotten waxed a lot faster if the person had simply gotten a bucket of water and a sponge when he found the hose unavailable. Similarly in development, you can often achieve your goal – or something that *sufficiently* satisfies the client's criteria – by taking a slightly different route that doesn't require you to make drastic changes to your codebase.

That being said, it is worth examining the resason why it happens. If you cannot do some code in the obvious way because doing it properly requires a refactoring of some other code, and that refactoring causes you to go down a rabbit hole, it might be that your code Has Some Problems. And fixing those problems is worthwhile, even if that does not directly contribute to your goal. In the example, the person has a broken house and a difficult relation with his neighbour, which really should be resolved at some point. So, when you find that you cannot easily implement some feature because of problems in your code, you might need to shave that yak. Though, keep the sprint goal in mind:

  - note down the problems, but do not immediately start refactoring
  - finish implementing the sprint task (or: something sufficient, after discussing with others) in the easiest way possible, but without making the code actively worse
  - put fixing those problems on the product backlog, and if they really are in the way of doing the current feature properly, perhaps prioritise this over the low-priority / backup tasks in the sprint

- **yo-yo problem**

The yo-yo problem occurs when a developer, on reading code, keeps going up and down the inheritance chain to follow the flow of the code, and has to keep many classes in their head. This is a sign of bad use of inheritance, and that the code needs to be refactored.

- **coding by exception**

  Coding by exception occurs when error-handling is added only if a particular error actually arises. This often leads to lots of error-handling code for specific issues, when it would be better to take a step back, look at the bigger picture, and implement more general exception handling (which can often do be done with less code, while also covering the cases that have not yet arisen).

- **error hiding**

  Error hiding is the practice of ignoring errors: for example, an exception is caught but ignored, or a null-check for an argument that should be null just leads to a return rather than error being thrown. This practice can be tempting because you might not want your client to see errors, but this causes problems in your code to be missed, and makes it hard to debug the code when consequent issues arise.

- **boat anchor**

  A boat anchor is a piece of code that is no longer used, but still kept in the software. When code becomes redundant, many developers are reluctant to delete it, because effort was invested in that code, and it might become necessary later. However, keeping it around can cause confusion ("what is this used for?"), causes people to maintain it even though that is not necessary ("I am fixing all instances of query_weight; it appears here, so apparently I need to fix this class too"), and makes the code overall less clean. Moreover, even if you do eventually need it, the rest of the code has likely changed enough that it will need significant changes.

  The solution is simple:

  You Aren't Going To Need it! YAGNI!

  Delete the code. It's in your version control anyway, so you can always get it back later if it does turn out to be necessary.

  The phrase *boat anchor* is also used for a different anti-pattern, where a company or development team has invested a lot of resources (usually time or money) into some hardware or API and then feels obliged to stick with it even though it becomes more and more clear that this is not the best idea.

- **premature optimisation**

  There is no doubt that the holy grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified."

– Donald Knuth, 1974

Premature optimisation happens when you design the code to be more efficient – spending time and often making the code harder to read and understand – before there is a clear indication that the efficiency of a simpler design is problematic. For example, writing a function using iteration rather than recursion when recursion would be more clear, and it's never going to be a bottlebeck. Or implementing an elaborate A* algorithm for finding the most efficient path in a graph instead of a 5-line search function, when in practice the graph will always be small, and visualing it on the screen takes 10 times as much time as the simple function would.

- **Cargo cult programming**

  After WW2, several aboriginal religions sprung up in the South Pacific based around the cargo planes that previously brought supplies. In these religions, people would build fake planes in the hope that this would bring the gods.

  Cargo cult programming refers to including code without understanding it properly, and as a result likely ending up at least partially with code that serves no real purpose. This could be code copied from other projects or Stackoverflow, or it could be including design patterns that are not actually needed. Essentially: if you do not understand fully what certain code does and why it works, you should probably not be putting it in your project!

62

Management anti-patterns

Finally, we list some anti-patterns that often occur in project management. As with the anti-patterns and the code smells, you can easily find more information on them on the internet; the ones I checked had an excellent wikipedia page.

- **death march**: continuing a project that can be easily predicted to fail
- **feature creep**: adding more and more features that aren't necessary
- **ninety-nine rule**: underestimating remaining time on an "almost complete" project

  The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time. – Tom Cargill

- **management by objectives**: focusing on metrics rather than quality

  This often happens when you are judged by those metrics. For example, this is why we *tell* you about code quality tools like BetterCodeHub, but do not require groups to use it and demonstrate a high score.

- **seagull management**: having managers and employees in contact only when problems arise

---

63

Some sources

- *Design Patterns: Elements of Reusable Object-Oriented Software* – Gang of Four
- `https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53`
- `https://linux.ime.usp.br/~joaomm/mac499/arquivos/referencias/oodmetrics.pdf`
- `https://medium.freecodecamp.org/the-code-im-still-ashamed-of-e4c021dff55e`