Principles and Patterns

21 February, 2022

Lecture overview

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

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

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
- (If there is time): Code smells and Anti-patterns
  - code smell: indication that there is a problem with your code
  - anti-pattern: bad solutions for often recurring problems

PRINCIPLES

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

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

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!

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!

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

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!

- 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() { ... } }

A conflict: YAGNI versus DRY?

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

  Alternative:
  - void set_gram_weight(int number)
  - int query_gram_weight()
  - void copy_weight(object other)
- 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!

Risk-driven design

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.

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.

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

Fail Fast

The system should not fail.
If it does, you should want to know about it!

- Don't write code to work around systems failing that should not fail.
- Let the system fail, so the problem is caught during development or early deployment!
- Disclaimer: the wisdom of this principle depends on the kind of software. . .

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Limit Published Interfaces

Some principles to obtain a decoupled, cohesive design

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

Some principles to obtain a decoupled, cohesive design

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

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.

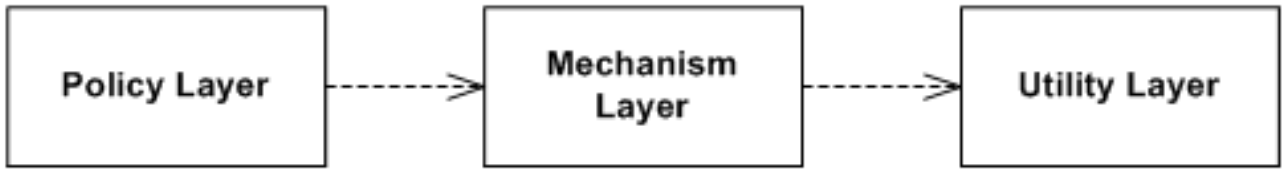
- A module/class should have only one reason to change.
- Bad example: a module that compiles and prints a report.
- Good example: a module that compiles a report.
- Good example: a module that is responsible for arithmetic reasoning.
  - large responsibility, but only one responsibility
  - may contain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

Some principles to obtain a decoupled, cohesive design

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

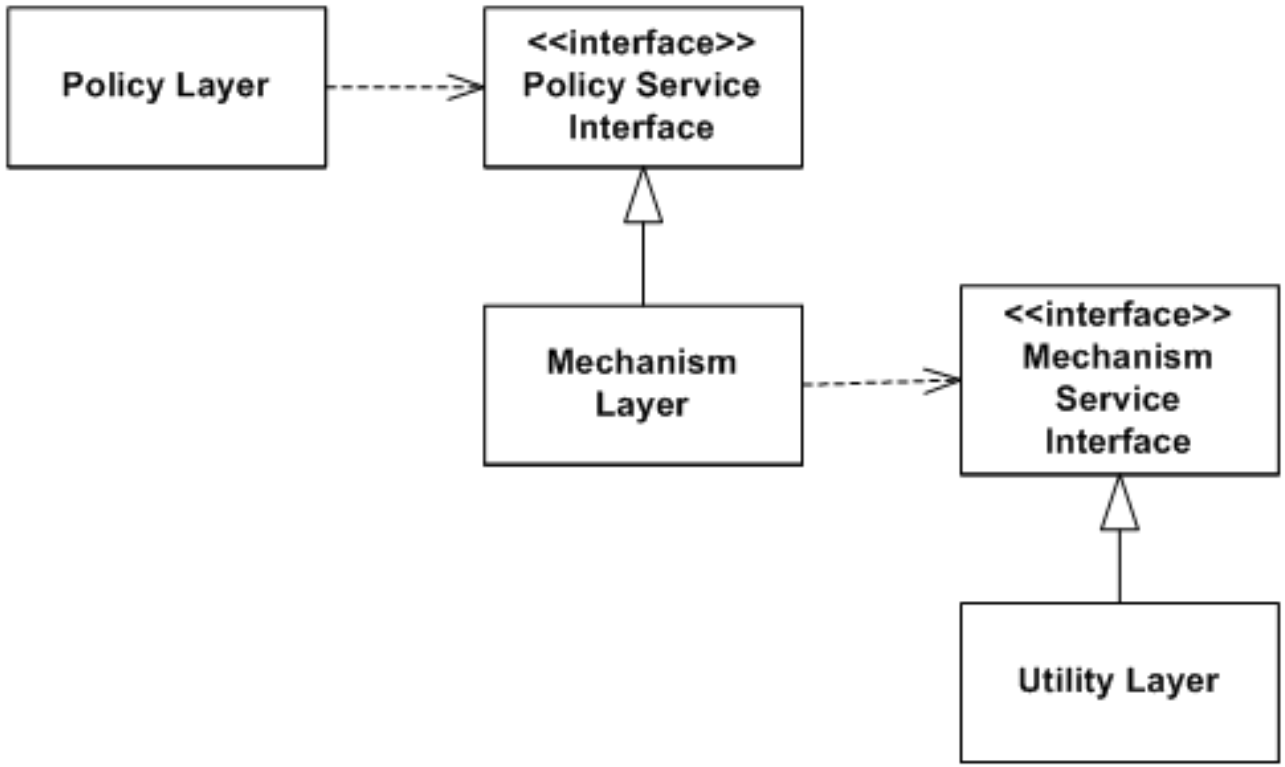
Dependency Inversion Principle

(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.



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.



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.

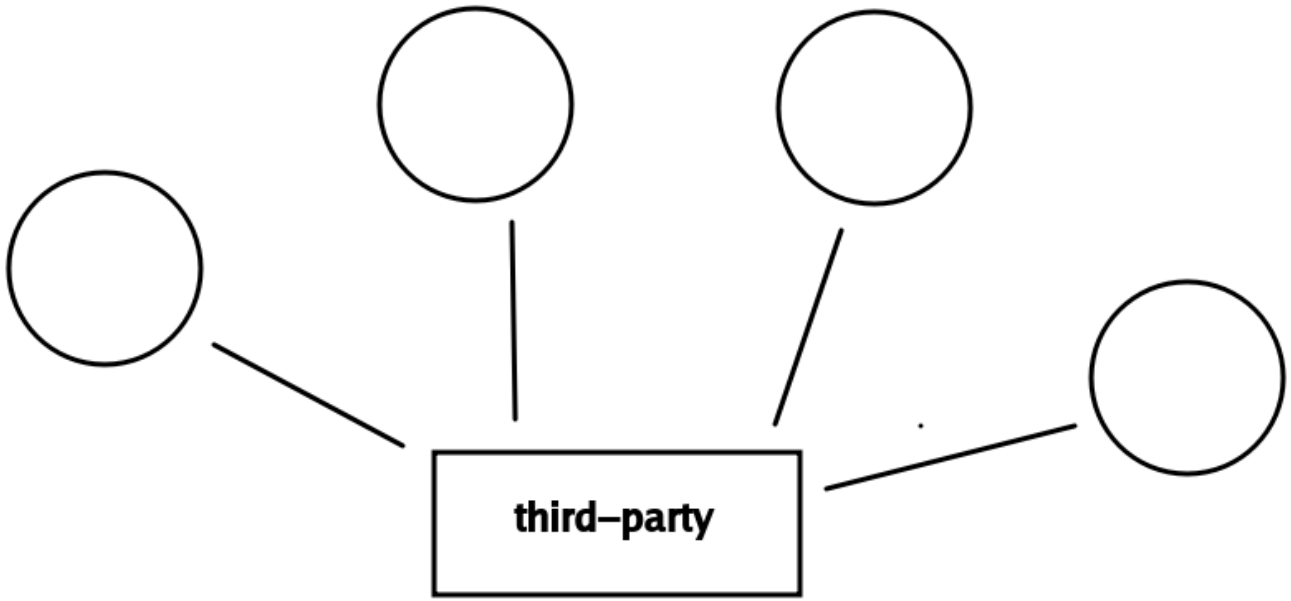
- Suppose high-level class A depends (via interaction coupling) on low-level class B.
- If a mechanism in B changes, we should not have to adapt A.
- Instead, we should have made an abstract interface B' on which A depends and which B implements.
- In essence, it becomes the role of B'' to capture the interaction aspect between A and B.

Some principles to obtain a decoupled, cohesive design

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

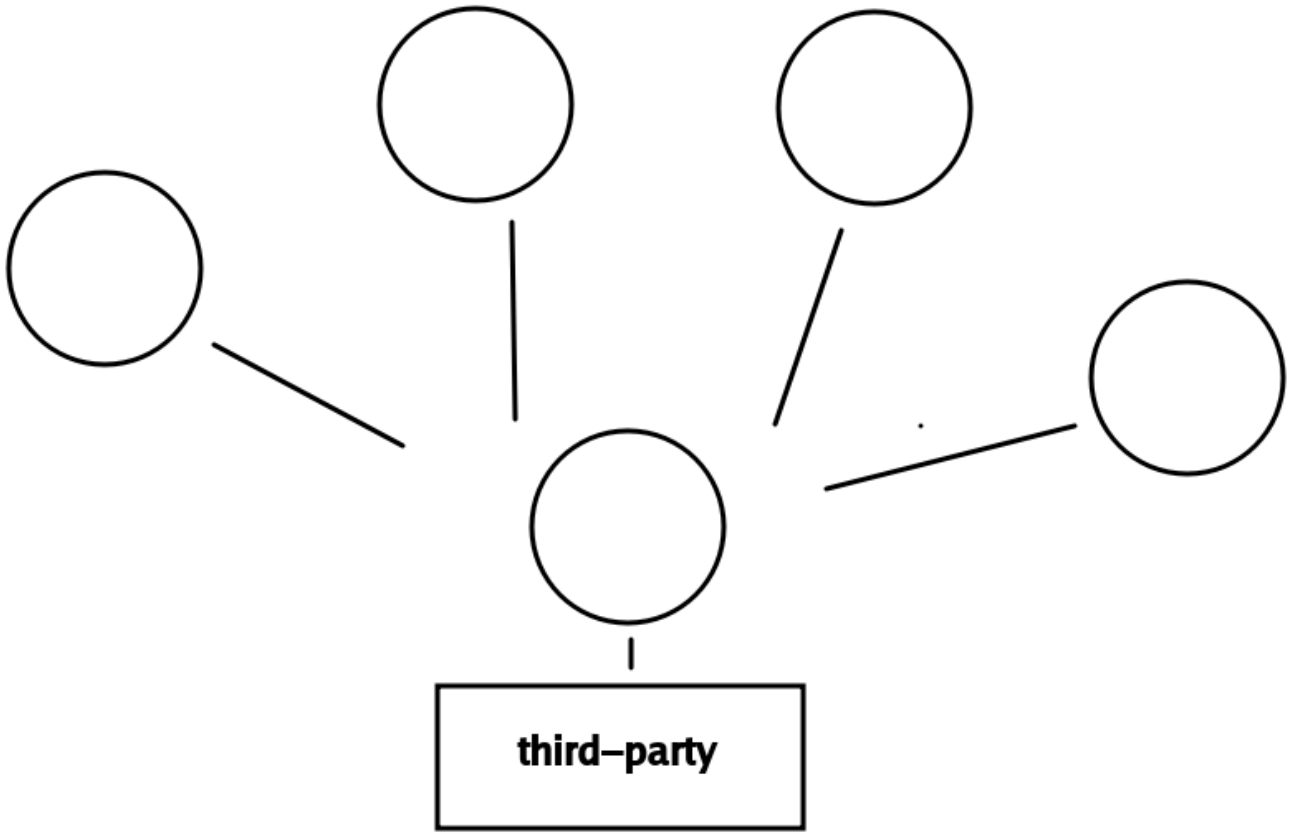
Isolate Third-party Components

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.



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.



Some principles to obtain a decoupled, cohesive design

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

Self-Documenting Code

```
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);
}
```

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);
}
```

PATTERNS

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

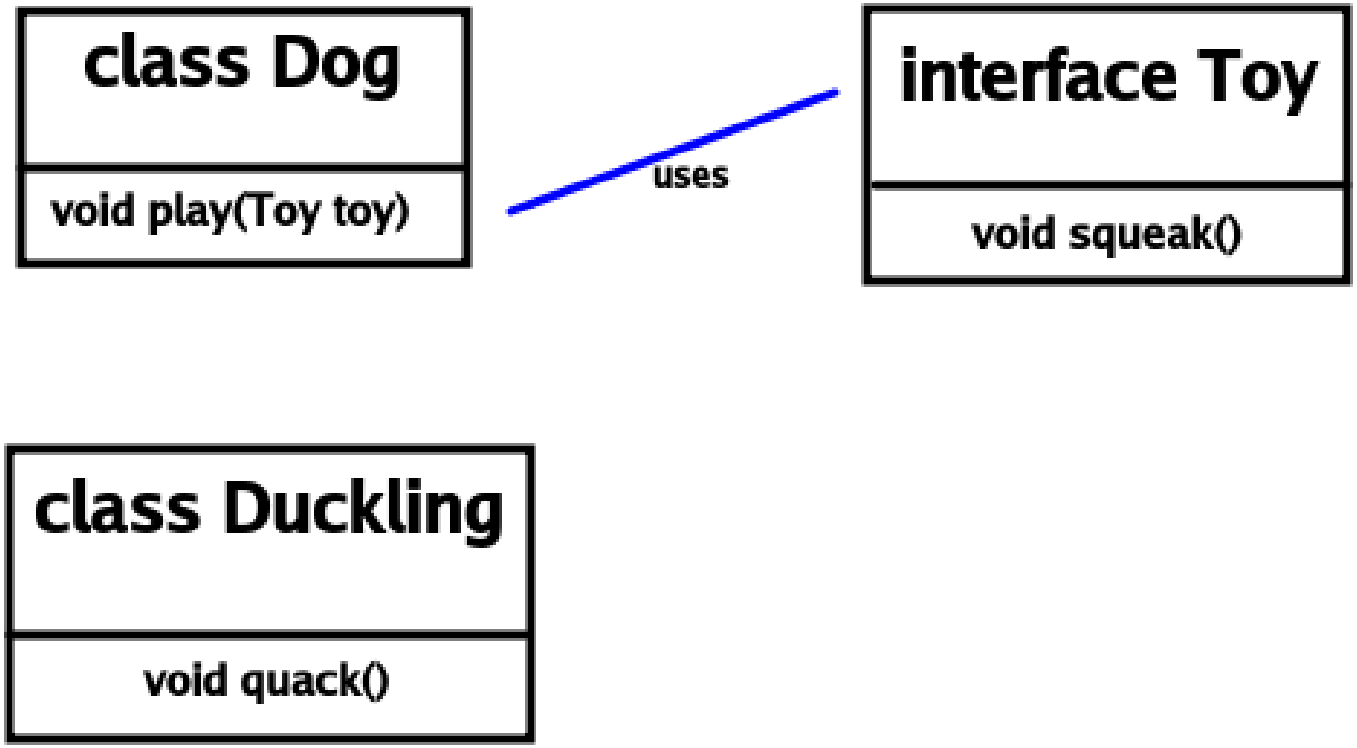
Some design patterns

- 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
  - Closely related: isolate third-party components!
- 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
- ...

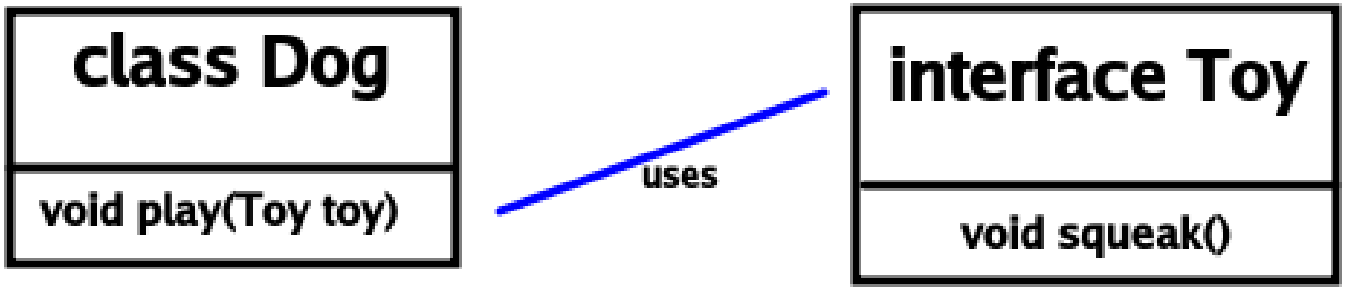
Some design patterns

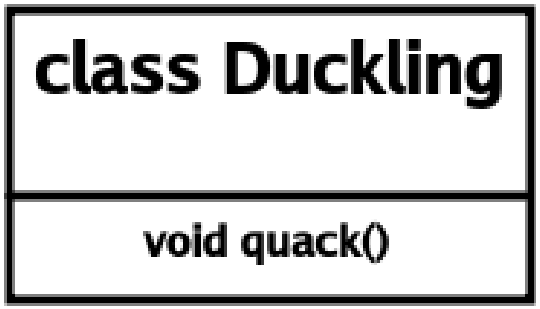
- Adapter pattern: use a wrapper to convert the interface of a class without modifying its source code
- Facade pattern: tidy up the interfaces to a number of related objects that have often been developed incrementally
  - Closely related: isolate third-party components!
- Observer pattern: tell several 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
- ...

Some design patterns



Some design patterns



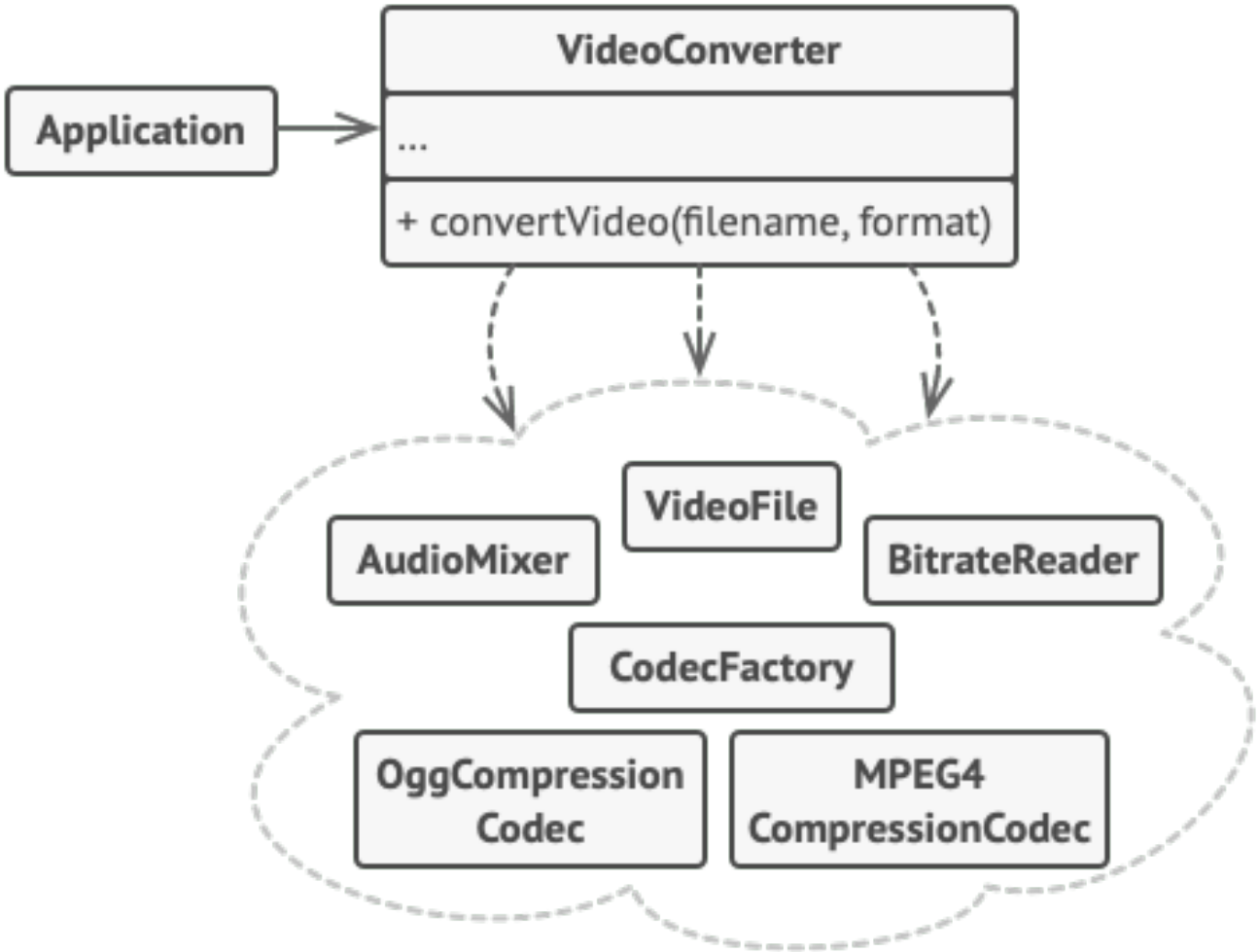



```
class DucklingAdapter
        implements Toy {
    Duckling duck;
    DucklingAdapter(Duck d) {
        duck = d;
    }
    void squeak() {
        duck.quack();
    }
}
```

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
  - Closely related: isolate third-party components!
- 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
- ...

Some design patterns



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
  - Closely related: isolate third-party components!
- 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
- ...

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
  - Closely related: isolate third-party components!
- 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
- ...

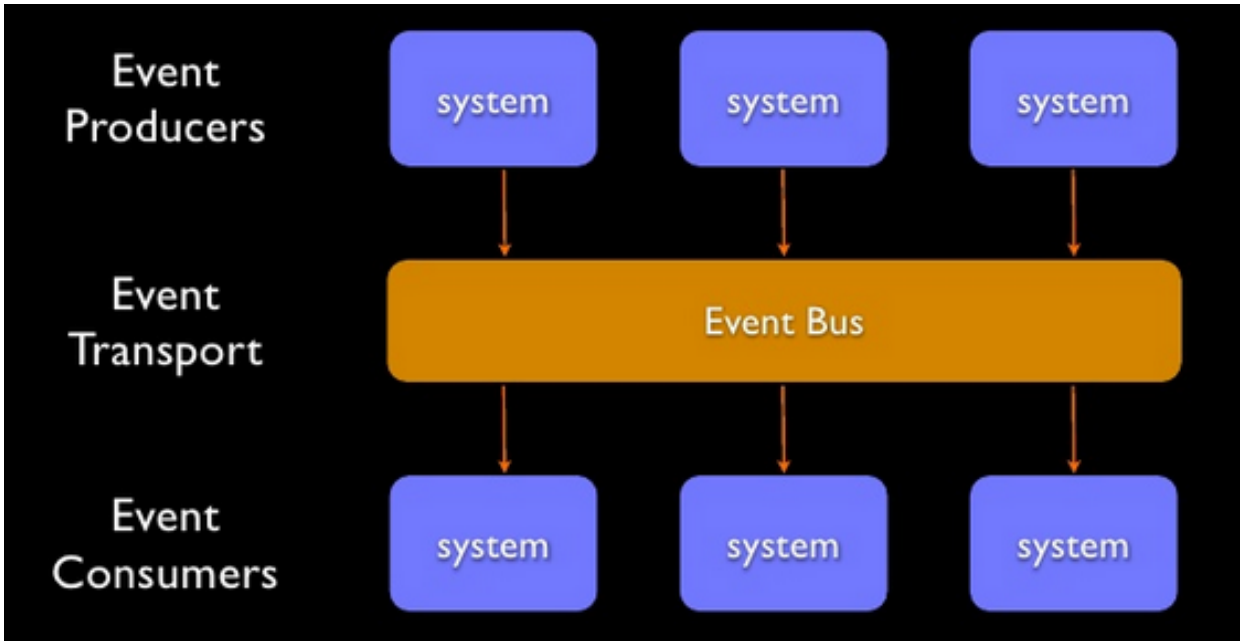
Observer Pattern

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

Observer Pattern

```
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(); }
}
```

Related: event-driven architecture pattern



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
  - Closely related: isolate third-party components!
- 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
- ...

Decorator pattern

- KeyComponentInterface
- KeyComponent implements 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))
- Example: windowing system with configurable borders, scrollbars

Example: a simple animation

```
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! **(8 combinations)**

- visible or invisible (difference in `draw`)
- moving or static blocks (difference in `update`)
- solid or immaterial (difference in `meet`)

Example: a simple animation

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

class BoringAnimObject implements AnimatedObject {
  public:
    void update() { }
    void draw() { }
    void meet(AnimatedObject other) { }
}
```

Example: a simple animation

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

Example: a simple animation

```
class VisibleAnimObject :  public DecorAnimObject {
  private:
    Model* model;
  public:
    VisibleAnimObject(AnimatedObject c) { super(c); }
    void draw() override { ⟦draw model⟧ }
}
class SolidAnimObject :  public DecorAnimObject {
  public:
    SolidAnimObject(AnimatedObject c) { super(c); }
    void meet(AnimatedObject other) {
      ⟦handle collision⟧
    }
}
```

Example: a simple animation

Challenge: how to get a solid movable block?

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

Exercise: a Pacman game

```
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!

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, use an abstraction instead
- abstractions play a critical role in several patterns

Modern languages typically have interfaces or abstract classes.
Even if not: most of this can be done through functions.

Testability

Principles, Patterns and Testability

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

- Single Responsibility Principle

The Single Responsibility Principle and Testability

```
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);
  }
}
```

The Single Responsibility Principle and Testability

```
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);
}
```

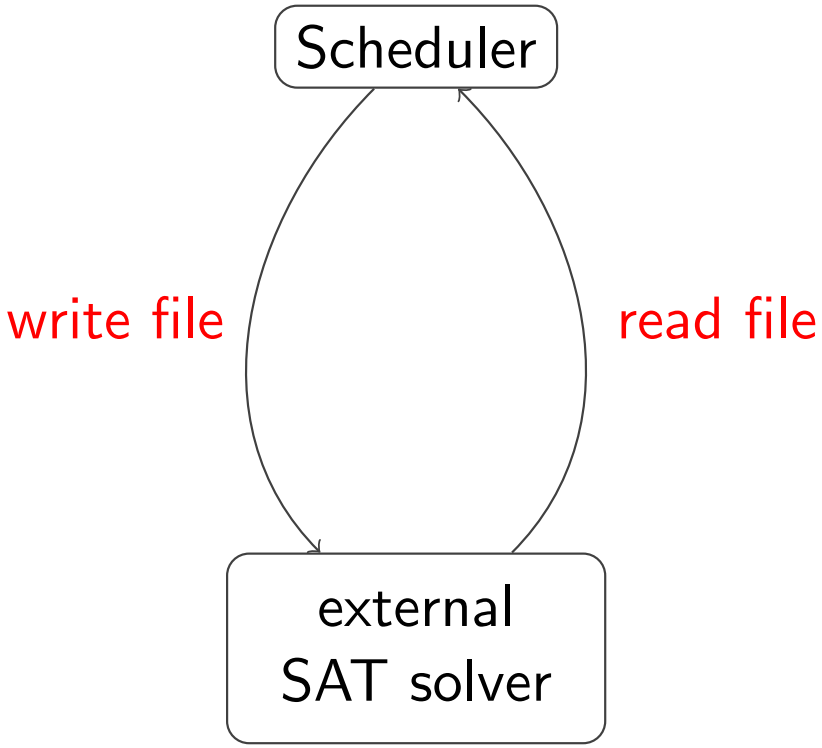
Principles, Patterns and Testability

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

- Single Responsibility Principle
- Isolate Third-party Components

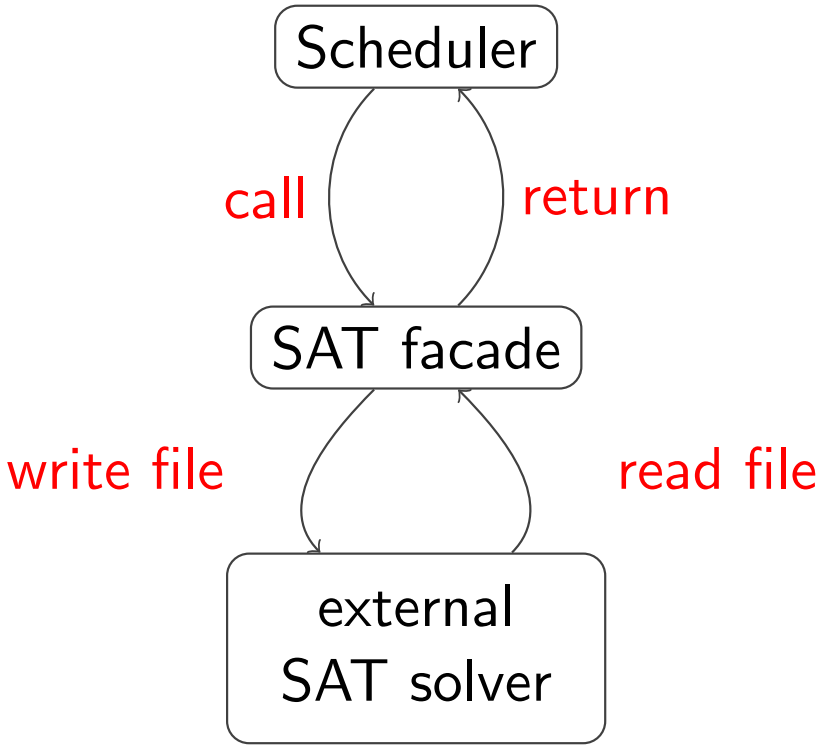
Isolating third-party components for testability

Hard to test:



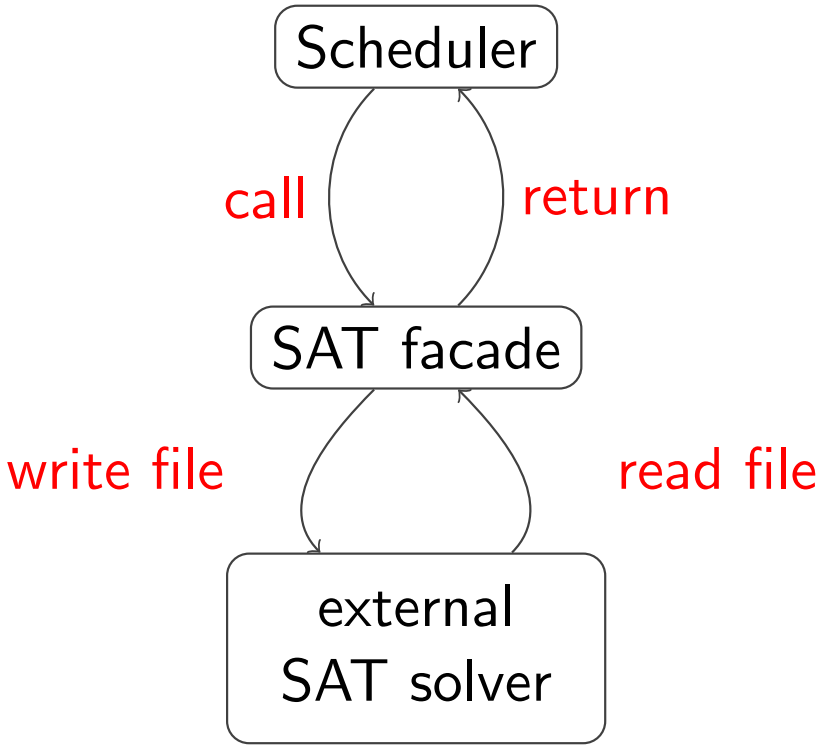
Isolating third-party components for testability

Easier to test:



Isolating third-party components for testability

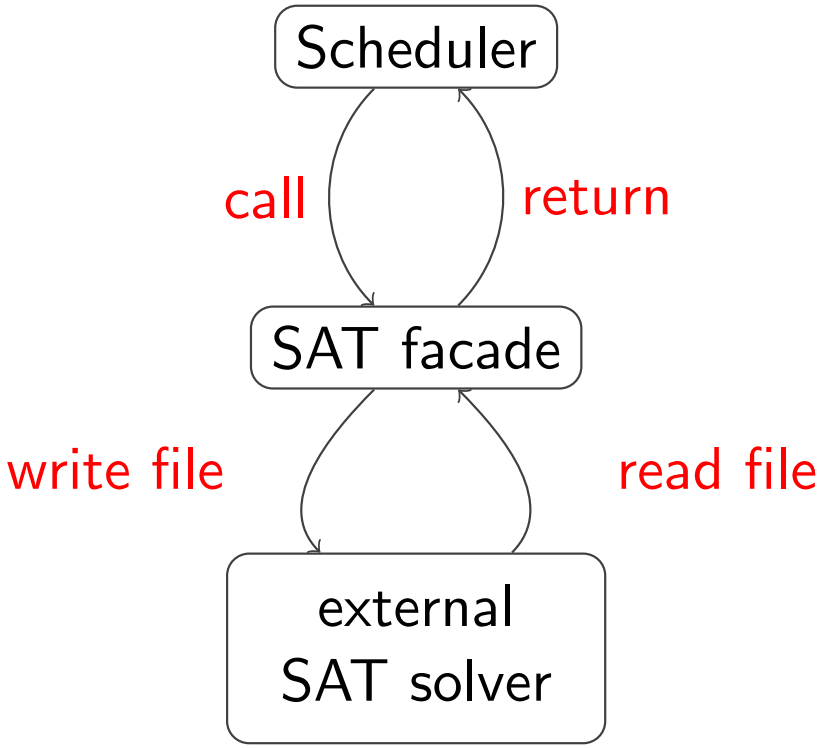
Easier to test:



$\Longleftarrow$ manually run test

Isolating third-party components for testability

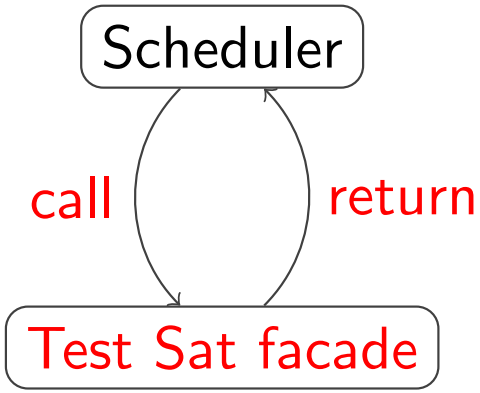
Easier to test:



$\Longleftarrow$ test automatically in isolation

Isolating third-party components for testability

Easier to test:



$\Longleftarrow$ test automatically in isolation

Testing using fake objects

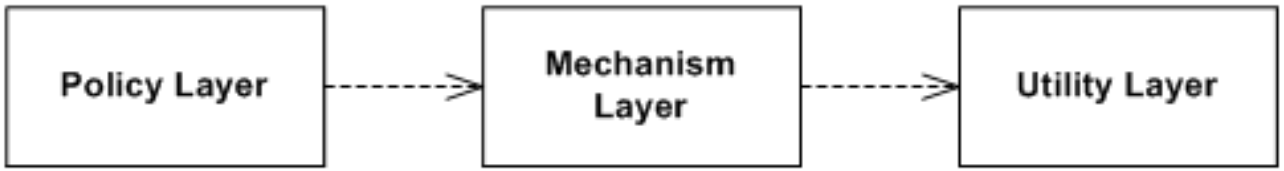
```
class TestSatSolver implements SatSolver {
  ArrayList<String> reqs;
  TestSatSolver() { reqs = new ArrayList<String>(); }
  public void addClause(Clause clause) {
    reqs.add(clause.toString());
  }
  public Solution solve() {
    return new Solution(false);
  }
}

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

Principles, Patterns and Testability

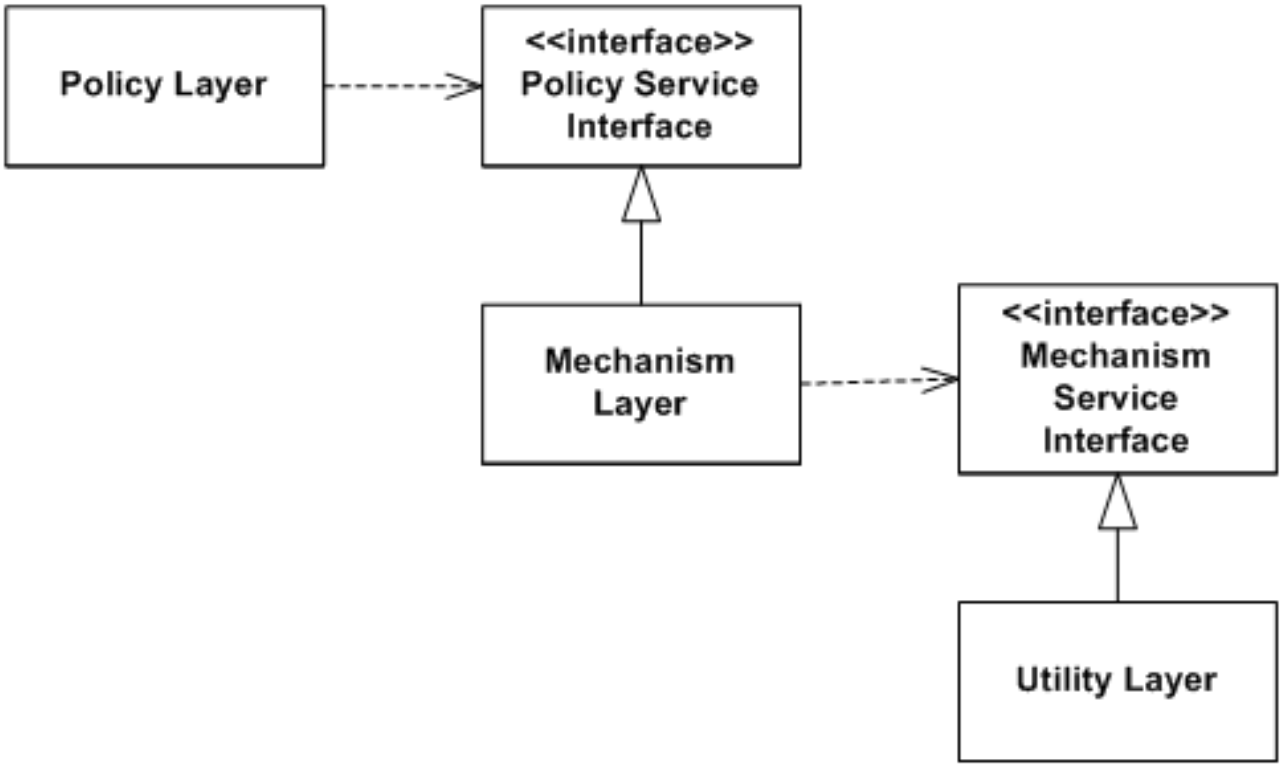
Observation: code with loose coupling and high cohesion is easier to unit test!

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

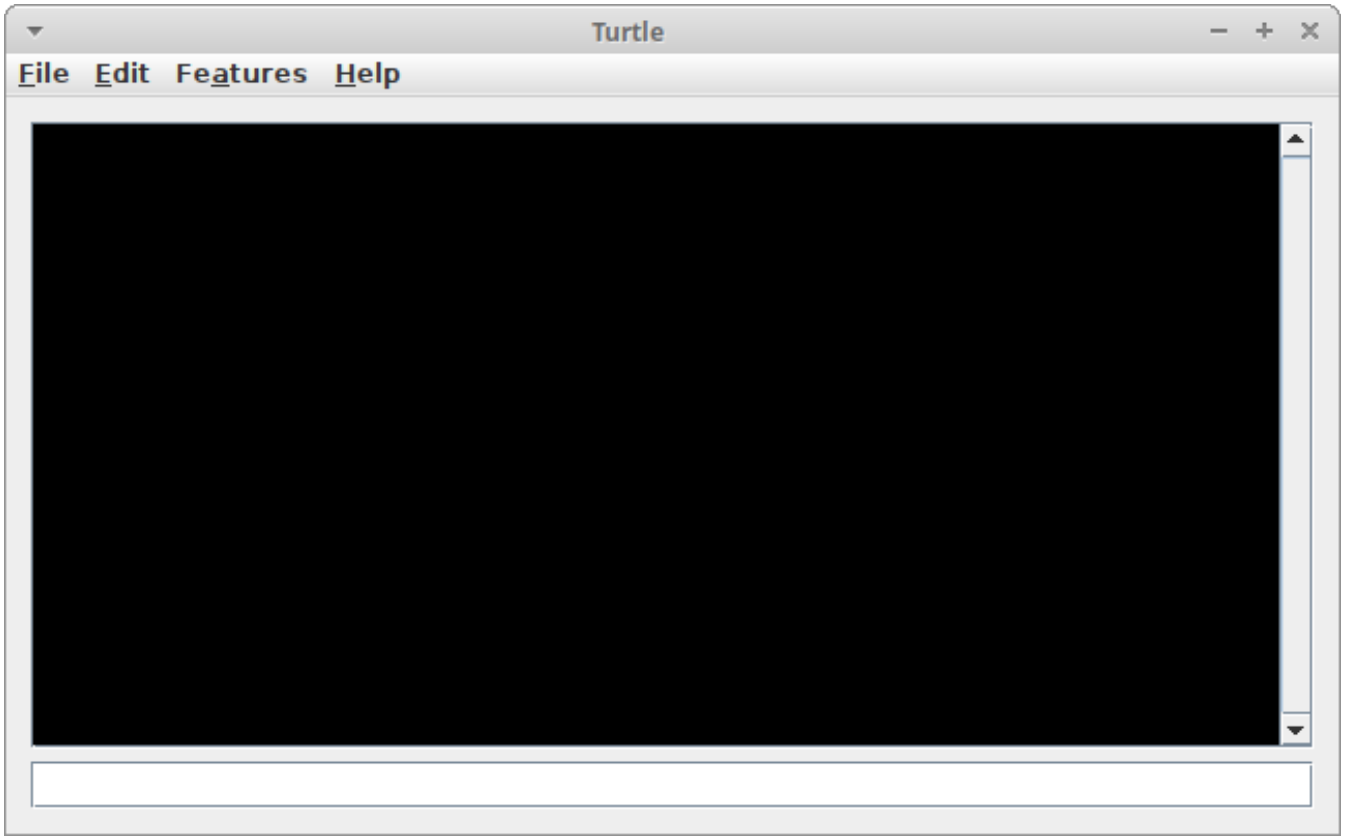
Testability by depending on abstractions



Testability by depending on abstractions



Class exercise: designing testable code



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.)

Class exercise: designing testable code

Naive implementation:

- Put it all in one class “input window”.

Class exercise: designing testable code

Naive implementation:

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

Better implementation:

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.

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:

Class exercise: designing testable code

Naive implementation:

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

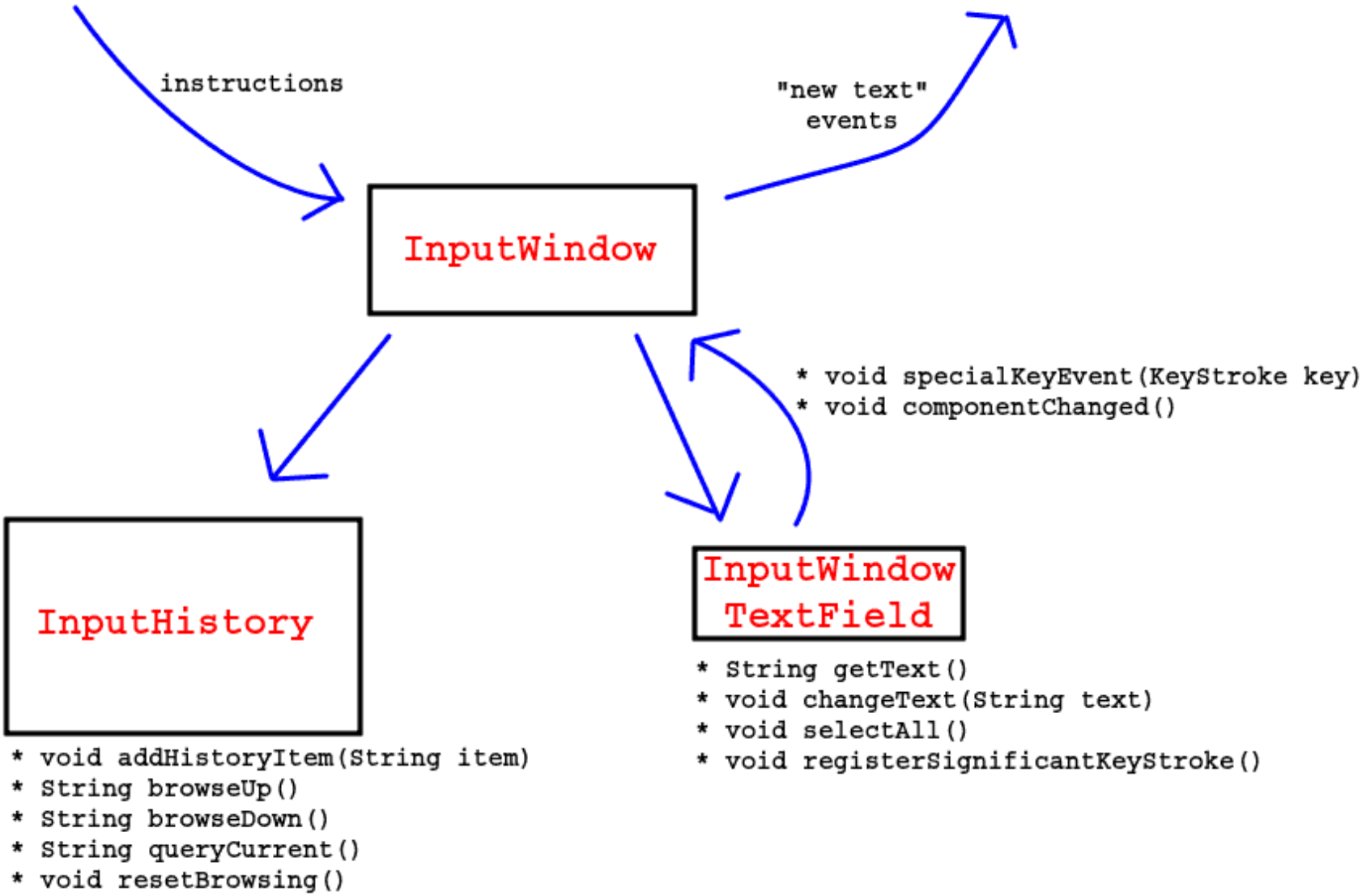
Better implementation:

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

Best implementation:

- Also separate the actual GUI component!

Class exercise: designing testable code



Class exercise: designing testable code

Note:

- 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, stub or mock objects. For example: call InputWindow.specialKeyEvent to indicate that return was pressed, and test whether it sends the text in the textfield stub on the event bus and passes it into the history spy.

Class exercise: designing testable code

Note:

- The InputWindowTextField is hard to test. Depending on your test framework, this may require manual testing. This can still be done systematically: define manual tests, 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 only minimal testing.
- 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.

Inheritance

Inheritance. . . awesome!

Inheritance. . . awesome?

Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

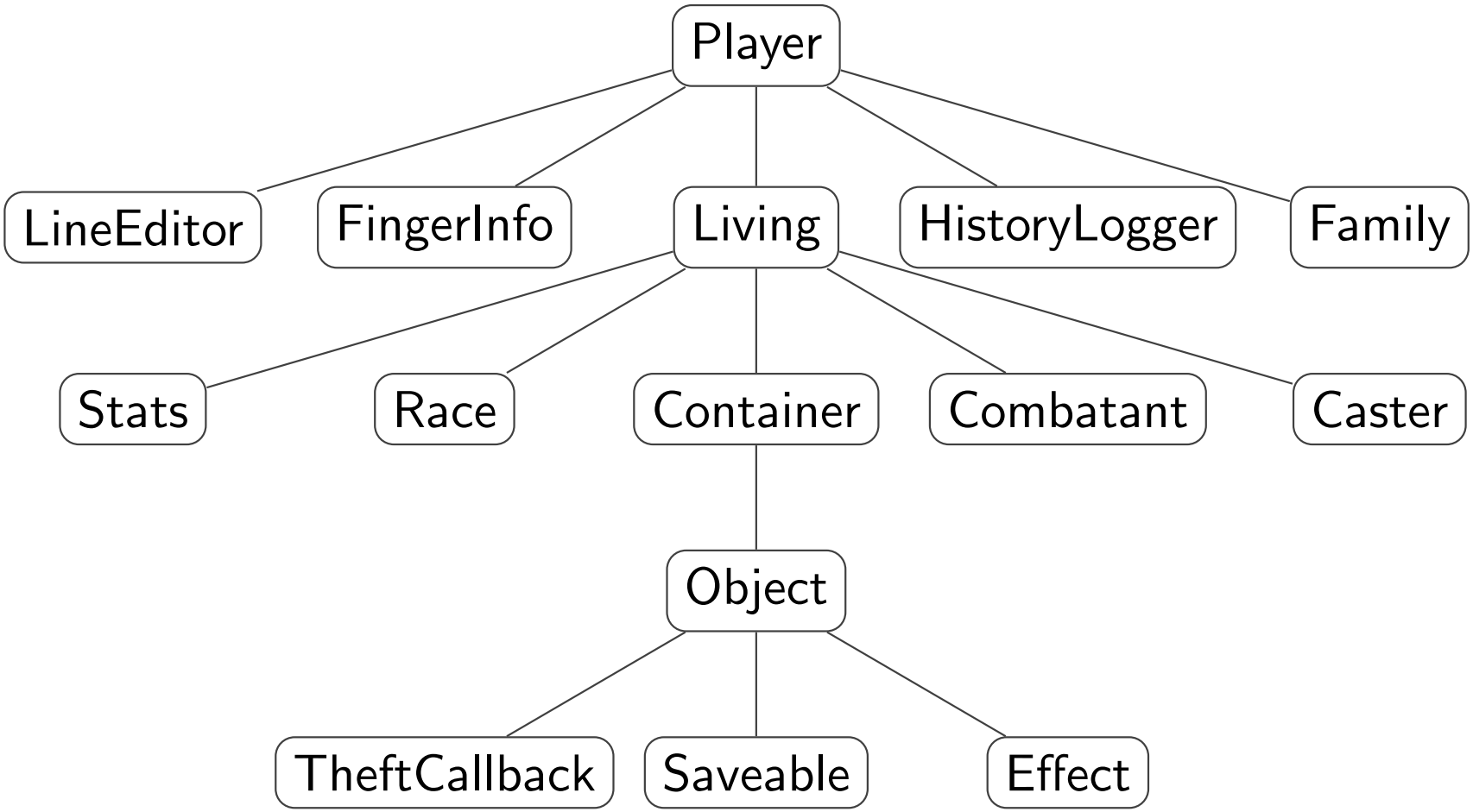
Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

Hierarchy depth

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

Hierarchy depth

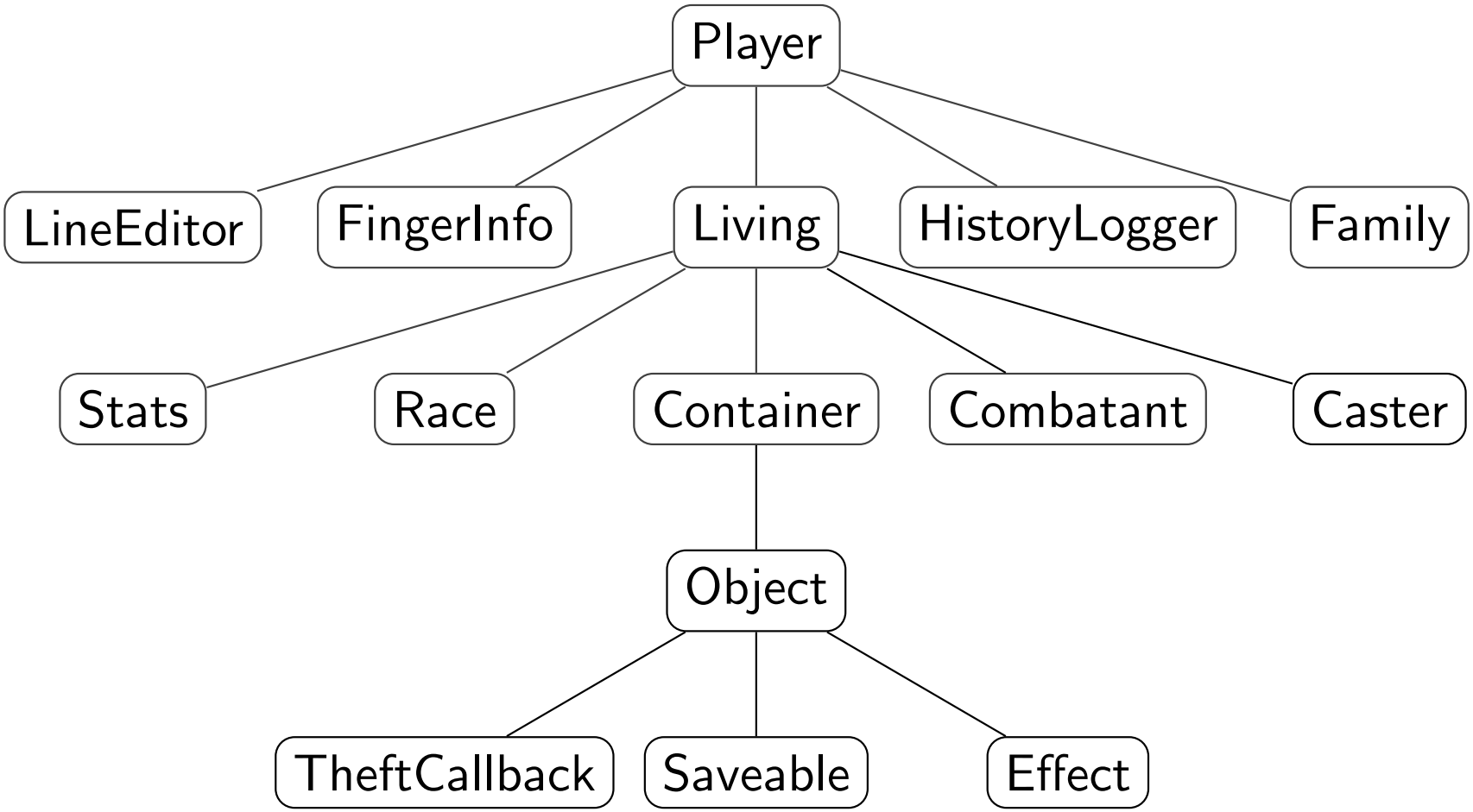


Inheritance. . . awesome?

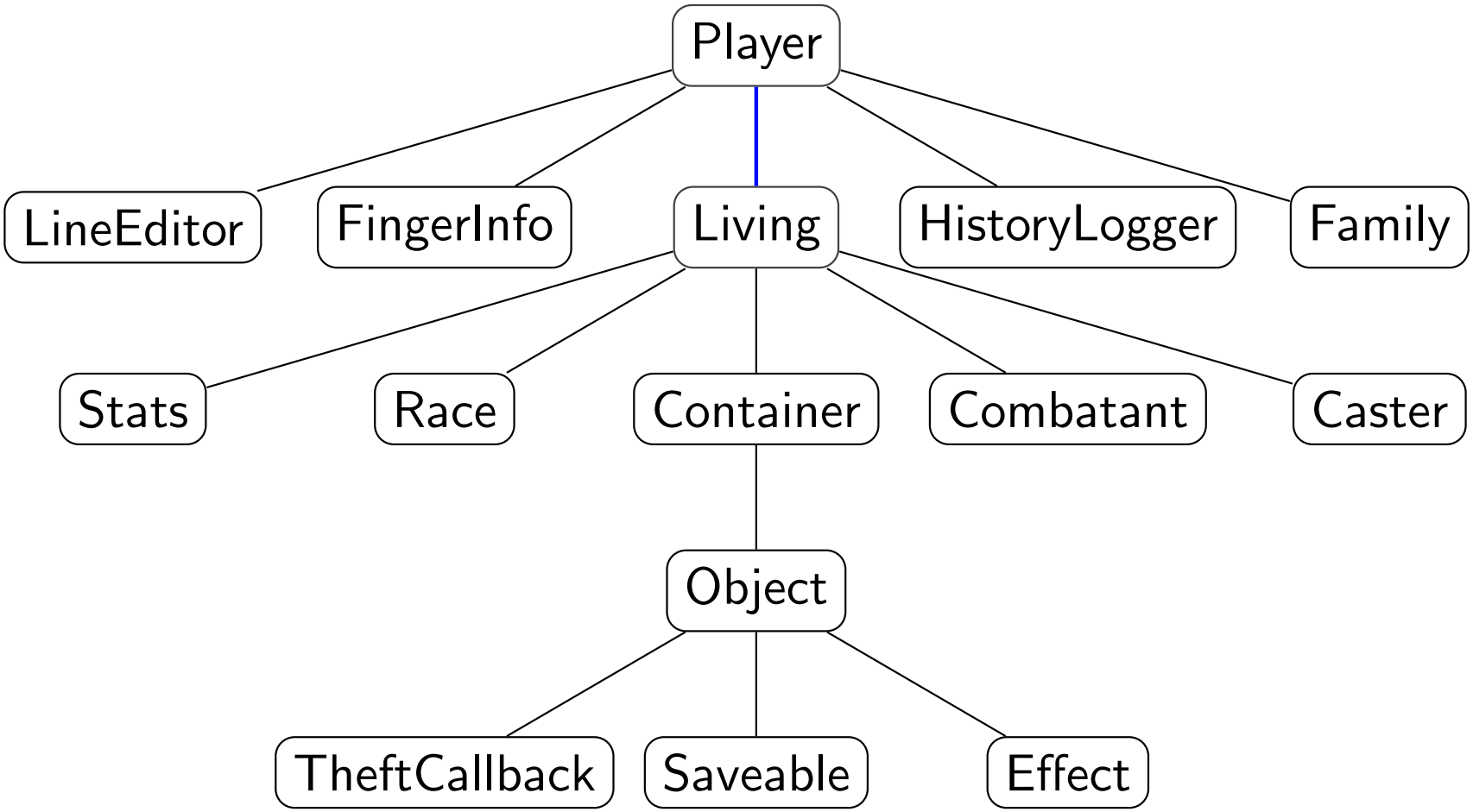
The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

– Joe Armstrong (creator of Erlang)

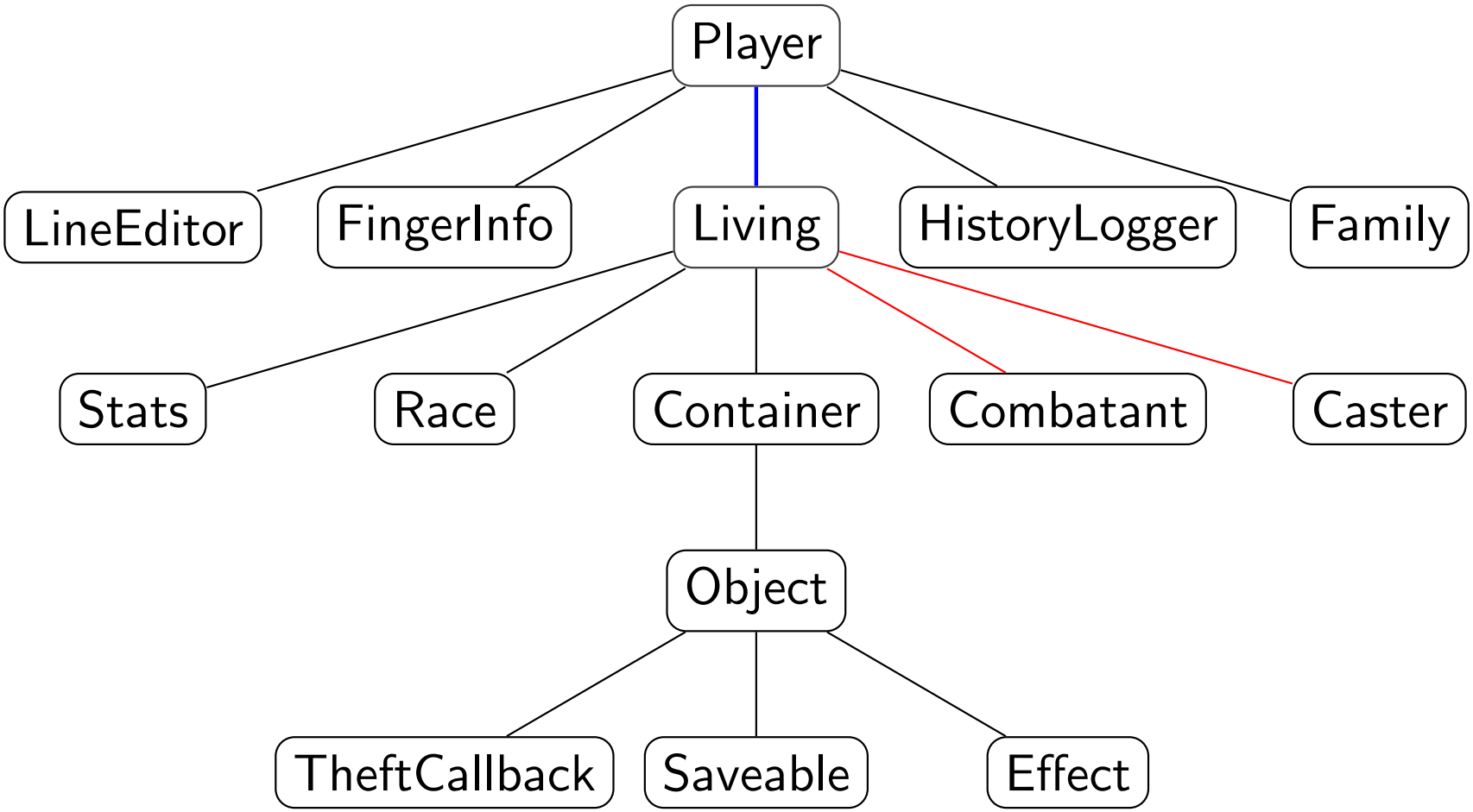
A deep hierarchy?



A deep hierarchy?



A deep hierarchy?



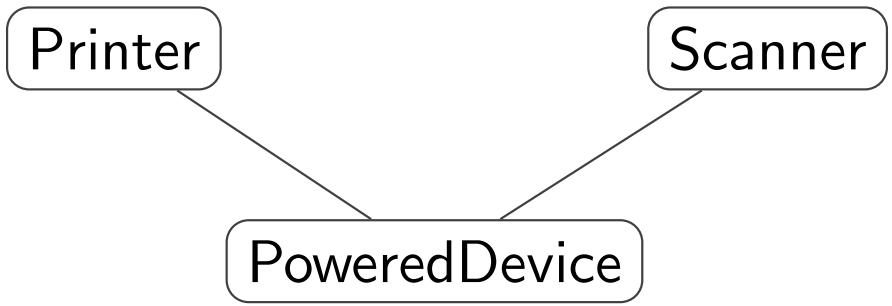
A deep hierarchy?

- replacing inheritance by containment does not fix this problem: to get the monkey we *still* need to get the whole jungle
- however, it keeps the interface cleaner, and thus makes it much easier to see that we can remove certain parts if we are not interested in the corresponding behaviour

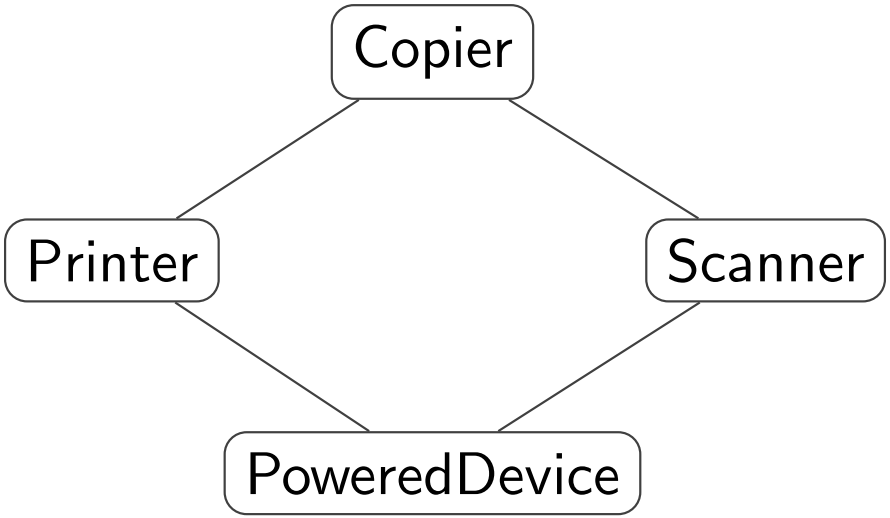
Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

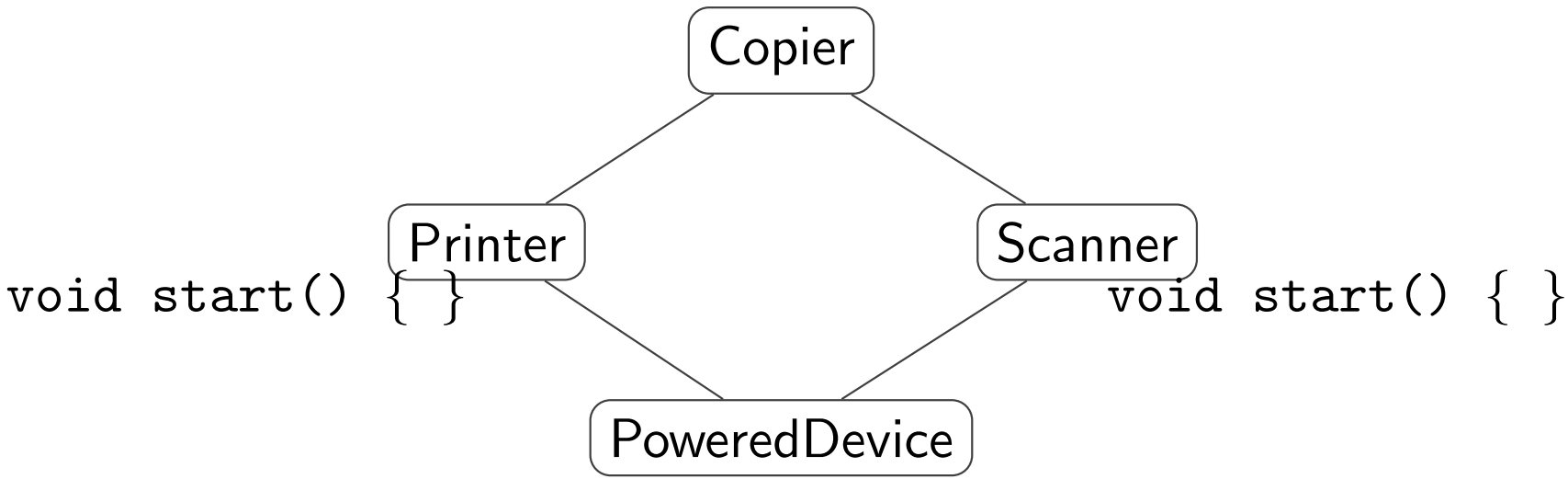
The diamond problem



The diamond problem



The diamond problem



The diamond solution

- Just don't do that.
- Instead, `Copier` contains an instance of `Printer` and an instance of `Scanner`, and can forward queries (e.g., `void start() { scanner.start(); printer.start(); }`).
- Or indeed: use a Decorator!

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));
  }
  ```

Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

Fragile base classes

```
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]);
  }
}
```

Fragile base classes

```
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;
  }
}
```

Fragile base classes

```
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]);
  }
}
```

Fragile base classes

```
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]);
  }
}
```

Fragile base classes

```
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]);
  }
}
```

Fragile base classes

- A possible solution: use a convention to never call public / non-final methods in the same class unless specifically indicated.
- Alternative: contain and delegate.

Inheritance. . . awesome?

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

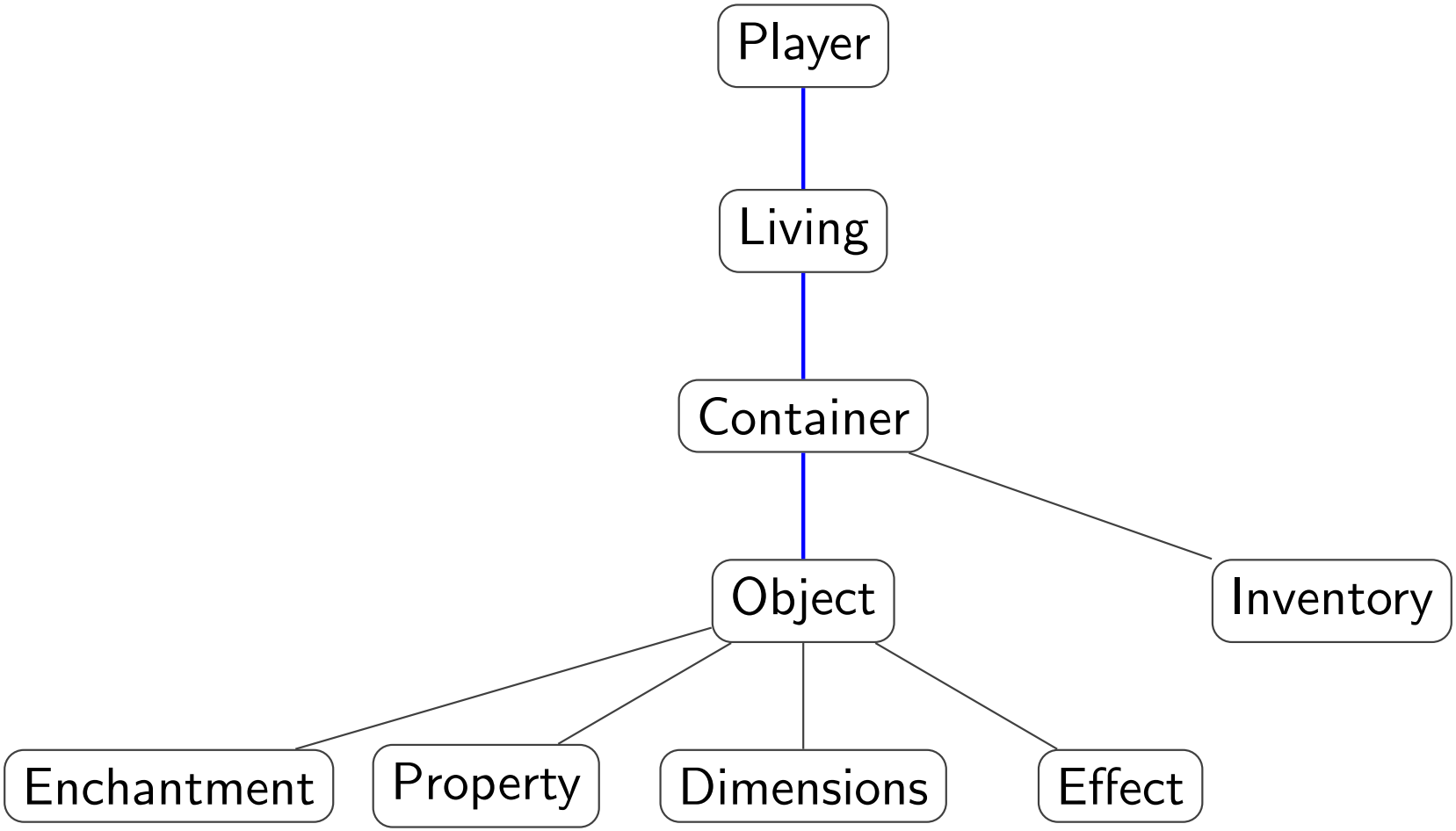
Composition over inheritance

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

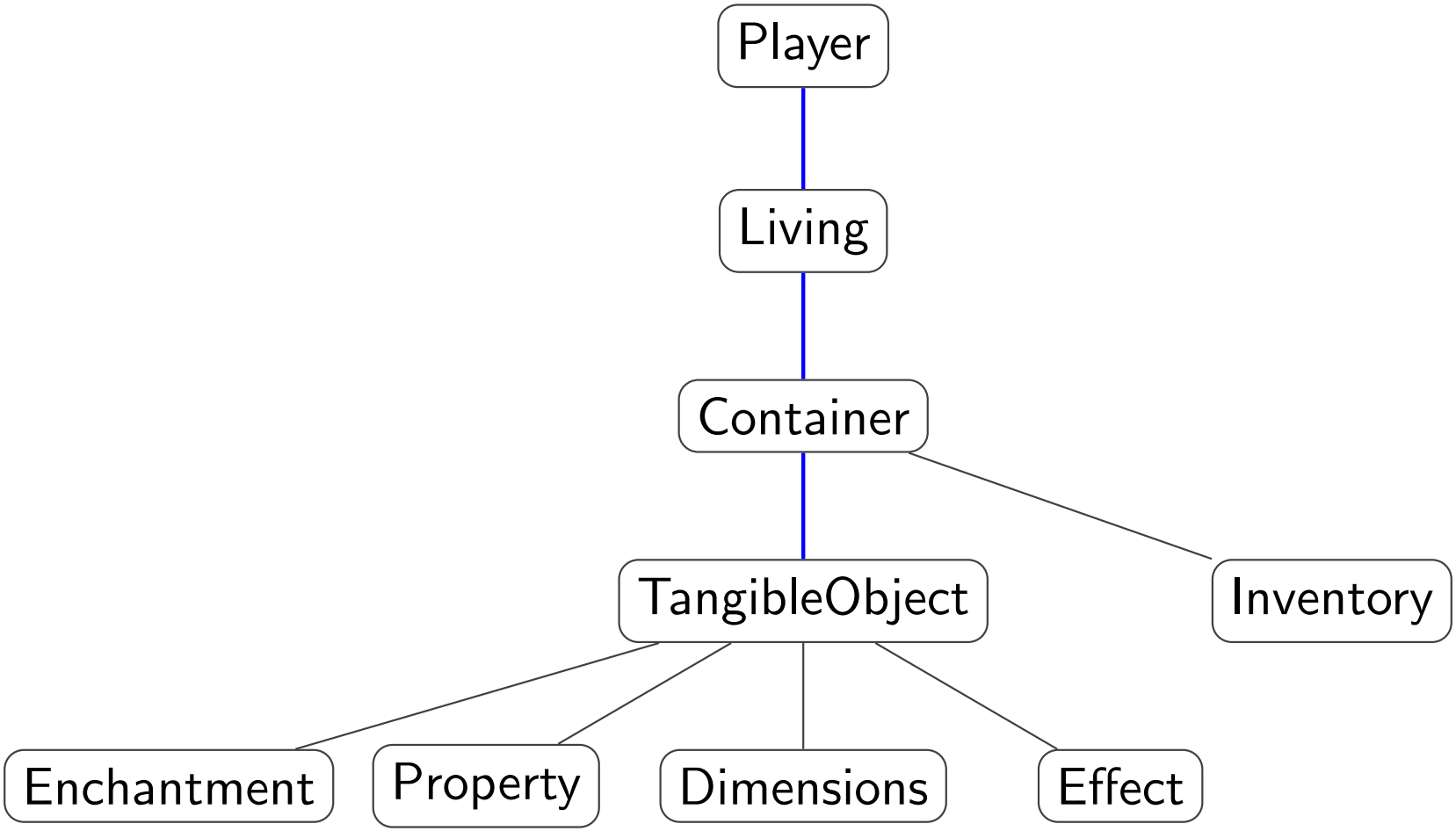
– Gang of Four

- Inheritance is white-box reuse, composition is black-box reuse.
- Interfaces offer all the advantages of polymorphism.
  (And great flexibility!)
- Delegation is a powerful method, which can often replace inheritance.
- Annoying in some popular languages. Do it regardless.
- Note: not a blanket "inheritance is bad"!
  - Inheritance is important, particularly for specialisation.
  - Inheritance is, however, overused.

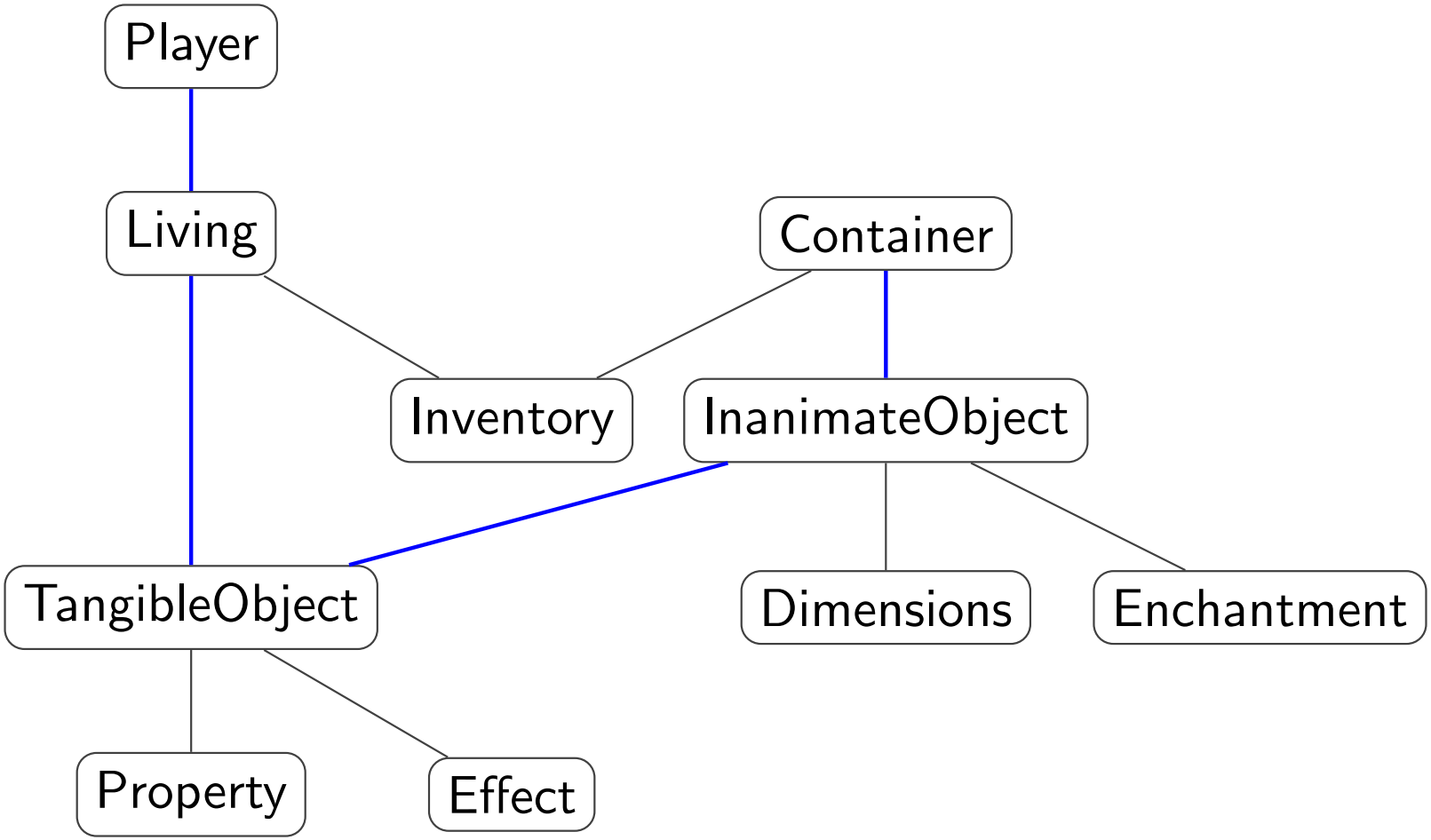
Exercise: the Discworld player object



Exercise: the Discworld player object



Exercise: the Discworld player object



Liskov's Substitution Principle

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

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

Smells and Anti-patterns

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.

Design / code smells

- cyclical dependencies
- inappropriate use of inheritance
- data clumps (missing abstraction)
- duplicate code
- unclear naming
- contrived complexity
- God object

Boy Scout Rule

Leave the campground cleaner than you found it.

- Leave no unit level code smells behind.
- Leave no bad comments behind.
- Leave no code in comments behind.
- Leave no dead code behind.
- Leave no long identifier names behind.
- Leave no magic constants behind.
- Leave no badly handled exceptions behind.

Anti-patterns

- yo-yo problem
- coding by exception
- error hiding
- boat anchor
- premature optimisation
- Cargo cult programming
- Yak shaving

Management anti-patterns

- 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
- management by objectives: focusing on metrics rather than quality
- seagull management: having managers and employees in contact only when problems arise

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`