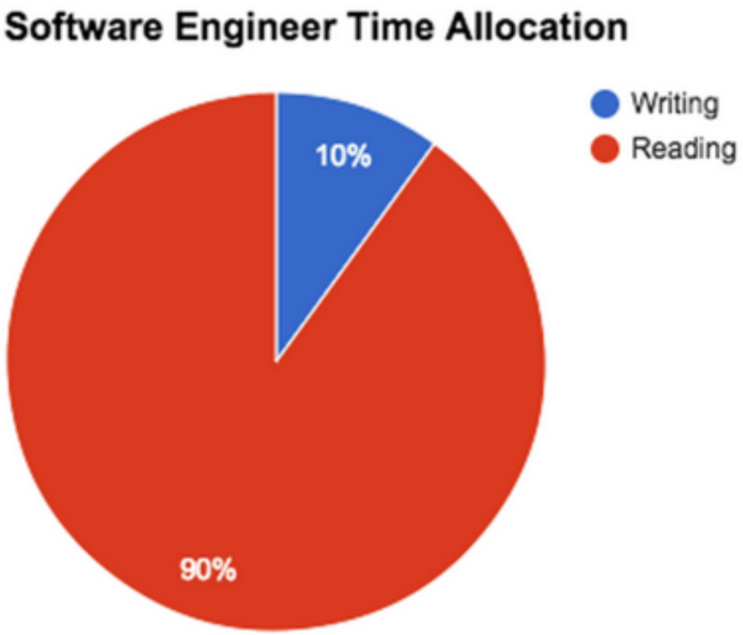
Recitation 13

Software Engineering Practices and Introduction to Design Patterns

Software Development is *chaotic*



During that 90% time:

Good engineers think, research, read the codebase, and recognize **design patterns**

How to be a ~~good~~ great engineer

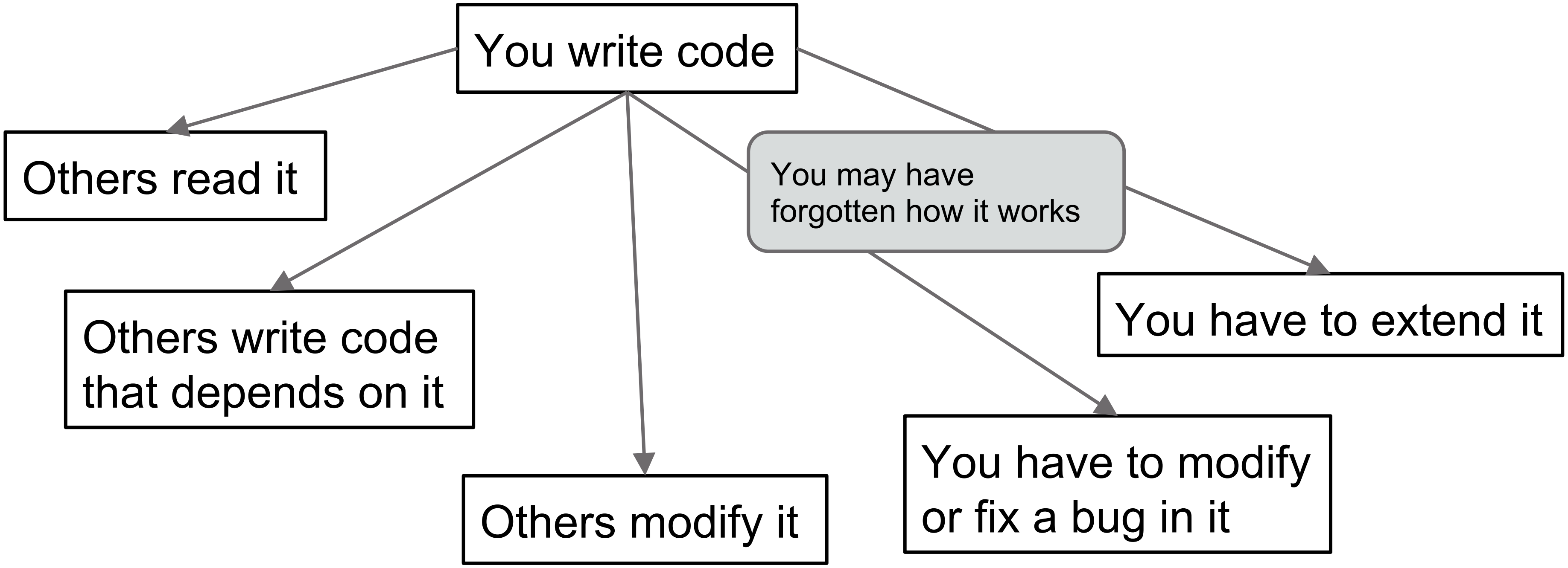
1. Focus on code clarity
    a. see style guidelines on course webpage

2. Adopt practices to help avoid bugs

3. Utilize design patterns

Coding Strategies

The future of your code...



You write code

Others read it

Others write code that depends on it

Others modify it

You may have forgotten how it works

You have to extend it

You have to modify or fix a bug in it

Method design

Good methods have a clear, crisp purpose

Good methods usually:

- are short
- are reusable
- have few parameters
- have few side-effects

Consequences of good methods:

- can test them independently
- make code more readable
- reduce likelihood of typos
- reduce redundancies

Wrapper methods

Wrapper methods don’t add much new functionality but increase readability and reduce typos

Example: ArrayList

```
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

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

Use abstractions

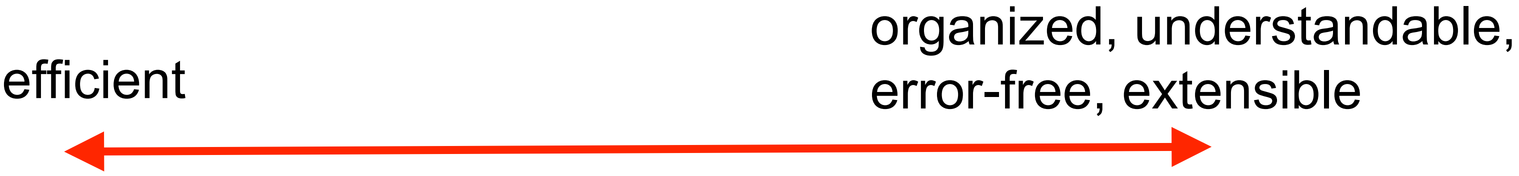
Any time you are unsure whether some behavior may change, **abstract** it (behind an **interface**, **method**, etc)

```
class Person {
        public boolean marry(Person p) {
        if (isRelated(p)) return false;
        ...
    }
    public boolean getRelatives(Person[] people) {
        for (Person p : people) {
                if (isRelated(p)) ...
        }
    }
}
```

if we change our definition of “is related”, the rest of the program remains intact

Clarity vs Efficiency

There is often a trade off between the two.
We want to find the right balance.



efficient

organized, understandable,
error-free, extensible

Examples:

- linear search vs binary search
- using bit manipulation vs regular arithmetic
- working directly with char arrays vs Strings
- caching objects locally for later use vs throwing them away

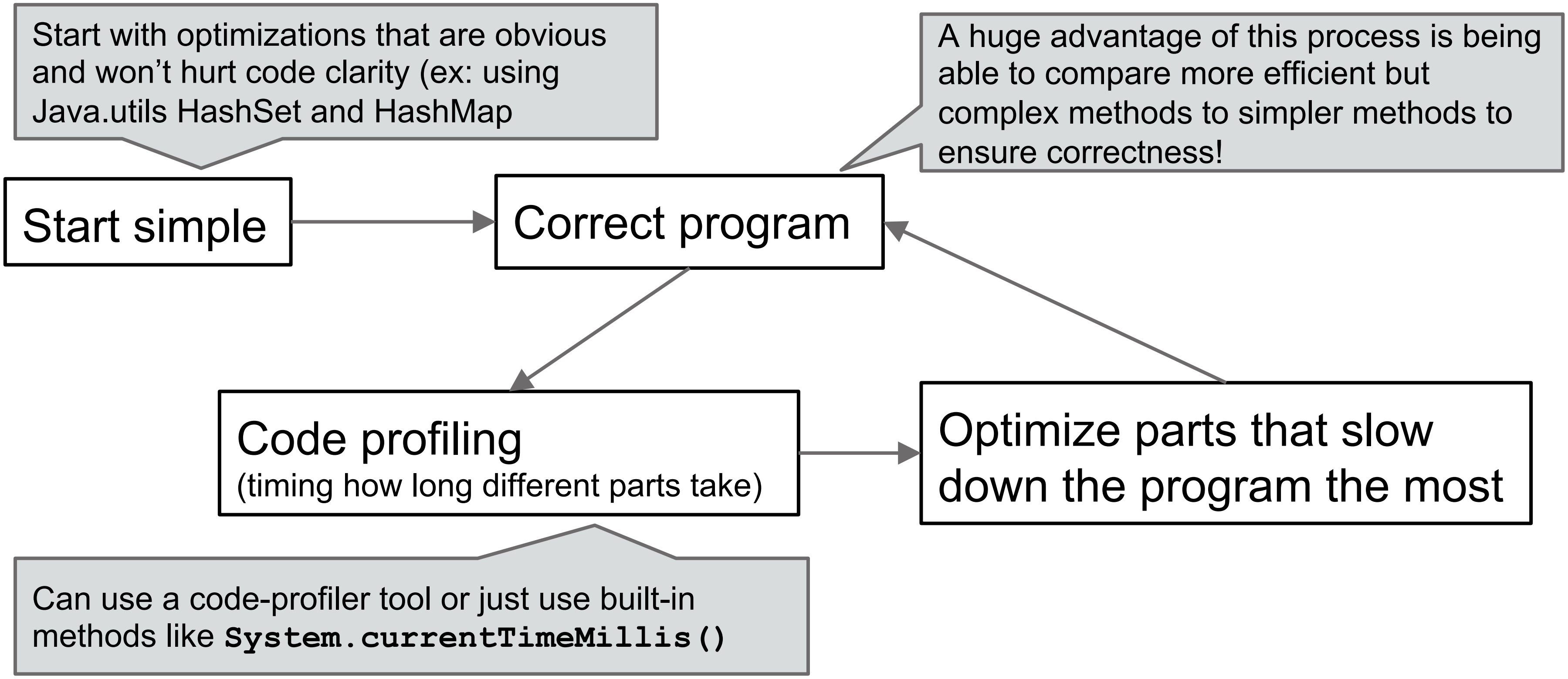
Premature optimization

Premature Optimization: trying to make code more efficient from the start. This is usually **bad.**

Why?

- You don't know in advance how slow different parts of the program will be; you may be wasting your efforts optimizing parts that are pretty good already
- You will often sacrifice clarity for efficiency
- It is almost always easier to take well-organized code and optimize it later than it is to take poorly-organized code and clarify it

How to optimize



Start with optimizations that are obvious and won’t hurt code clarity (ex: using Java.utils HashSet and HashMap

Start simple

Correct program

A huge advantage of this process is being able to compare more efficient but complex methods to simpler methods to ensure correctness!

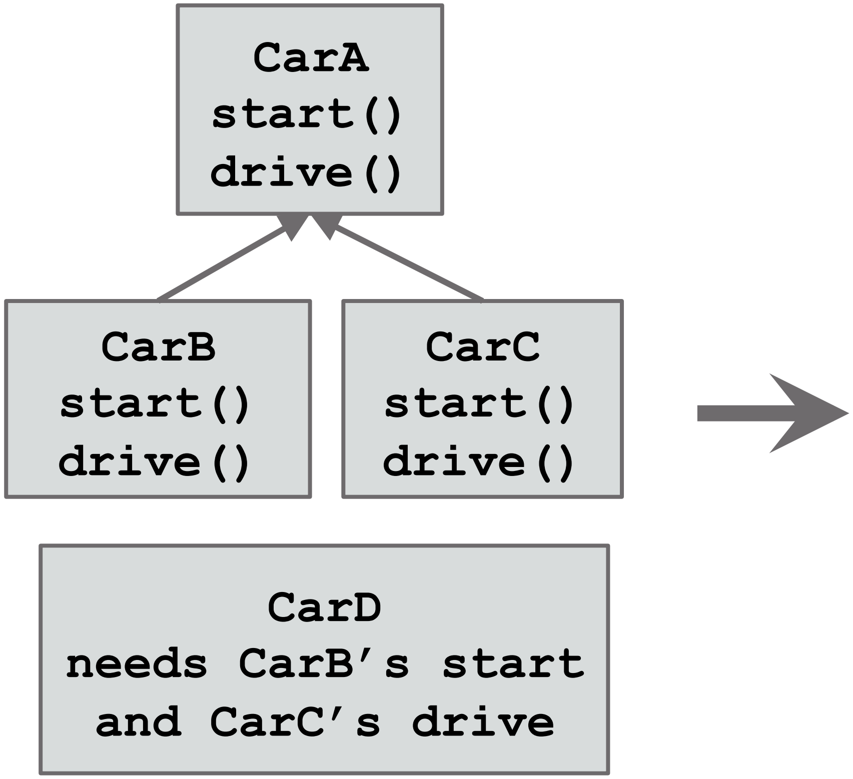
Code profiling

(timing how long different parts take)

Optimize parts that slow down the program the most

Can use a code-profiler tool or just use built-in methods like `System.currentTimeMillis()`

Composition vs Inheritance



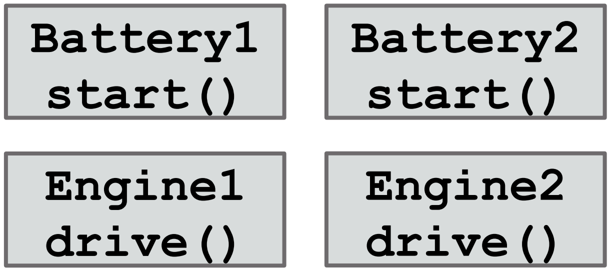
CarA
start()
drive()

CarB
start()
drive()

CarC
start()
drive()

CarD
needs CarB's start
and CarC's drive

If you find your class hierarchy needing to inherit from multiple classes or you need more flexible objects, try composition:



Battery1
start()

Battery2
start()

Engine1
drive()

Engine2
drive()

Cars have batteries and engines. They can pick which type, and even change on the fly

How to avoid bugs!

Never use copy/paste!

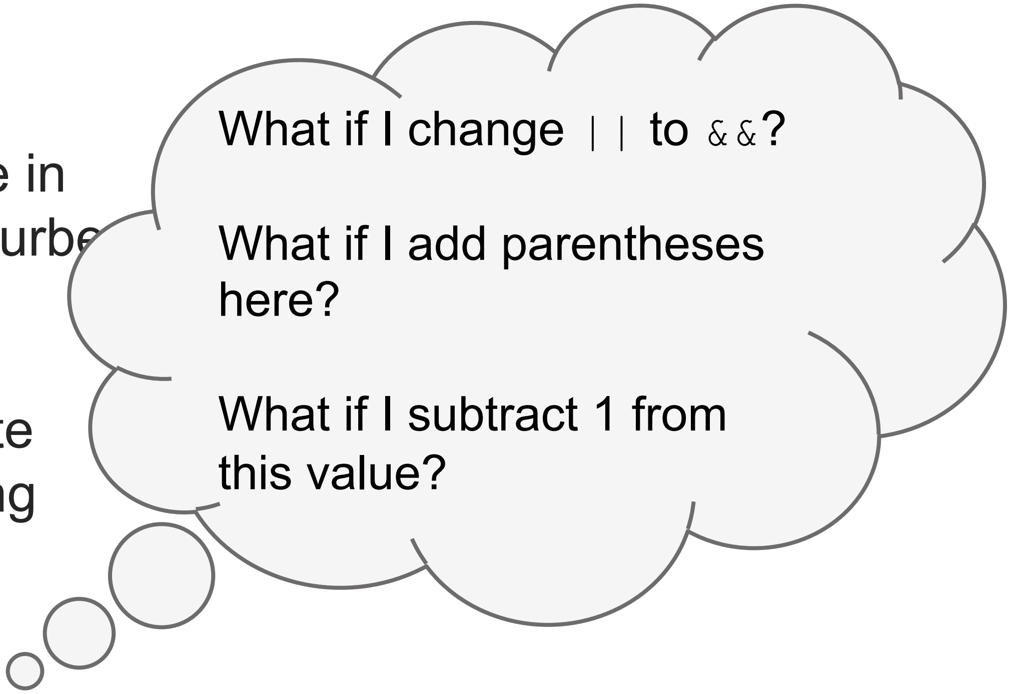
- You may introduce needless bugs unknowingly
- You should probably refactor your code to a new helper method

What NOT to do: Shotgun debugging

Shotgun debugging:

- a process of making relatively undirected changes to software in the hope that a bug will be perturbe out of existence.

- has a relatively low success rate and can be very time consuming



What if I change `||` to `&&`?

What if I add parentheses here?

What if I subtract 1 from this value?

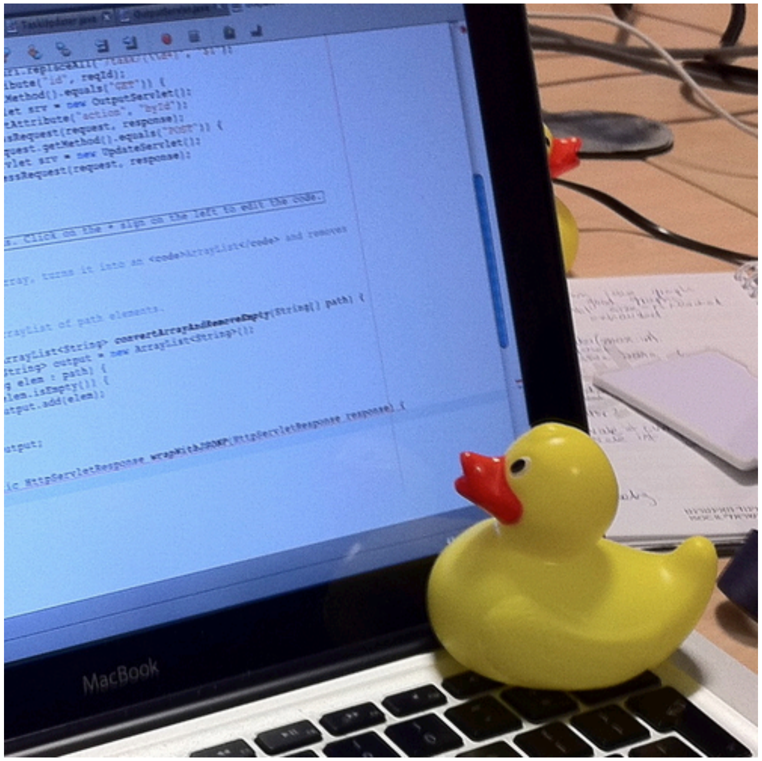
http://en.wikipedia.org/wiki/Shotgun_debugging

Rubber duck debugging

- The process of walking a rubber duck through your code, explaining out loud

- Try to do this before even running your code!

- Rely more on your reasoning than the test output

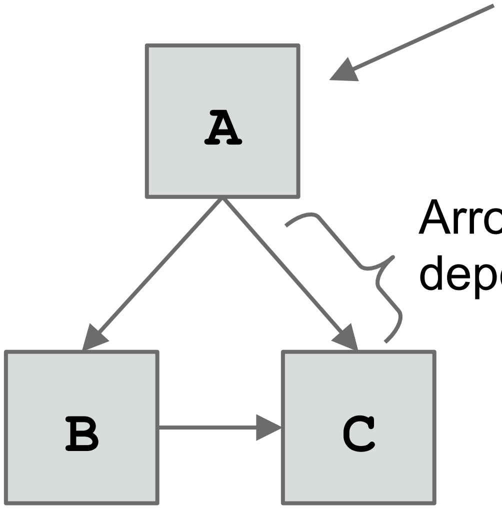


http://en.wikipedia.org/wiki/Rubber_duck_debugging

Class diagrams

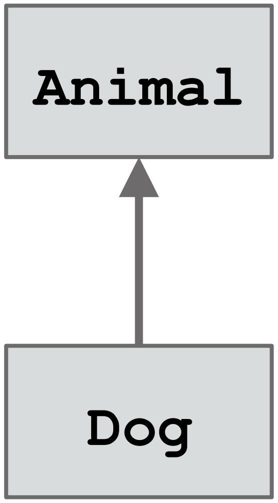
Dependency:
The dependent class relies on the independent class.

Example:
Class A has a parameter, local variable, or field that is of type C. Class C is isolated from A and B.



class A

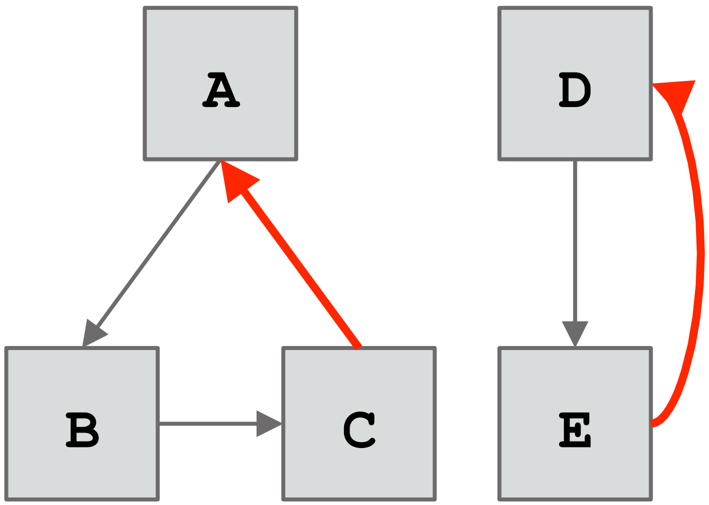
Arrow shows dependency on class C



Inheritance arrow

Class diagrams help us *visualize* the design of a system.

Generally, avoid cyclic dependencies



Cyclic dependencies:

1. Make your code harder to read and maintain
2. Make it difficult to isolate portions of your codebase to find bugs and test
3. Make it hard to ensure all objects are updated and valid

Ideally, your module structure should be a **directed acyclic graph**.

Utilize Design Patterns

Design Patterns

- solutions to general code design problems
- A description, a template, not code

Why?

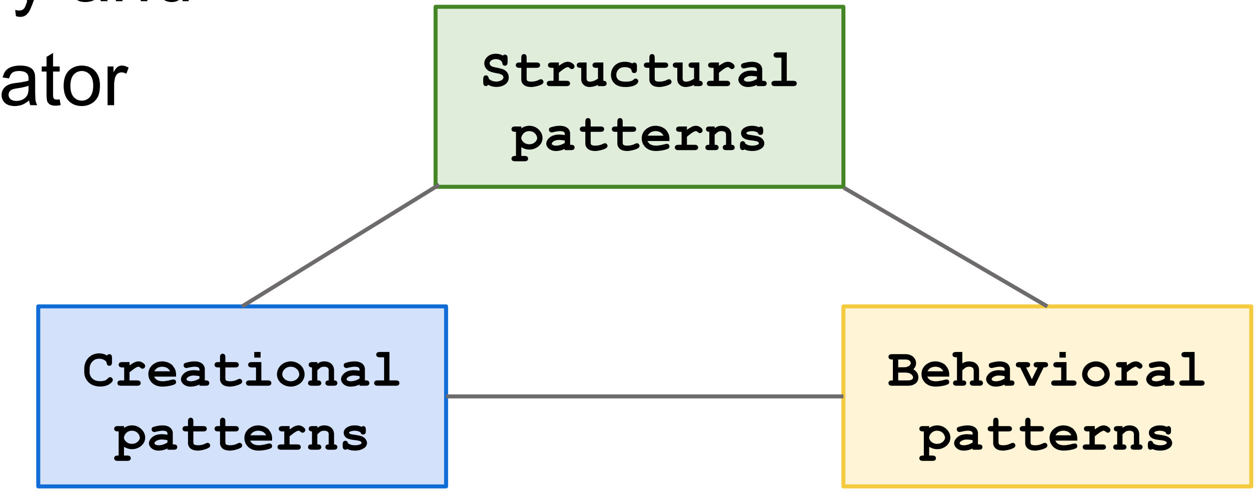
1. Common terminology for developers
2. Best Practices that stand the test of time
3. Makes code reliable and effective

Design Patterns

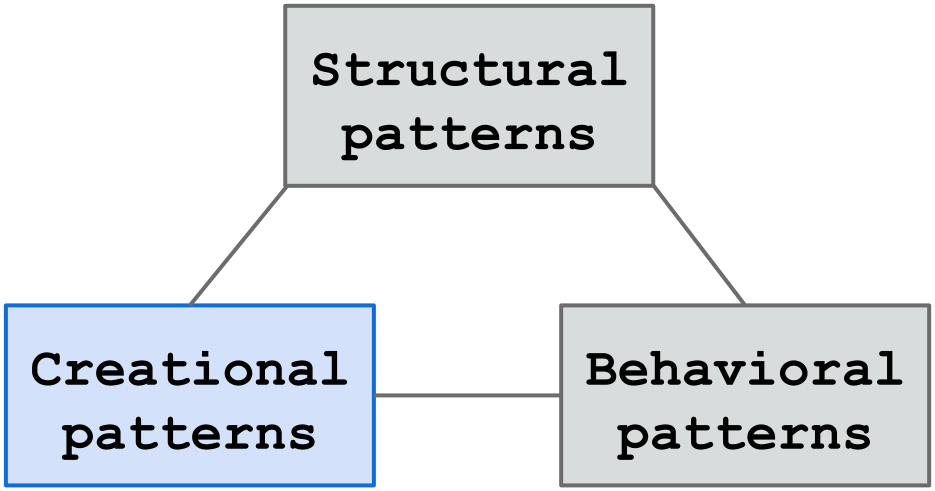
We will talk about two common patterns:

Factory and

Decorator



Design Patterns: Creational

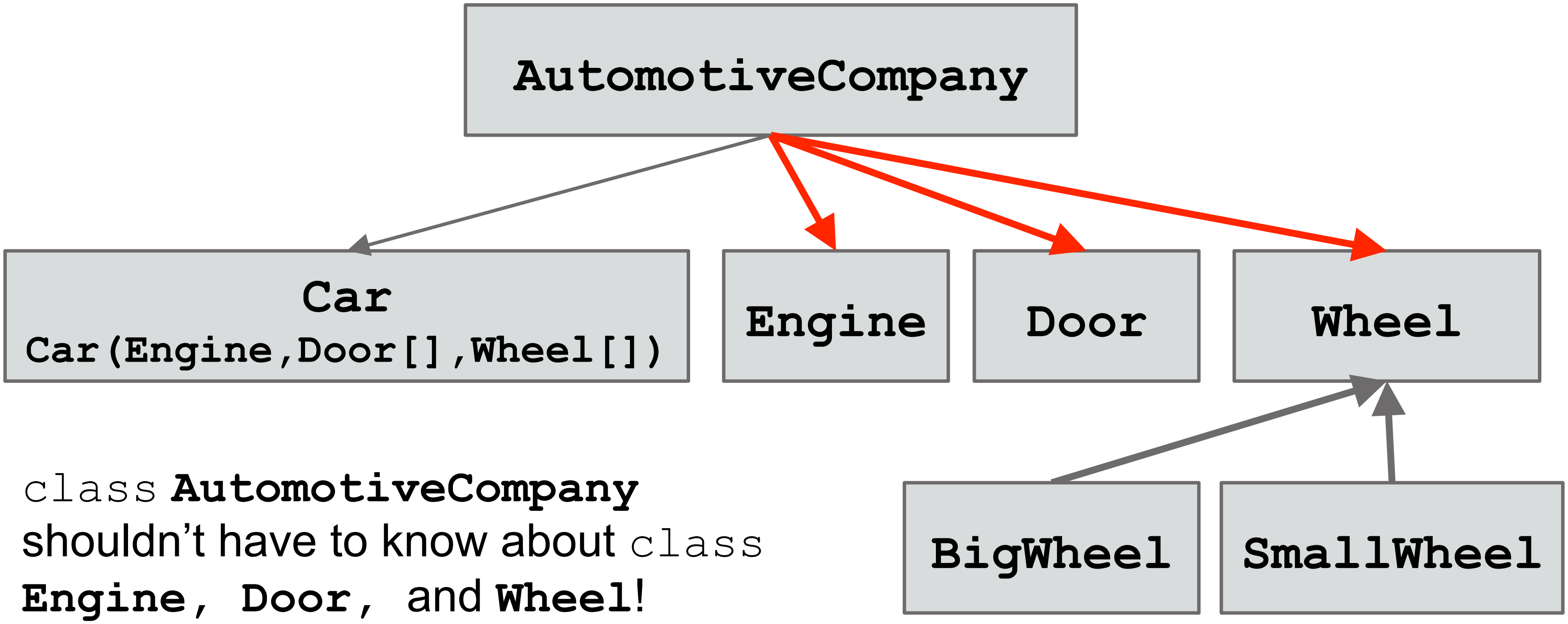


Creational patterns: Provide ways to create objects **without** using the `new` keyword.

Popular example:

- Factory method

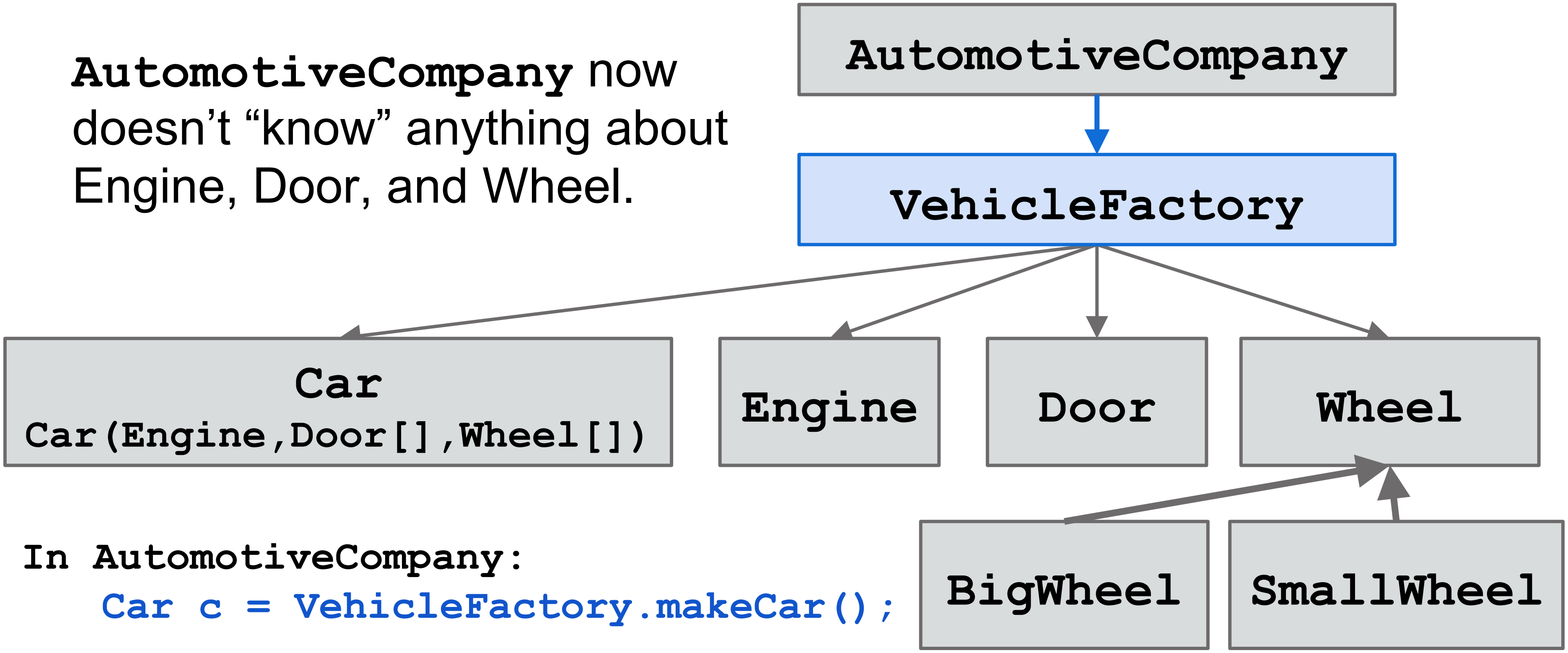
Problem



`class` **`AutomotiveCompany`** shouldn't have to know about `class` **`Engine`**, **`Door`**, and **`Wheel`**!

Fix: Factory method Pattern

AutomotiveCompany now doesn't "know" anything about Engine, Door, and Wheel.



In AutomotiveCompany:

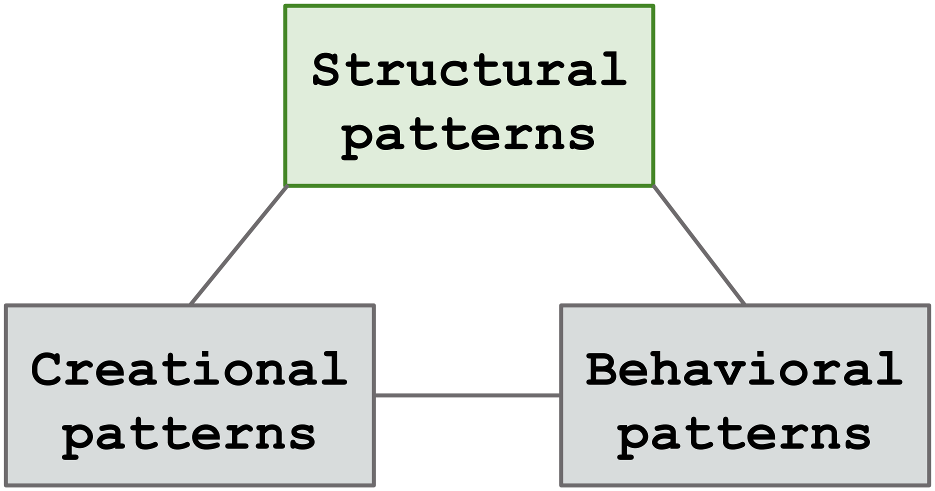
```
    Car c = VehicleFactory.makeCar();
```

Benefits of Factory method

1. More encapsulation!
   a. `AutomotiveCompany` cannot “see” how objects are constructed
   b. `AutomotiveCompany` is more readable with fewer dependencies

2. Allows for an Object Pool!
   a. Don’t necessarily need to reallocate objects! Can reuse old ones.

Design Patterns: Structural



Structural patterns:

Patterns that identify and realize relationships between entities

Popular example:

- Decorator

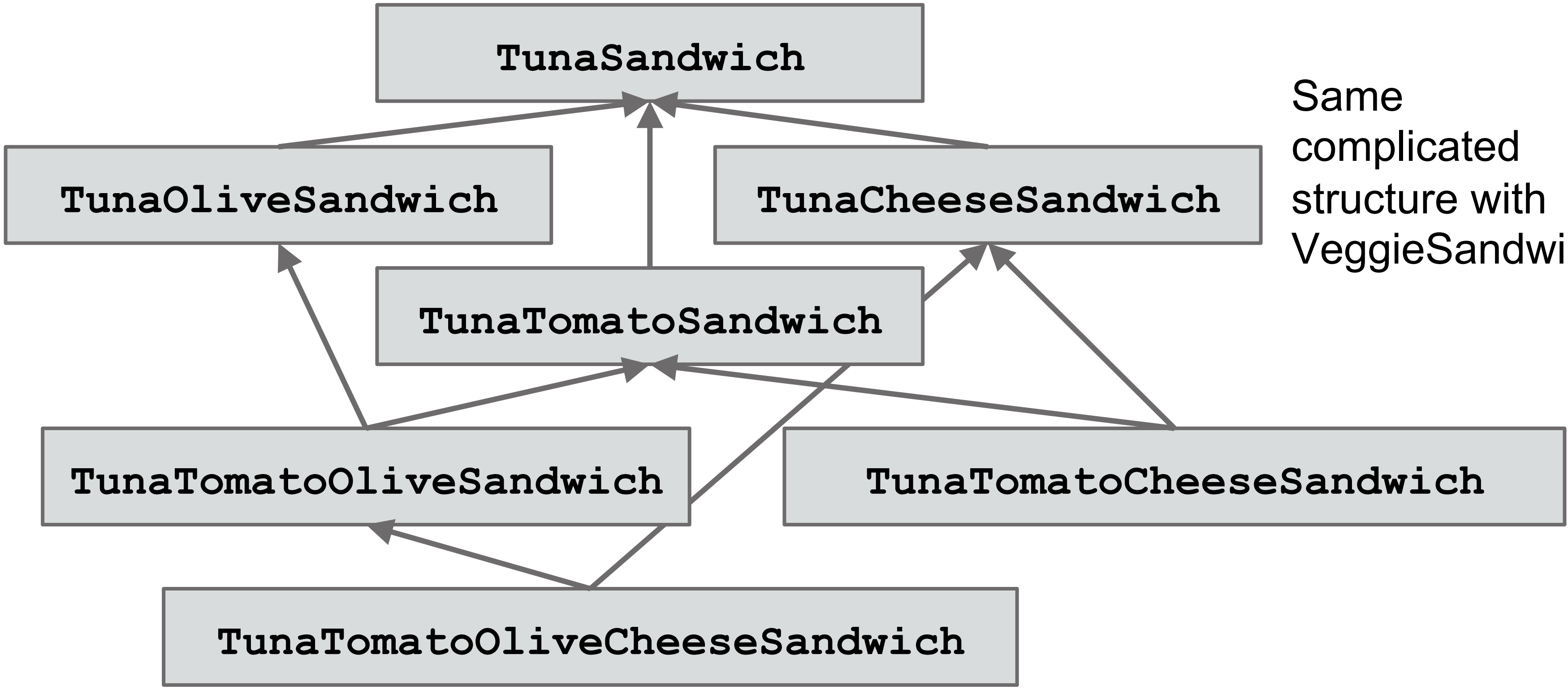
Scenario: Sandwich Shop

Menu

Veggie Sandwich	…	$4
Tuna Sandwich	…	$5
Add Cheese	…	$1
Add Olives	…	$2
Add Tomato	…	$3

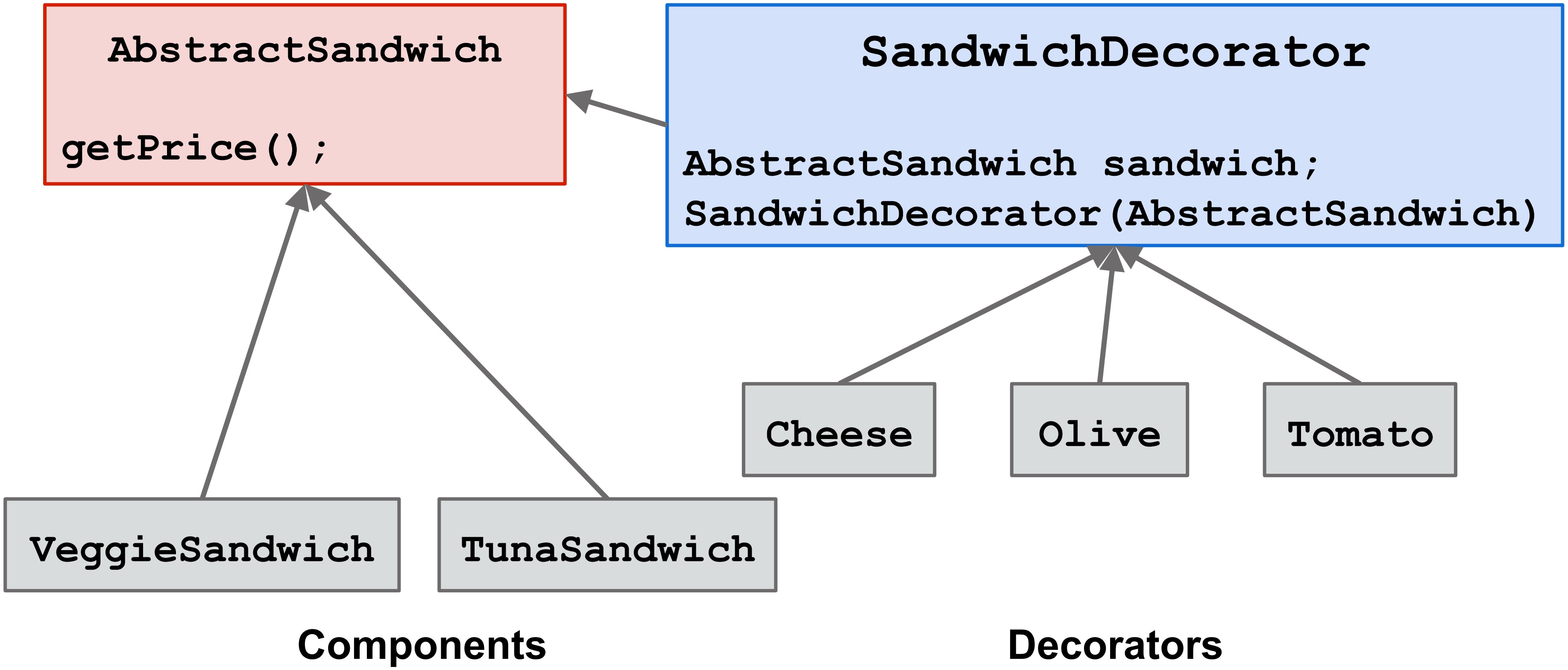
We need to return the correct price.
How do we represent **all** combinations of toppings at runtime/dynamically?

Problem: Sandwich Shop

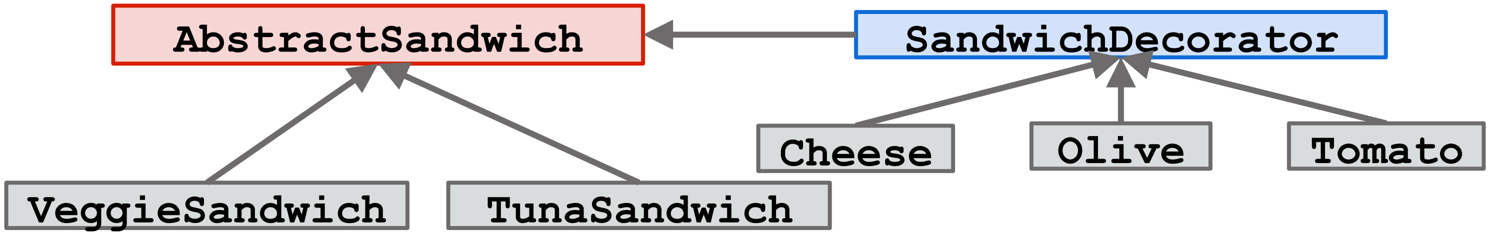


Same complicated structure with VeggieSandwich!

Fix: Decorator Pattern



Fix: Decorator Pattern



Components

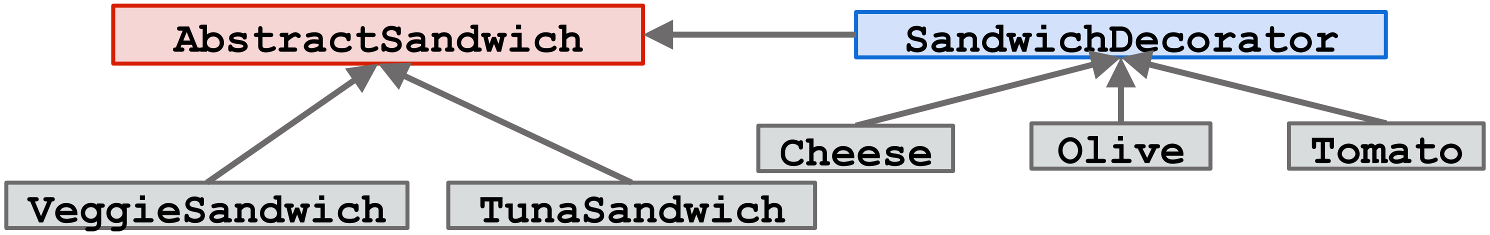
```
class VeggieSandwich {
    double getPrice(){
        return 4;
    }
}
```

Decorators

```
class Tomato {
    double getPrice(){
        return sandwich.getPrice()
             + 3;
    }
}
```

Recursive nature of the decorators

Fix: Decorator Pattern

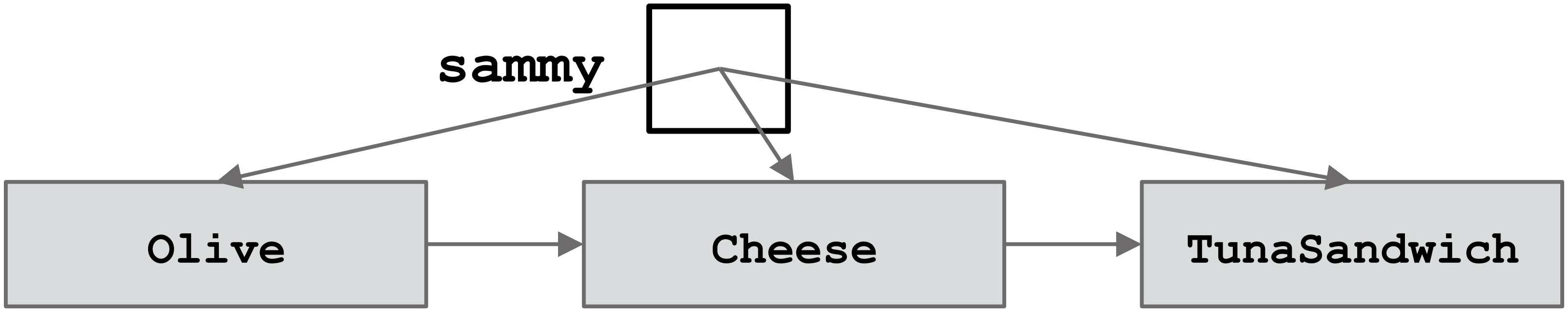


Making a Tuna sandwich with cheese and olives:

```
AbstractSandwich sammy = new TunaSandwich();
sammy = new Cheese(sammy);
sammy = new Olive(sammy);
System.out.println(sammy.getPrice());
```

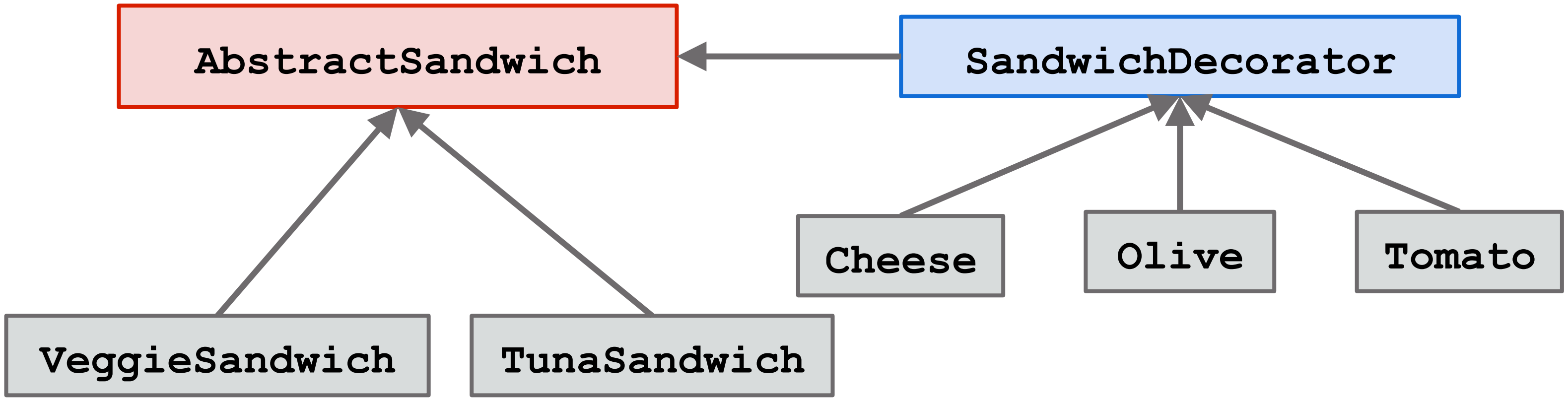
Fix: Decorator Pattern

```
AbstractSandwich sammy = new TunaSandwich();
sammy = new Cheese(sammy);
sammy = new Olive(sammy);
System.out.println(sammy.getPrice());
```



Essentially a linked list of objects of type AbstractSandwich, where a component (eg TunaSandwich) is the base case. Olive and Cheese were the decorators.

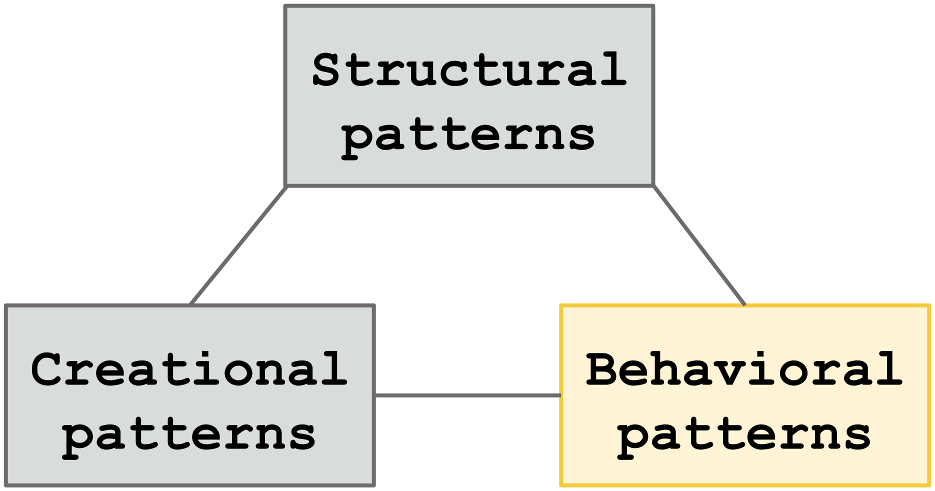
Benefits of the Decorator Pattern



Components **Decorators**

1. Much simpler design!
2. Useful when you would like to add features to a component at runtime

Design Patterns: Behavioral



Behavioral patterns:

Patterns that dictate how objects communicate and share data

Popular examples:

- Visitor
- Observer

Design Patterns References:

There are many more patterns to learn!

Seminal Book:

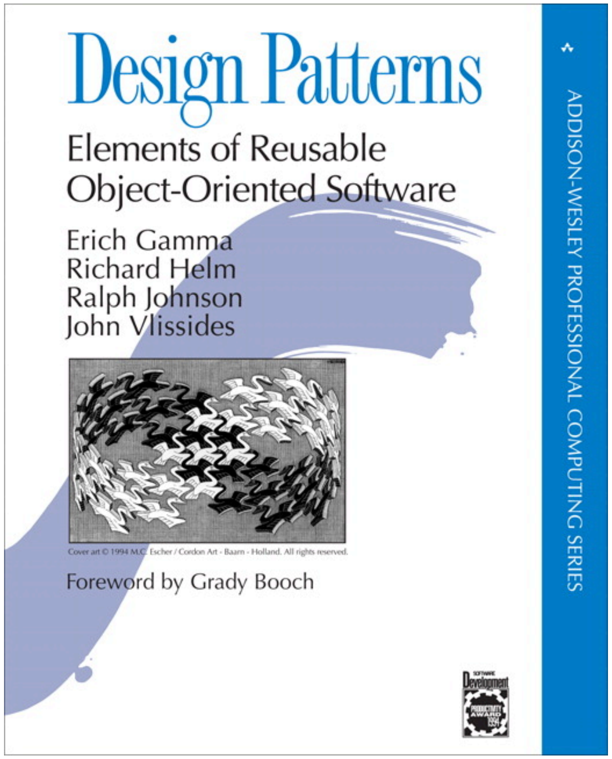
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

cornell CS PhD

Good online references:

http://www.codeproject.com/Articles/430590/Design-Patterns-of-Creational-Design-Patterns

https://sourcemaking.com/design_patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Good rules of thumb for success

"Program testing can at best show the presence of errors but never their absence."

- Edsger W. Dijkstra

1. Think fully before testing!
2. Document your code and communicate effectively with your team
3. Write methods that have a clear, crisp purpose

Thank you!
Have a great summer!