Principles and Patterns

26 February, 2020

Lecture overview

- agile development
- agile design
- principles and patterns
- software testing

Lecture overview

- agile development
- agile design
- principles and patterns
- software testing

Recap: agile development

- A mindset:
  - Adaptability: be prepared for changing requirements; react to changes in the world
  - Communication: work closely with the client, communicate in-team all the time
  - Respect and responsibility: developers have input on planning and priorities, but are responsible for their results
- Scrum

Recap: agile design

- How to write code that supports changing requirements?
  - information hiding
  - low coupling
  - high cohesion
- 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
- Anti-patterns: bad solutions
  - often recurring problems have bad solutions that are often used

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Self-Documenting Code
- Isolate Third-Party Components
- Limit Published Interfaces

The SOLID principles

- Single Responsibility Principle
- Open-Closed Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

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".

The Open-Closed Principle

Entities should be open for extension, but closed for modification.

- Extension should not involve/require changing existing code.
- Original meaning: use inheritance!
- Meaning changed as programming languages and methodologies developed.
- Now: code should rely mostly on interfaces and abstract classes, which are open for extension. Implementations need not be.

The Open-Closed Principle

```
class GraphicEditor {
  public void drawShape(Shape s) {
    if (s.queryType() == 1) drawRectangle((Rectangle)s);
    else if (s.queryType() == 2) drawCircle((Circle)s);
  }
  public void drawCircle(Circle r) { ...  }
  public void drawRectangle(Rectangle r) { ...  }
}
interface Shape { int queryType(); }
class Rectangle implements Shape {
  ...
  int queryType() { return 1; }
}
class Circle extends Shape {
  ...
  int queryType() { return 2; }
}
```

The Open-Closed Principle

```
class GraphicEditor {
  public void drawShape(Shape s) {
    s.draw(_myCanvas);   }
}

interface Shape {
  public void draw(Canvas canvas);
}

class Rectangle implements Shape {
  ...
  public void draw(Canvas canvas);
}
class Circle implements Shape {
  ...
  public void draw(Canvas canvas);
}
```

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`.

Interface Segregation Principle

No client should be forced to depend on methods it does not use.

- Accomplish by splitting large interfaces into role interfaces.

Interface Segregation Principle

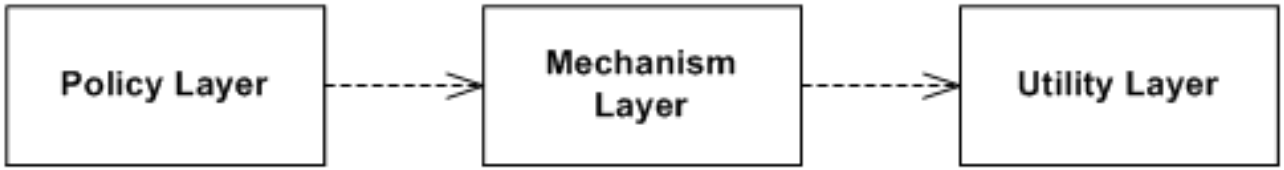
```
public interface Vehicle {
  void drive();
  void refuel(int amount);
}

public class Car implements Vehicle {
  void drive() { ... }
  void refuel(int amount) { ... }
}

public class Bike implements Vehicle {
  void drive() { ... }
  void refuel(int amount) { throw new Exception(); }
}
```

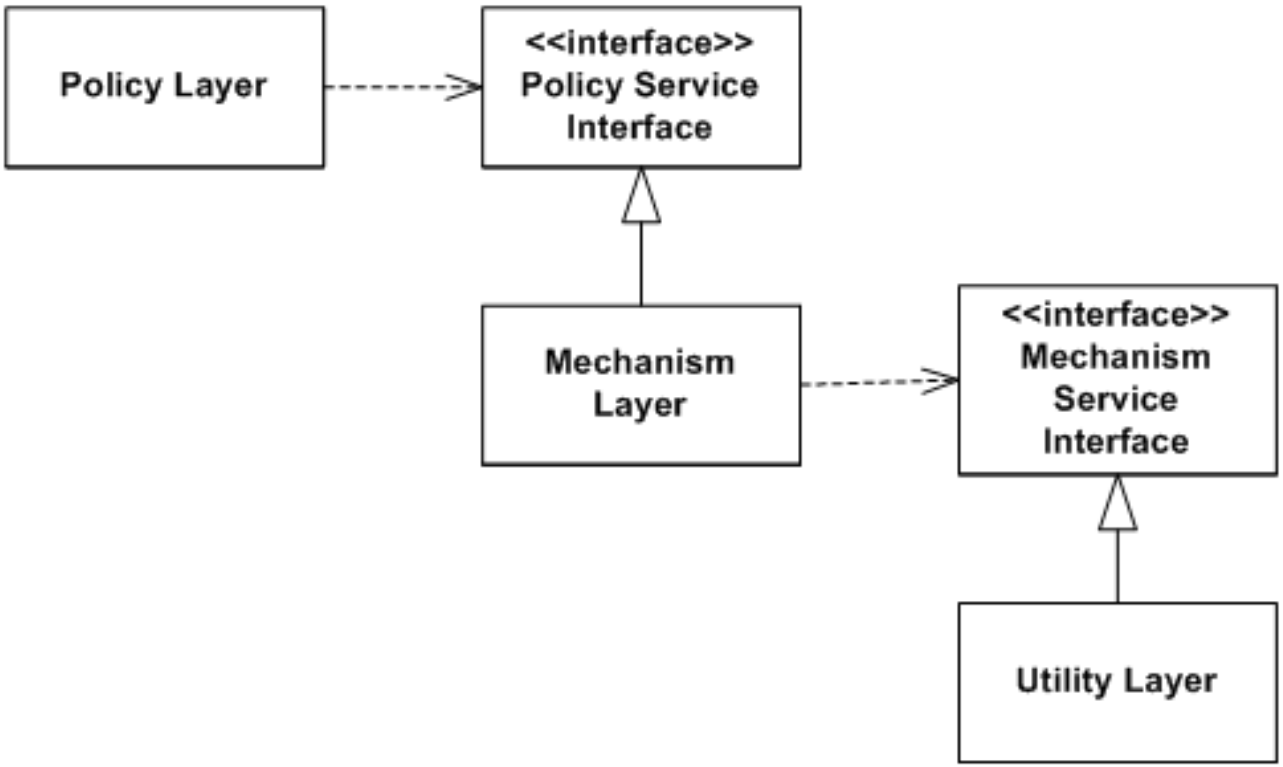
Dependency Inversion Principle

(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.

The SOLID principles

- Single Responsibility Principle
- Open-Closed Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Honourable mention: Law of Demeter

Don't talk to strangers!

Method `f` of class `A` may only talk to:

- `A` itself
- variables of `A`
- global variables
- the parameters to `f`
- any objects created within `f`

Alternative formulation: use only one dot.

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

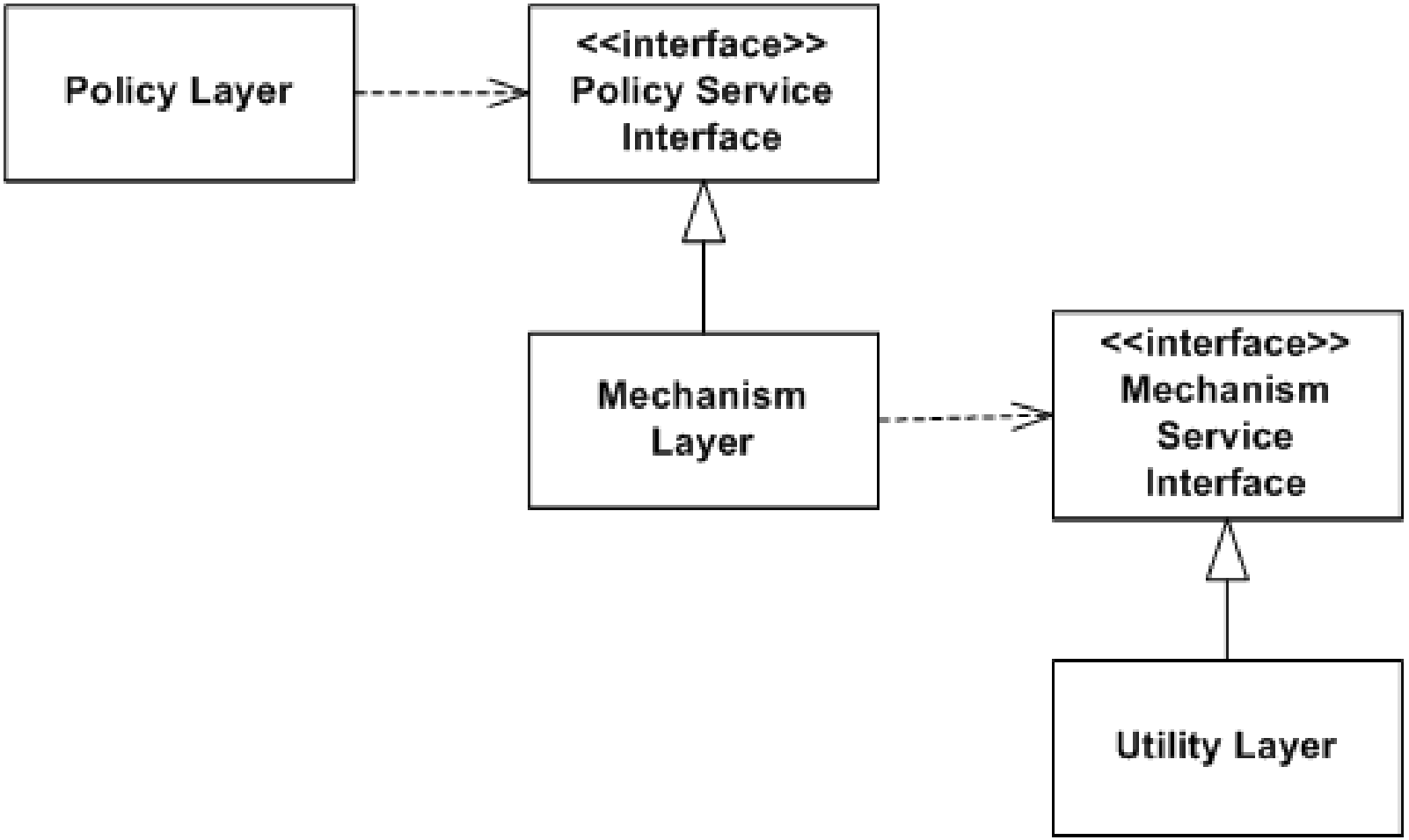
Some design patterns

- 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
- ...

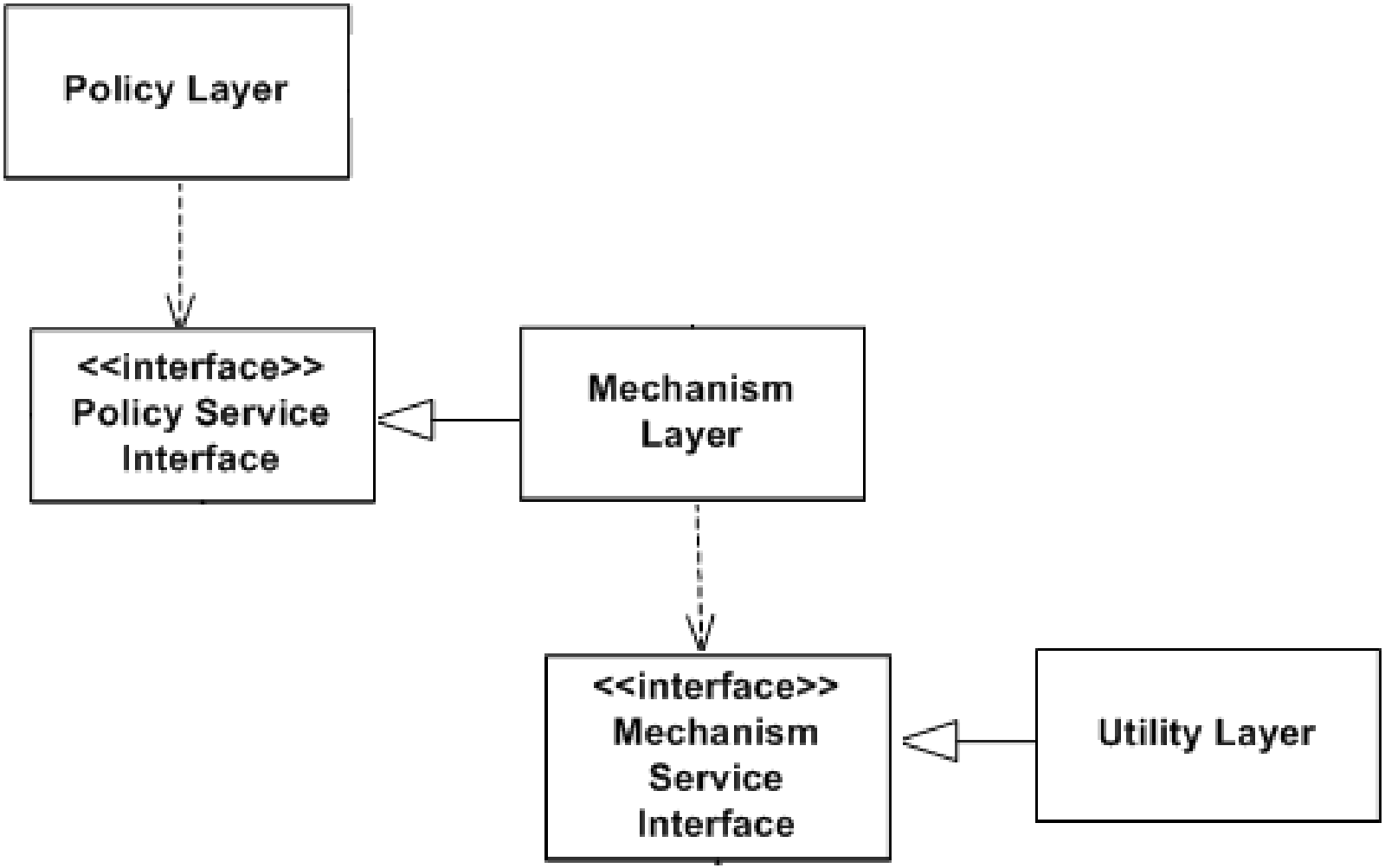
Some design patterns

- 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



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

- 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(); }
}
```

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(c), but overrides method x
- Extension2(c) inherits Decorator(c), but overrides methods y, z
- ConcreteDecorator inherits Extension1(Extension2(KeyComponent))
- Example: windowing system with configurable borders, scrollbars

Example: a simple animation

```
class AnimObject {
  protected:
    Position position;
    Model *model;
  public:
    virtual void update() { }
    virtual void draw() { // draw model }
    virtual void meet(AnimObject other) { }
}
```

Example: a simple animation

```
class InvisibleObject :  public AnimObject {
  public:
    void draw() override { }
}
class SolidObject :  public AnimObject {
  public:
    void meet(AnimObject other) {// handle collision}
}
class MovableObject :  public AnimObject {
  private:
    Direction direc;
    int speed;
  public:
    void update() { // update position }
}
```

Example: a simple animation

Challenge: how to get a solid movable block?

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 (now): 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!

Inheritance. . . awesome!

Inheritance. . . awesome?

Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

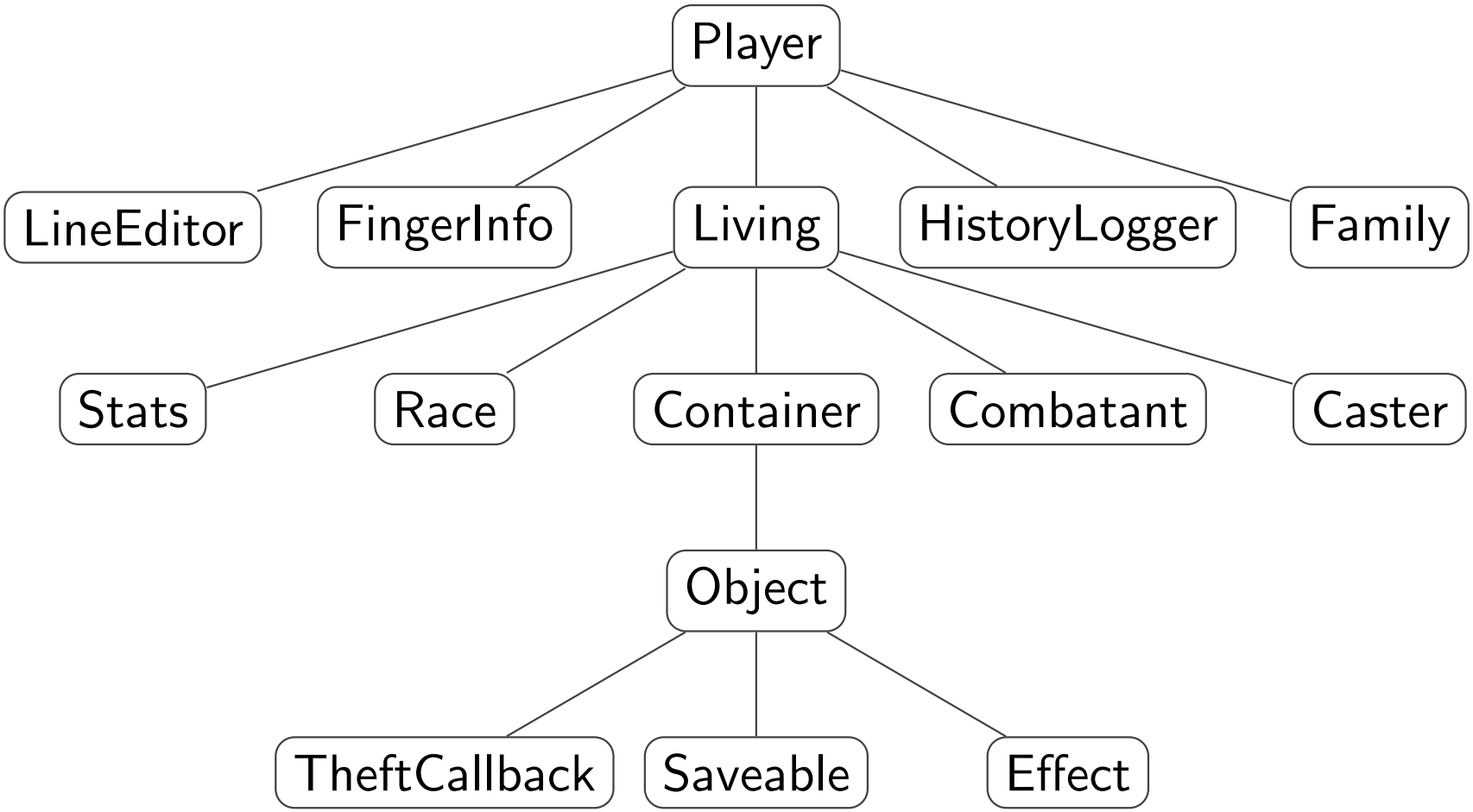
Inheritance. . . awesome?

- 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

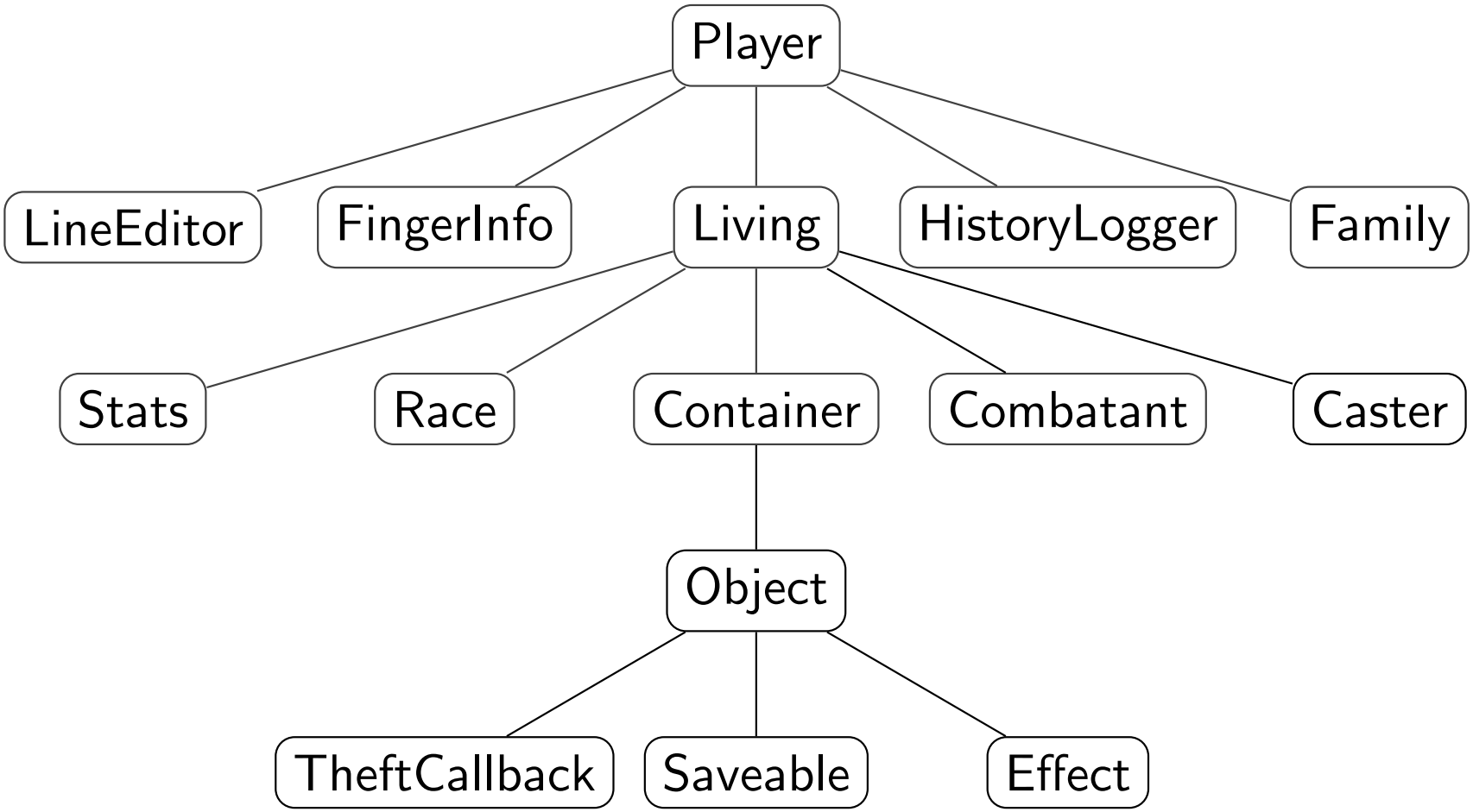


Inheritance. . . awesome?

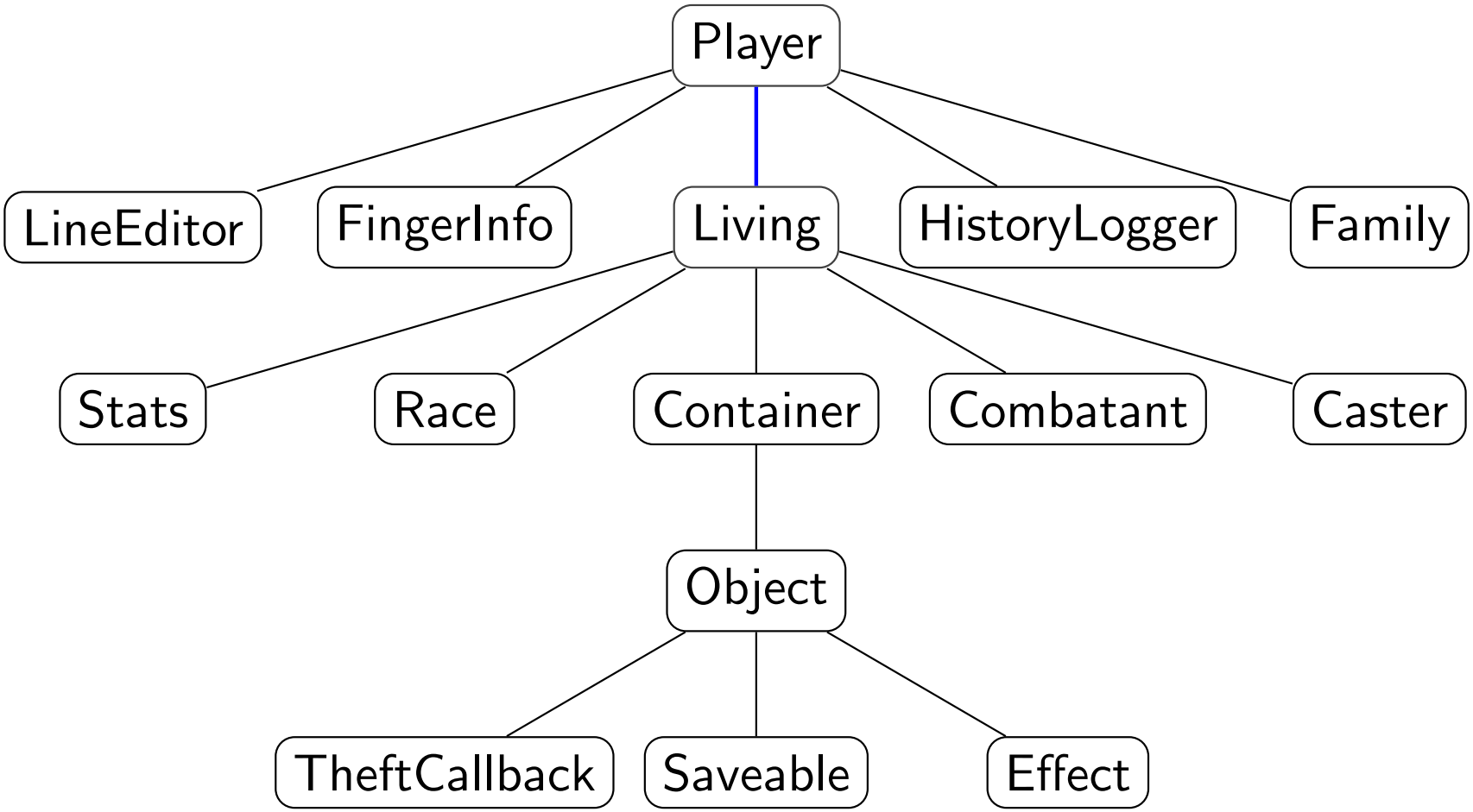
The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

– Joe Armstrong (creator of Erlang)

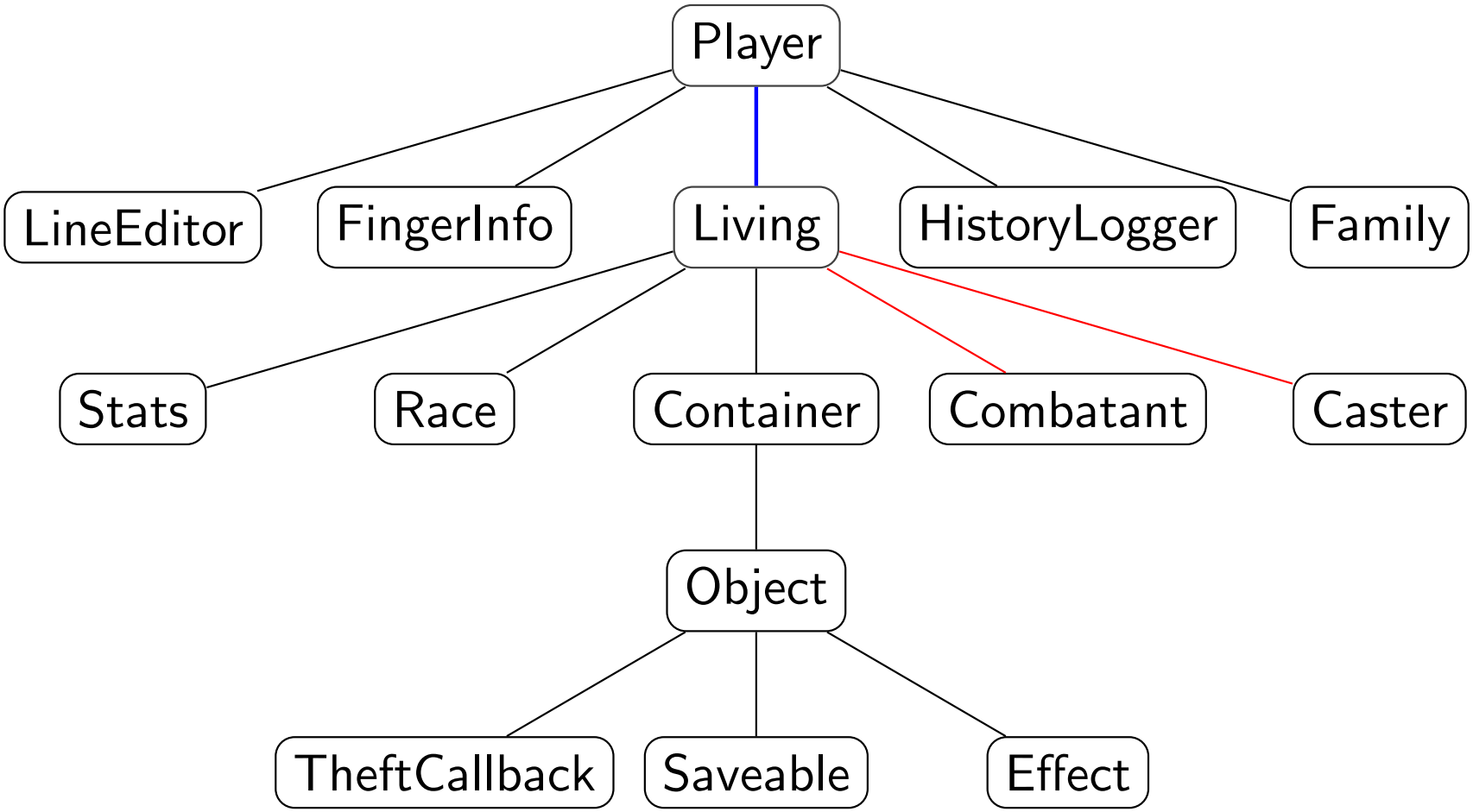
A deep hierarchy?



A deep hierarchy?



A deep hierarchy?



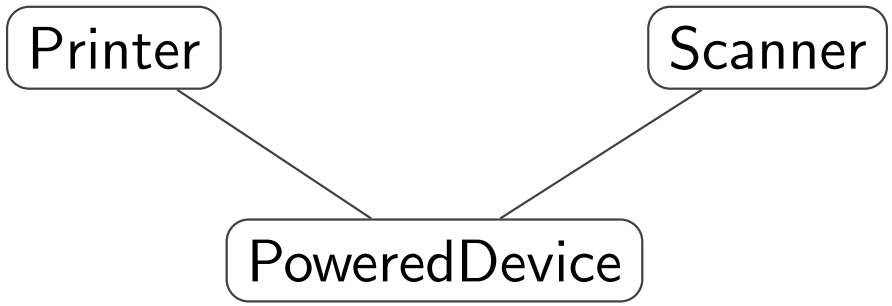
A deep hierarchy?

- 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

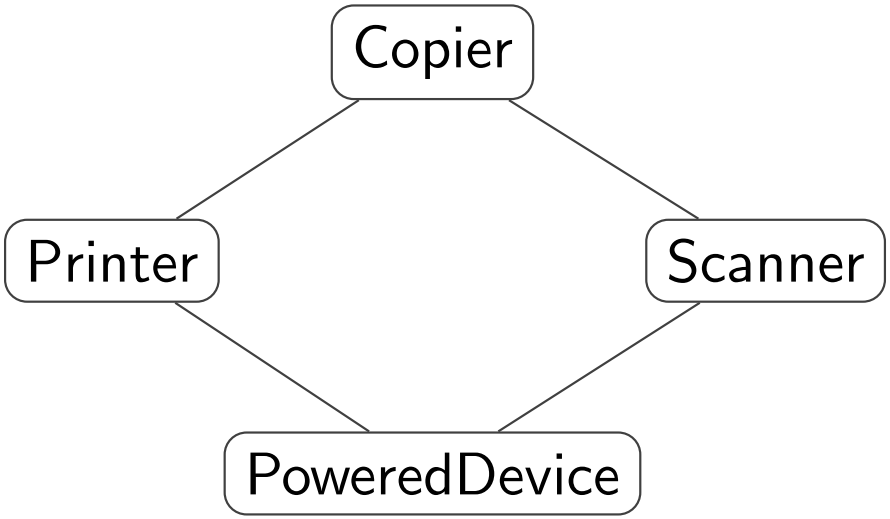
Inheritance. . . awesome?

- hierarchy depth
- the diamond problem
- fragile base classes

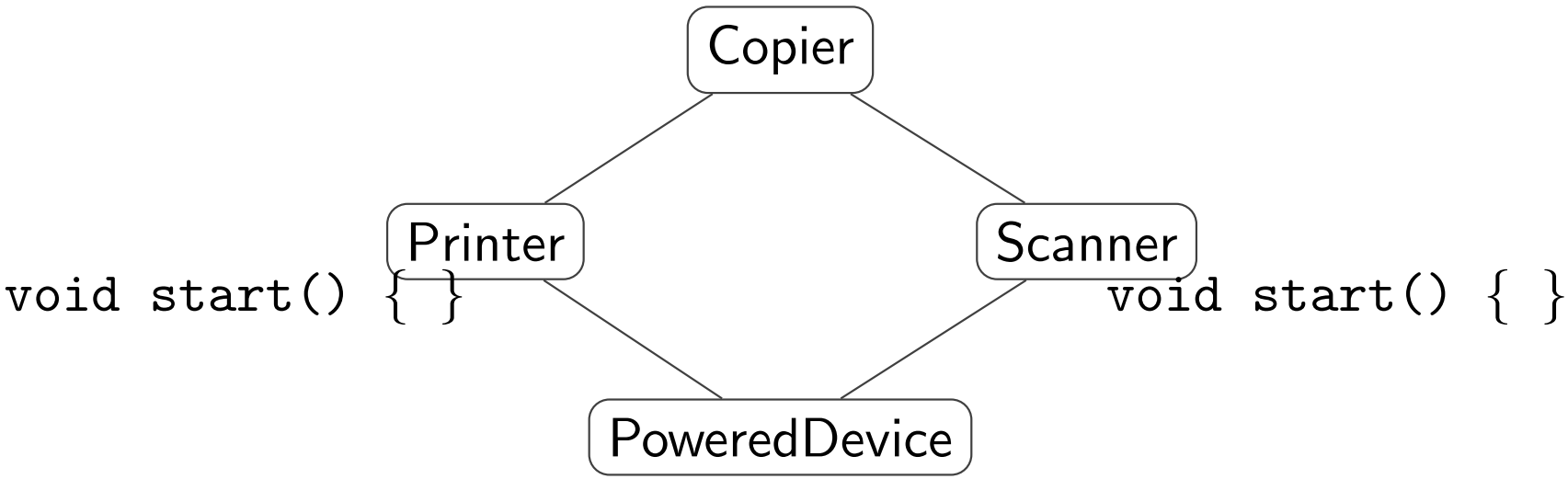
The diamond problem



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.

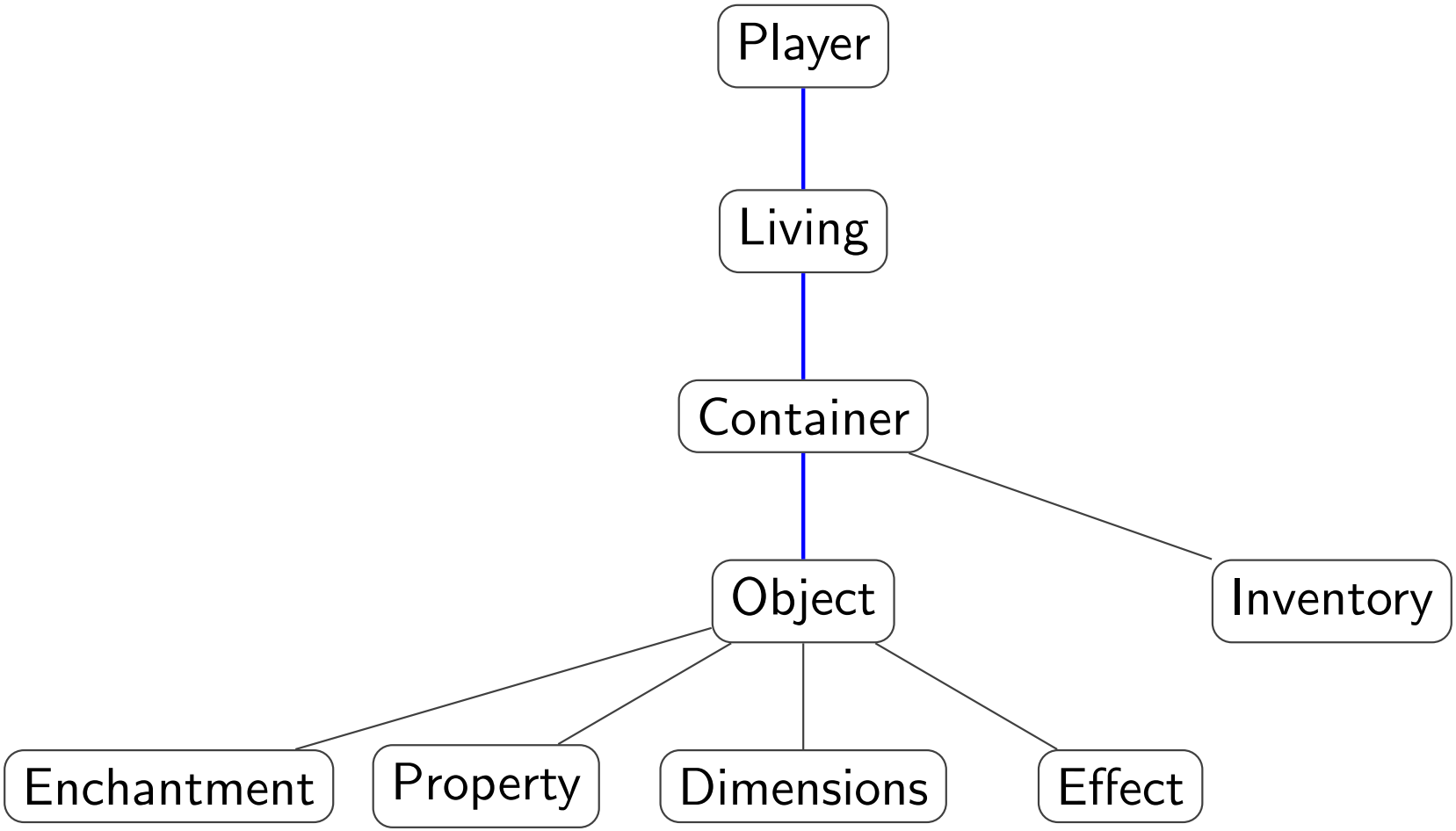
Composition over inheritance

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

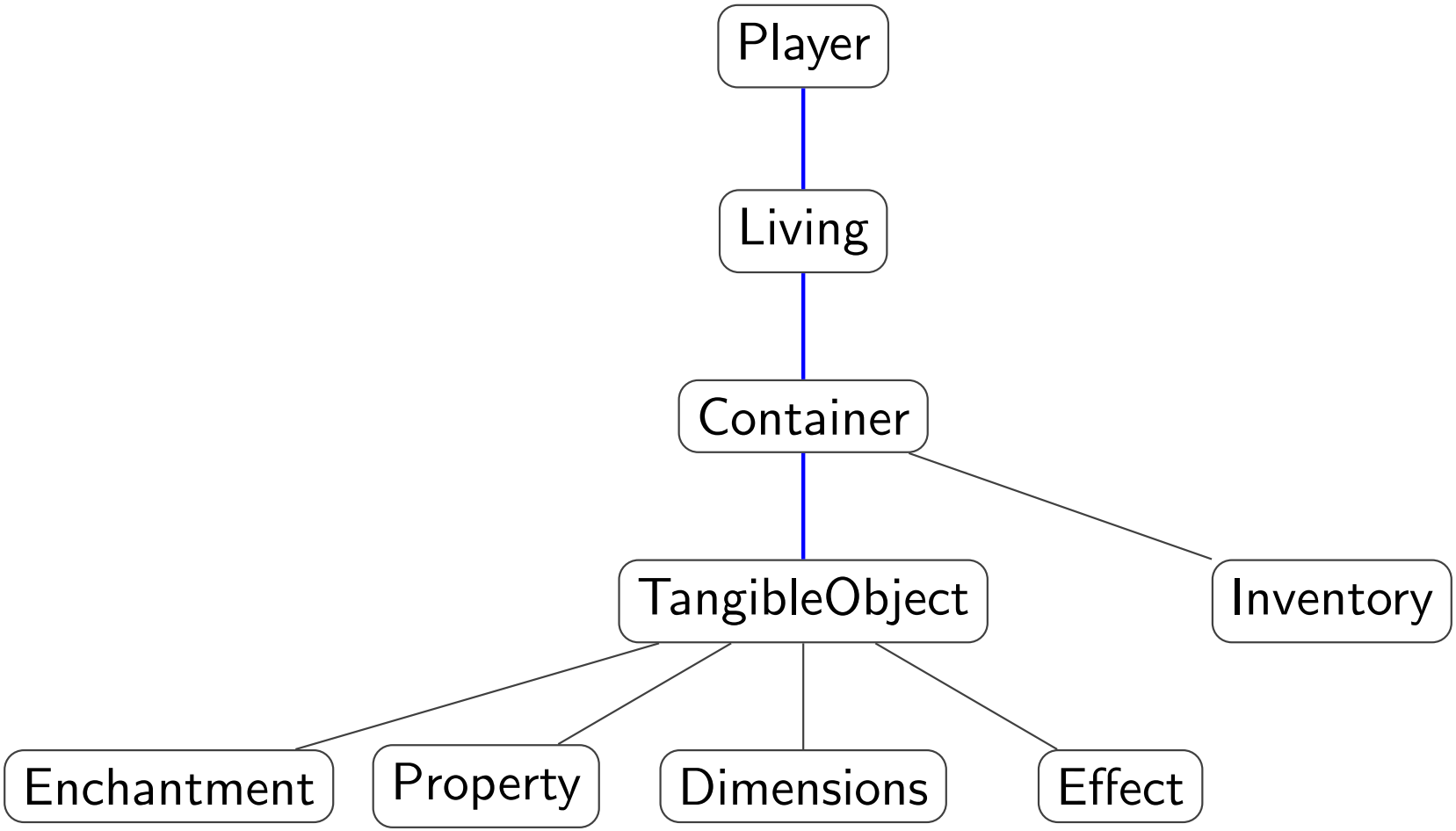
– Gang of Four

- 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.

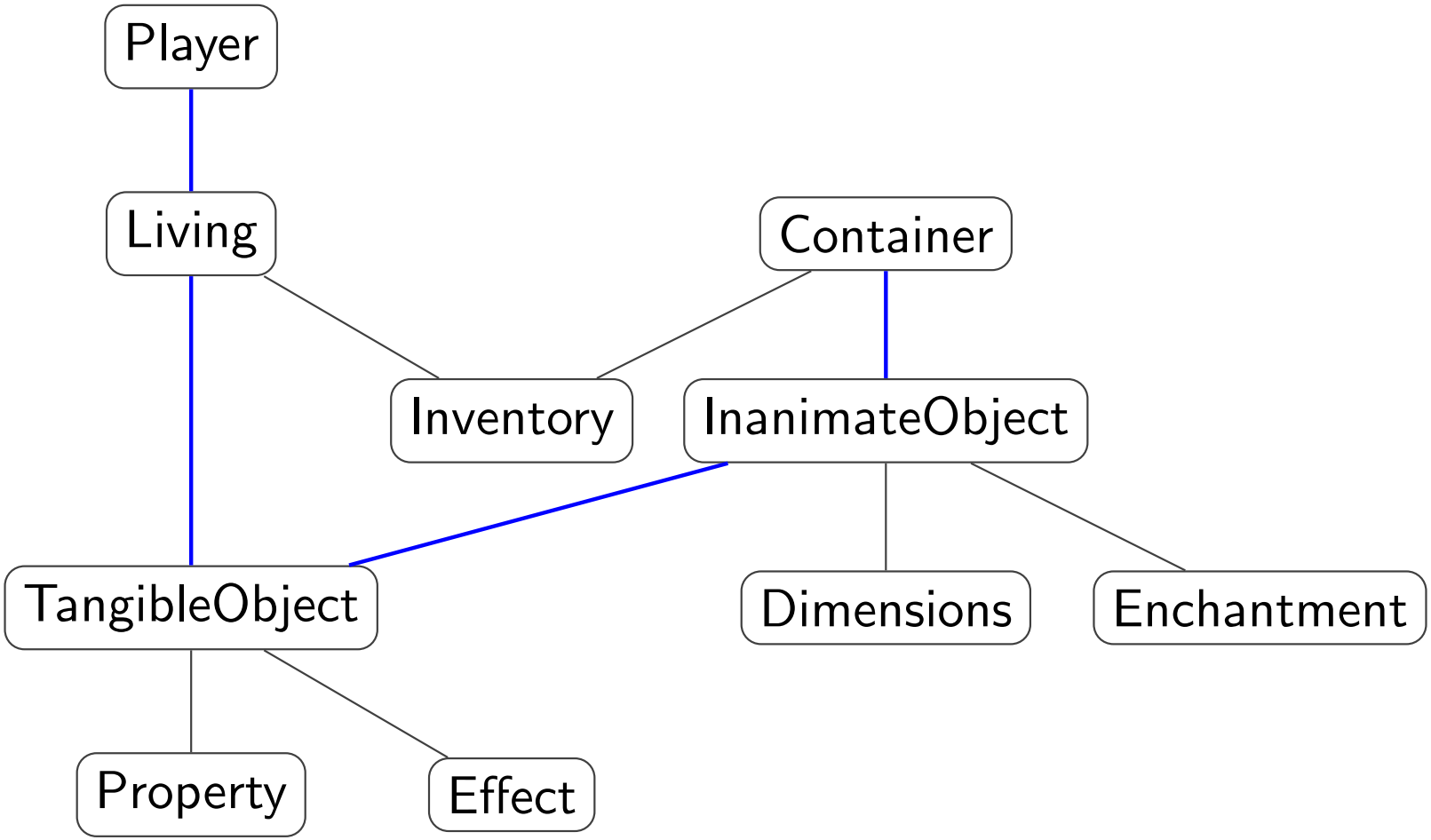
Exercise: the Discworld player object



Exercise: the Discworld player object



Exercise: the Discworld player object

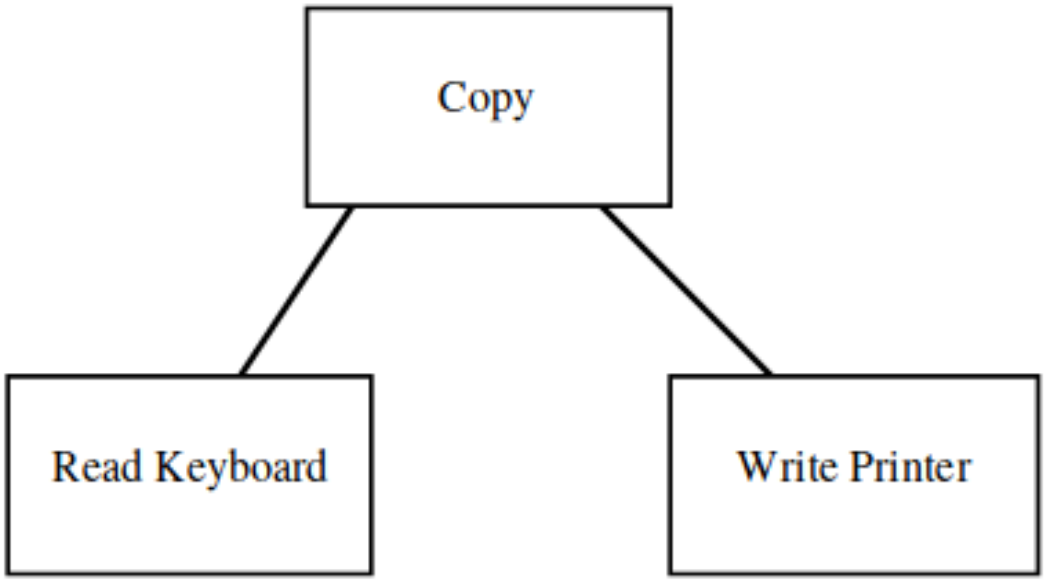


Dependencies

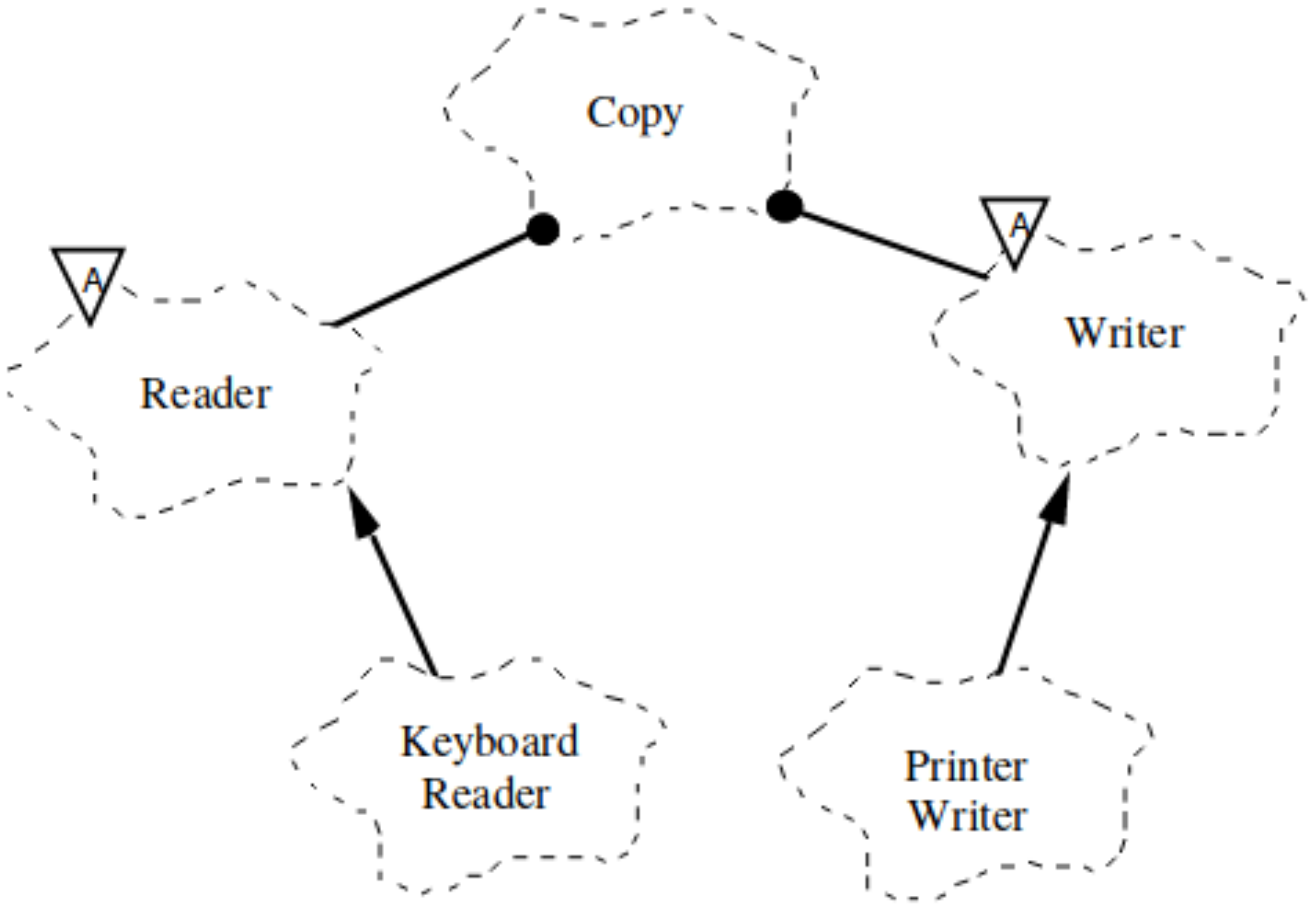
A design is rigid if it cannot be easily changed. Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When the extent of that cascade of change cannot be predicted by the designers or maintainers the impact of the change cannot be estimated. This makes the cost of the change impossible to estimate. Managers, faced with such unpredictability, become reluctant to authorize changes. Thus the design becomes rigid.

– Robert C. Martin

Dependencies



Dependencies



Dependencies

Why is the new design robust, maintainable, reusable?

$\Longrightarrow$ the targets of the dependencies are stable

- they depend on nothing at all

```
interface Writer { public void write(char c); }
interface Reader { public char read(); }
```

- they are (or might be) used by many other classes

Good dependency: a dependency on something very stable!

Class categories

- Sometimes a number of classes are interdependent.
- Little sense in measuring dependencies between them.
- Classes in a category:
  - are closed together against any force of change;
  - are reused together;
  - share some common function.

Some metrics

For a category C of classes:

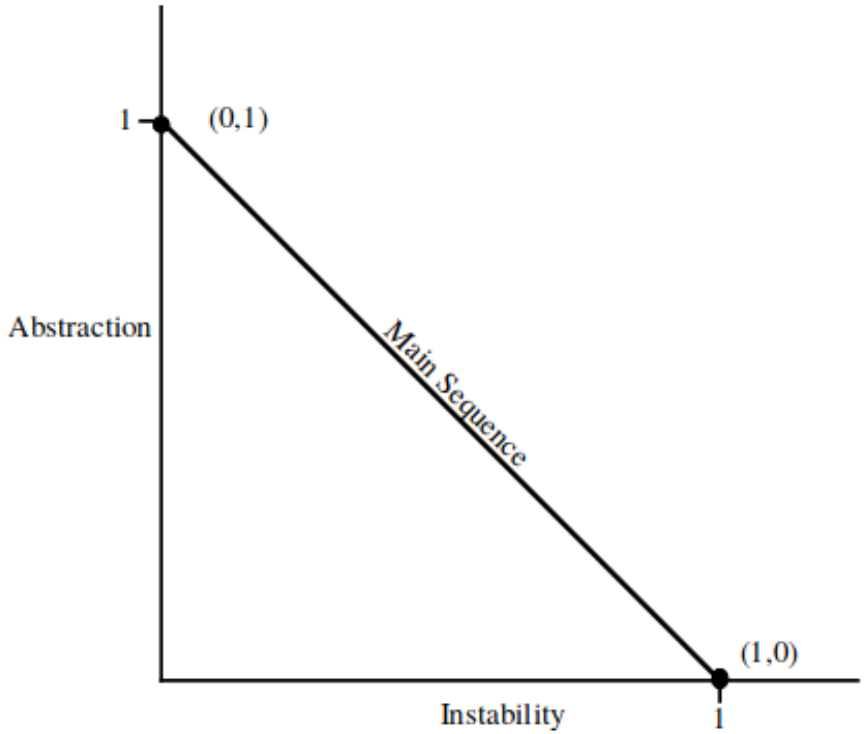
- Let Ca be the number of classes outside C that rely on C.
- Let Ce be the number of classes outside C that C relies on.
- The instability of C is

$$\frac{\text{Ce}}{\text{Ca} + \text{Ce}}$$

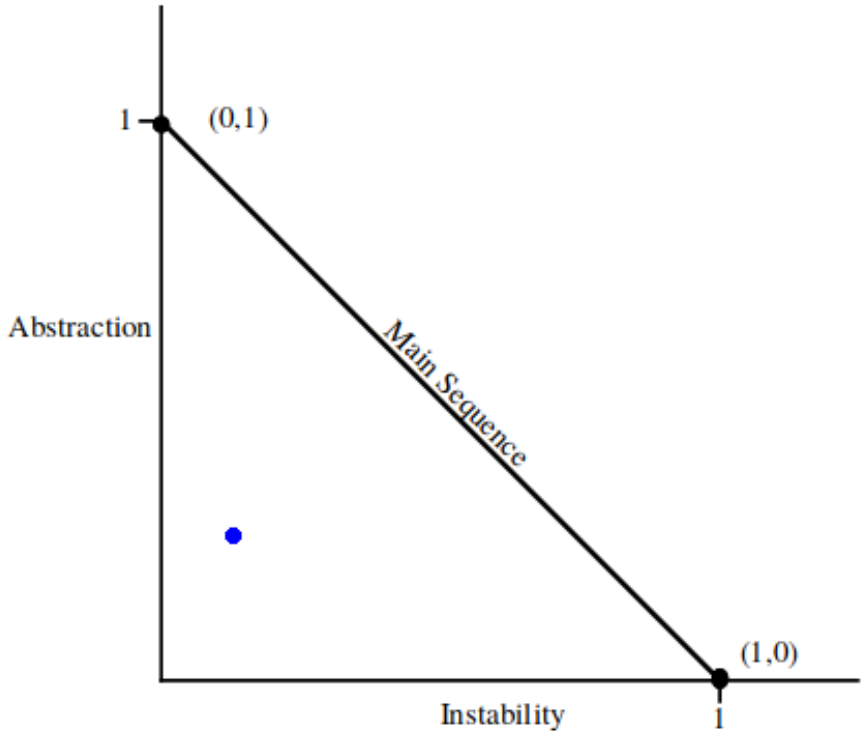
- But: not all classes need to be stable!
- The abstractness of C is

$$\frac{\#\text{abstract classes in C}}{\#\text{total classes in C}}$$

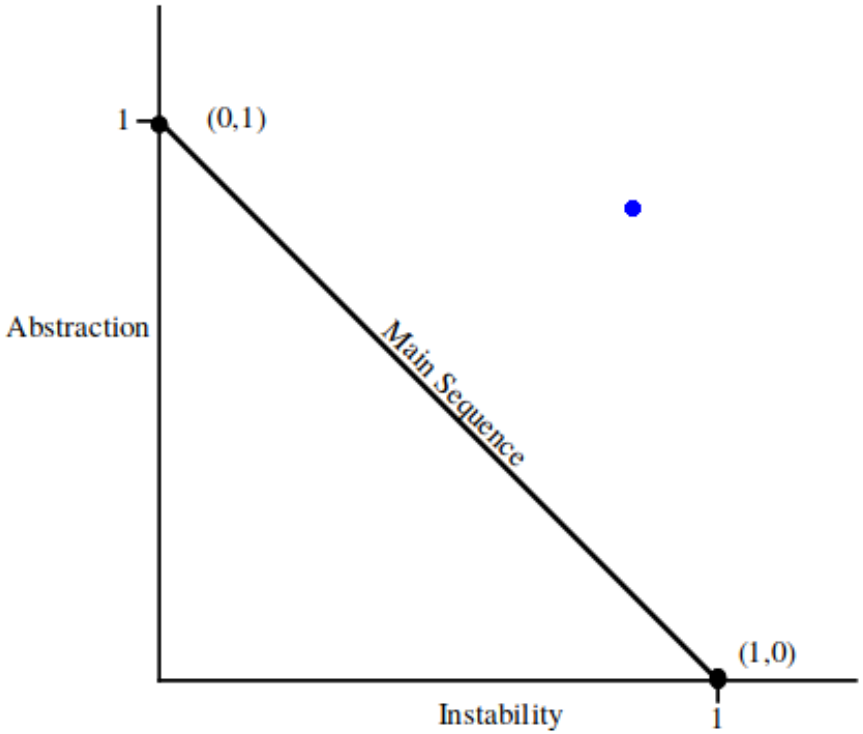
Some metrics



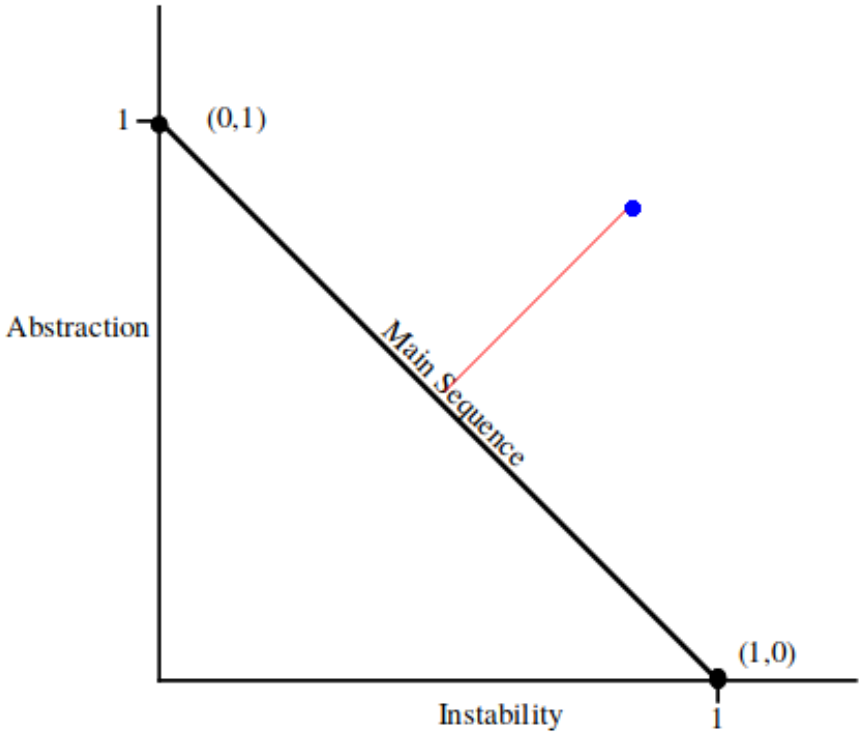
Some metrics



Some metrics



Some metrics



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.

Code smells and anti-patterns

Design / code smells

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

Code smells – practical metrics in BetterCodeHub

- Code
  - write small units of code ($\leq$ 15 lines per method)
  - write simple units of code ($\leq$ 4 branch points per method)
  - write code once ($\leq$ 6 lines of copied code)
  - keep unit interfaces small ($\leq$ 4 parameters per method)
- Architecture
  - separate concerns in modules ($\leq$ 400 lines per module)
  - couple components loosely (typically $\leq$ 10 incoming calls, no cyclical dependencies)
  - keep architecture components balanced (6-12 top-level components)
  - keep your codebase small ($\leq$ 200 000 lines of code)
- Way of Working
  - write automated tests that cover all code
  - leave your code clean (remove inaccessible code)

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`