Additional Design Patterns

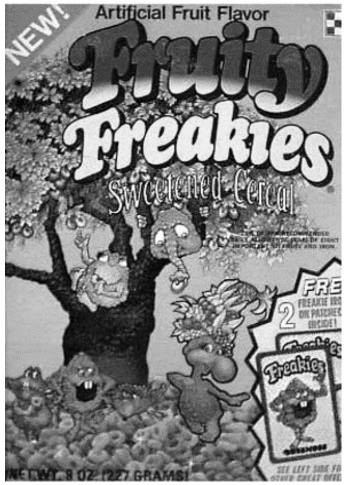
Chain of Responsibility
and
Flyweight

Jason Robison
CSCI 5448

The Chain of Responsibility Pattern

Chain of Responsibility

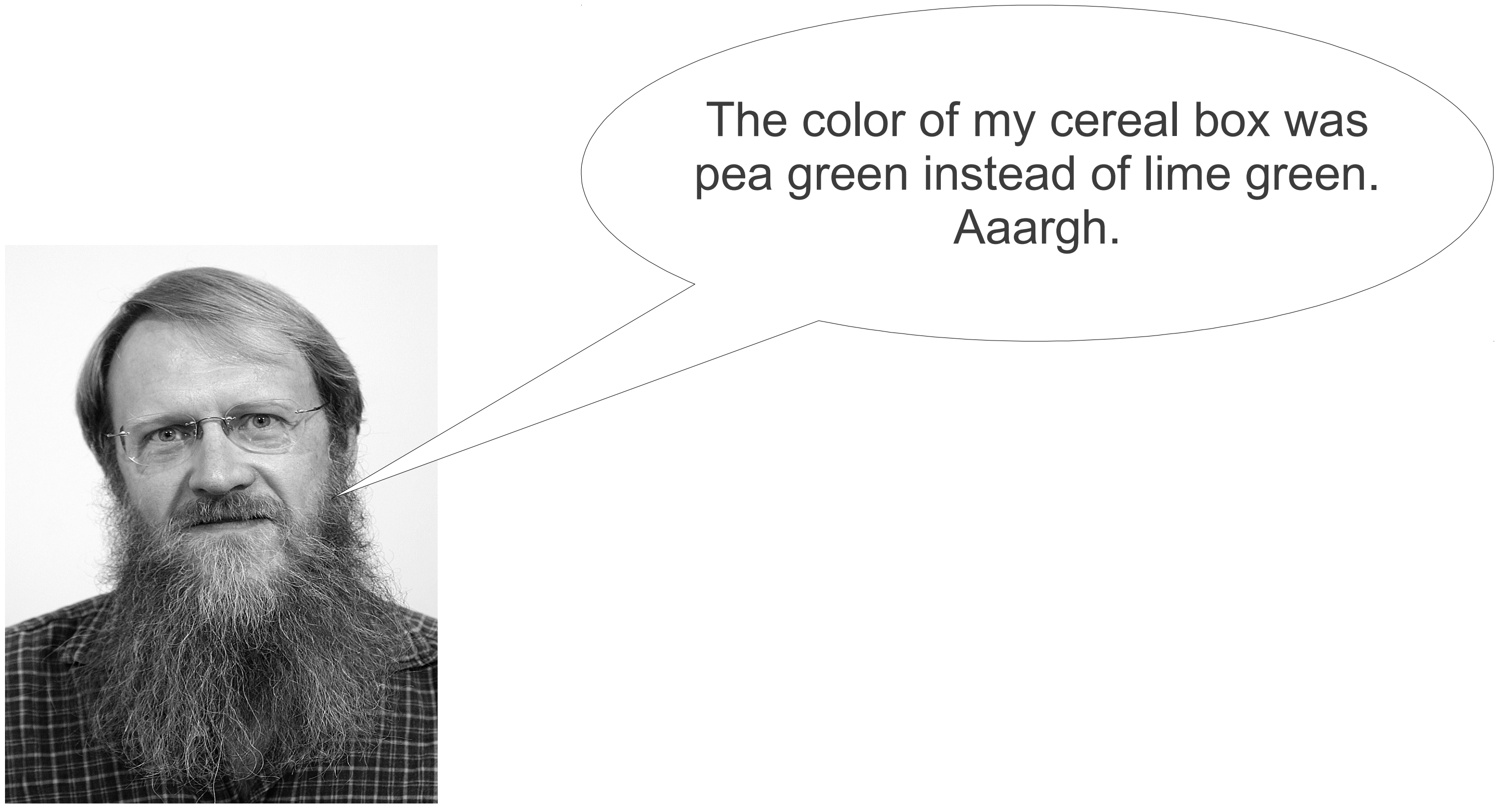
- Ted has a job handling customer complaints for the most popular breakfast food in the country, Fruity Freakies cereal.



Ted

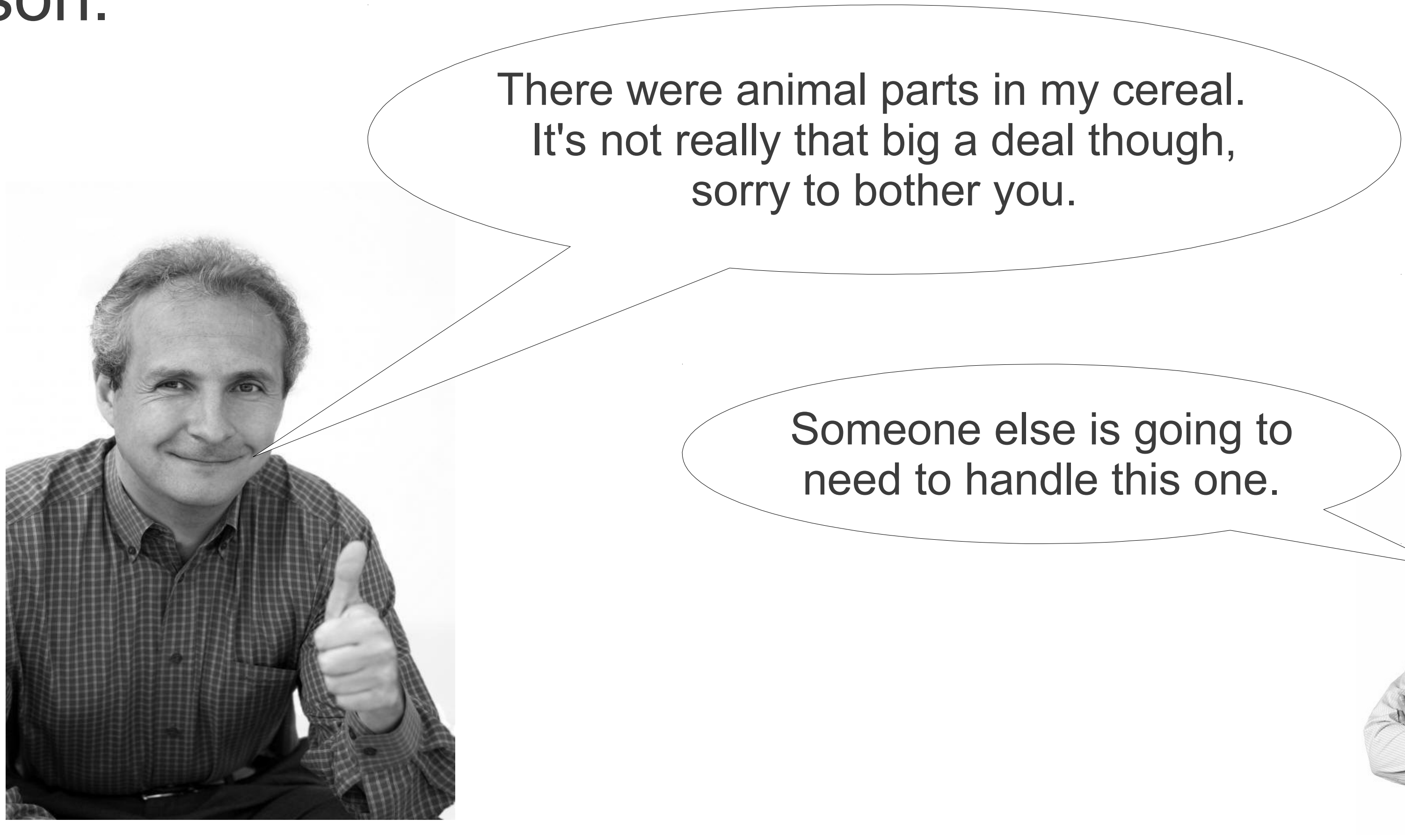
Chain of Responsibility

- Some complaints aren't as important. Ted just handles these complaints by himself.


The color of my cereal box was pea green instead of lime green. Aaargh.

Chain of Responsibility

- But other complaints are more important. These types of complaints Ted needs to send to one of his bosses. Ted tries his best to forward the requests to the right person.


There were animal parts in my cereal. It's not really that big a deal though, sorry to bother you.
Someone else is going to need to handle this one.

Chain of Responsibility

- Some problems need to go to Ted's immediate supervisor. Really important problems need to go all the way to the CEO.
- Ted has been working at Fruity Freakies for a while and has a decent idea of where to forward certain requests, for the most part.

Chain of Responsibility

- But Ted has gotten a little sick of dealing with things by hand, and decides to write a computer program. Ted took many software classes in college, and indeed later switched from computer science to customer service (he just didn't get to interact with enough angry people in CS for his tastes). But Ted remembers the lessons learned from his study of software design and thinks he can design a software system he can use to deal with all the beaurocratic decisions for him.

Chain of Responsibility

- Ted writes the following code:

```
if ( problemType == 1 ) {
   Ted ted = new Ted();
   ted.handleProblem(problem);
}
else if ( problemType == 2 ) {
   Supervisor supervisor = new Supervisor();
   supervisor.handleProblem(problem);
}
else if ( problemType == 3 ) {
   MiddleManager manager = new MiddleManager();
   manager.handleProblem(problem);
}
else if ( problemType == 4 ) {
   CEO ceo = new CEO();
   ceo.handleProblem(problem);
}
```

Chain of Responsibility

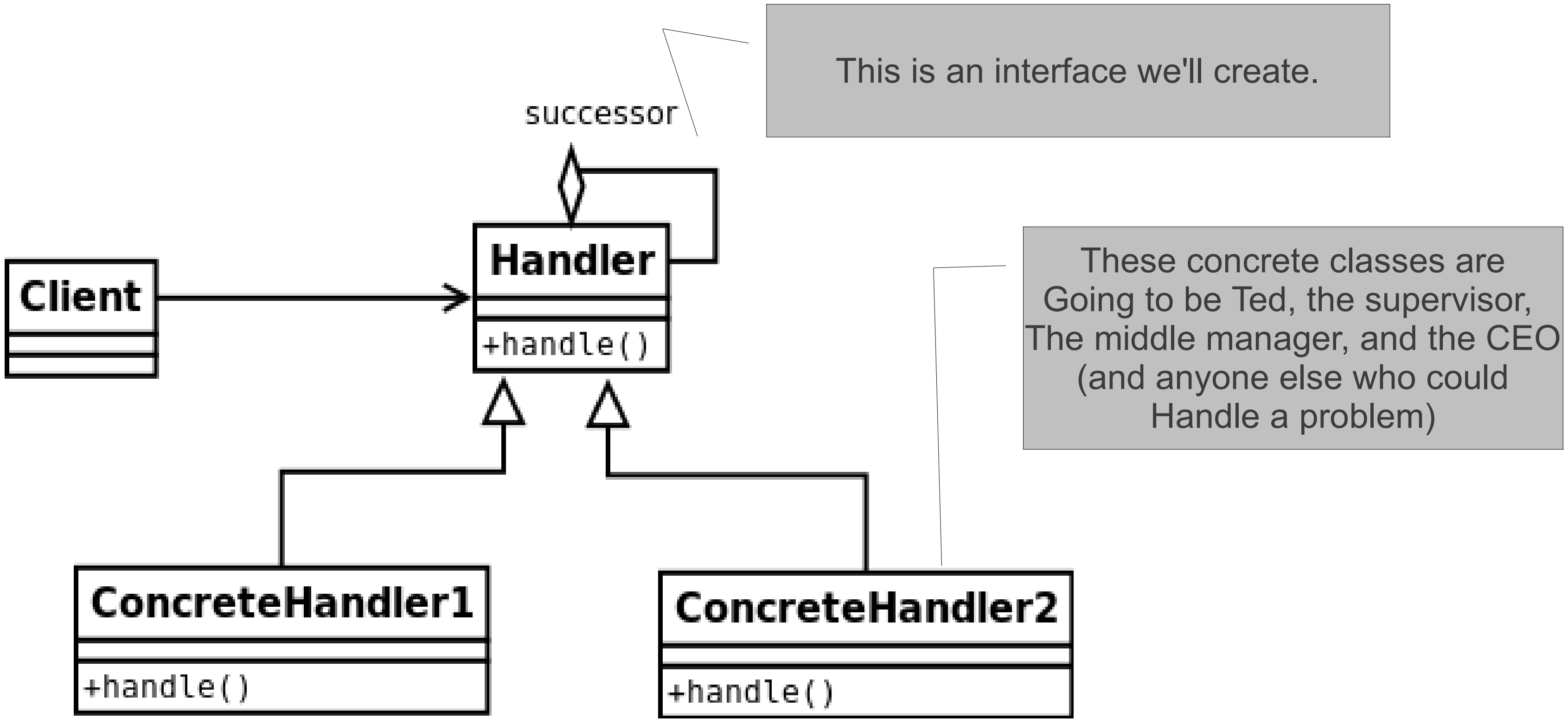
- And he feels great about his solution! Even better, it works!
- Well... kind of. Two big problems develop:
  - First, the CEO can be a cranky fellow and doesn't like too many things demanding his attention. It's always possible someone lower on the chain *could* conceivably handle a problem even if it is very important.
  - Second, Freaky Fruities is big on “process”. The company often tinkers with the job responsibilities of its managers. And so Ted is often changing around his if statements.
- Dang, Ted thinks. I should have remembered design principles from my object oriented analysis and design class – encapsulate what varies.

Chain of Responsibility

- Ted remembers a presentation from that same OOA&D class on a pattern called the Chain of Responsibility. It stands out in his mind because the presentation coincidentally featured a character also named Ted who was also interested in customer service. In fact, it was that presentation that made Ted start thinking about changing his life and going into customer service at all.

Chain of Responsibility

- Introducing the Chain of Responsibility Pattern

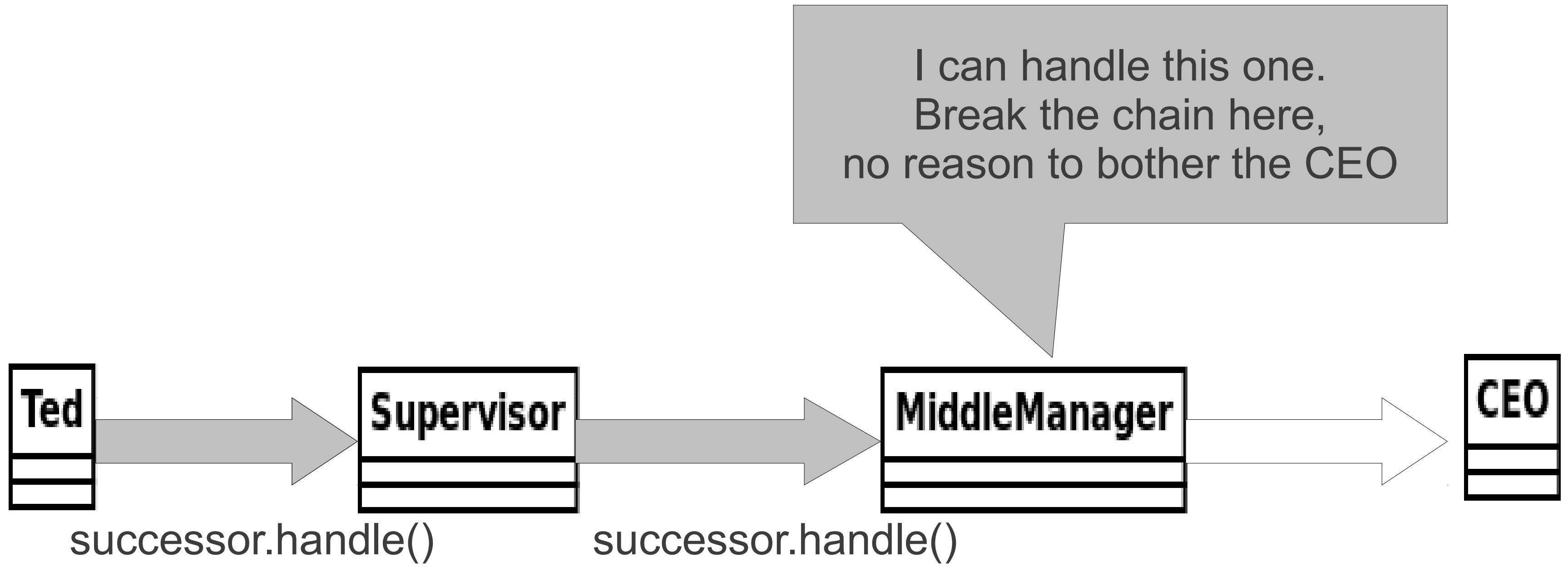


This is an interface we'll create.

These concrete classes are
Going to be Ted, the supervisor,
The middle manager, and the CEO
(and anyone else who could
Handle a problem)

Chain of Responsibility

- The handlers will be chained together, and once someone in the chain can handle a problem, we stop.



Chain of Responsibility

- Uses and advantages:
  - Use in cases where a client doesn't care who handles a request, only that it is handled.
  - Decouples clients from handlers of their requests.
- Drawbacks:
  - There is no guarantee a request will be handled at all – it could fall off the end of the chain.

Chain of Responsibility

- First Ted does the easy part and codes up the interface:

```
public interface ProblemHandler
{
   public void handleProblem();
}
```

Chain of Responsibility

- Next, codes up his classes to make them usable by the CoR pattern (as it is frequently abbreviated). Here's his MiddleManager class:

```
public class MiddleManager implements ProblemHandler {

   private ProblemHandler successor;

   public void setSuccessor(ProblemHandler handler) {
      successor = handler;
   }

   public void handleProblem(Problem problem) {
      if (canHandle(problem)) {
         System.out.println("Middle manager handling the problem, " +
            "stopping the chain...");
      }
      else {
         successor.handleProblem(problem);
      }
   }
}
```

Chain of Responsibility

- And finally, Ted edits his client program to make use of the chain:

Notice how we can now code to an interface instead of an implementation

```
ProblemHandler ted = new Ted();

ProblemHandler supervisor = new Supervisor();

ProblemHandler middleManager = new MiddleManager();

ProblemHandler ceo = new CEO();
```

Setting up the chain is as easy as setting the successors. If the chain changes later, this is the only place we need to change it

```
ted.setSuccessor(supervisor);

supervisor.setSuccessor(middleManager);

middleManager.setSuccessor(ceo);


ted.handleProblem(problem);
```

Chain of Responsibility

- Now Ted can go back to what he really loves: the people.

Chain of Responsibility

- Variations
  - In classic CoR, the chain stops when one object in the chain handles the request. One variation is to keep passing the request along, even after it has been handled.
  - Another variation is that in addition to continuing to pass the request along the chain, handlers may modify the request itself. In order to keep the original request unchanged, this is sometimes done with the aid of the Decorator pattern – each handler that handles the request may decorate the request.

Chain of Responsibility

- Known modern uses:
  - Microsoft Windows hook framework:
    - http://msdn.microsoft.com/en-us/library/ms644959
  - Java servlet filters:
    - http://www.oracle.com/technetwork/java/filters-137243.html

The Flyweight Pattern

Flyweight

- Bill is the lead developer for SandwichCorp. SandwichCorp is the worldwide leader in delivering fresh sub sandwiches right to your door. It's Bill's job to work on the software SandwichCorp uses to place orders online and let the individual stores know how to prepare and deliver the sandwiches.
- Bill and the rest of team are good developers. They've used object oriented analysis and design in their program, and as a result the program is thankfully relatively easy to change, update, and upgrade based on customer needs.

Flyweight

- Using proper object oriented design, the team abstracts the idea of a “sub sandwich” with a SubSandwich object. A SubSandwich has a sandwich type, a destination address for the sandwich to be delivered to, and “extras” that the sandwich preparer should add to the sandwich.
- The SubSandwich class:

```
public class SubSandwich {
   private SandwichType sandType;
   private String destinationAddress;
   private String sandwichExtras;

   public SubSandwich(SandwichType t, String dest, String extras) {
      this.sandType = t;
      this.destinationAddress = dest;
      this.sandwichExtras = extras;
   }

   public void prepare() {
      System.out.println("Preparing sandwich with " + sandwichExtras + " extras...");
   }

   public void deliver() {
      System.out.println("Delivering sandwich to " + destinationAddress + "...");
   }
}
```

Flyweight

- When it's time to prepare and order a new sandwich, the code looks something like this:

```
SubSandwich sandwich
    = new SubSandwich(SandwichType.HAM, "Destination Address", "Extras");
sandwich.prepare();
sandwich.deliver();
```

Flyweight

- But Bill and his team have run into a problem. SandwichCorp's operations have grown immensely over the past year. And during peak load times, systems running the software have actually started to report out of memory errors.
- It appears that there's just too darn many SubSandwich objects on the heap and it's using up way too much memory!

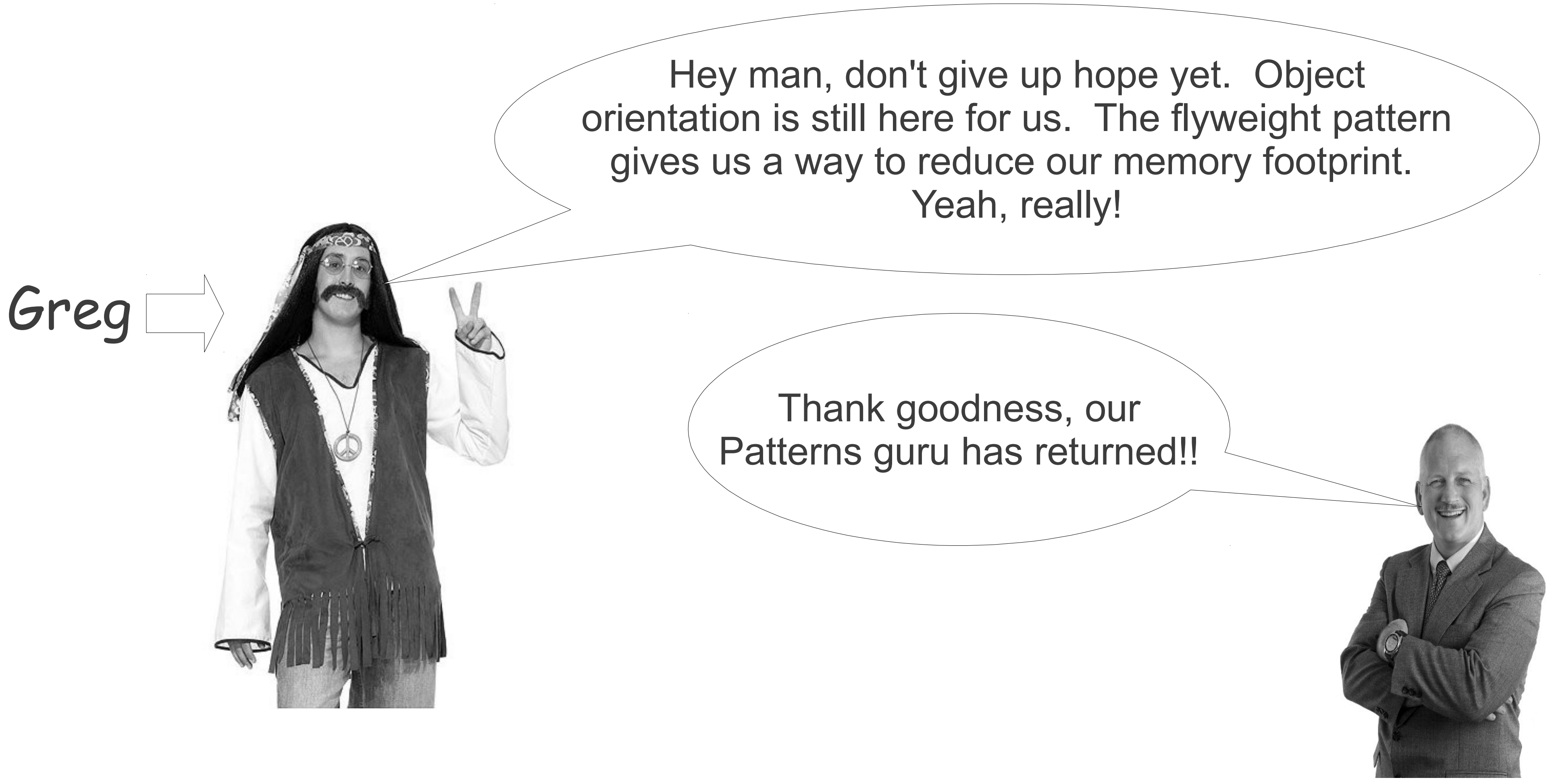
Flyweight

- Bill and the team discuss the problem in depth. There's a general frustration among the team, a feeling that using proper object oriented technique has ended up biting them in the back.

I told you we should have written the whole program in assembly
sigh

Flyweight

- Greg shows up late after returning from a life journey, and overhears the conversation.

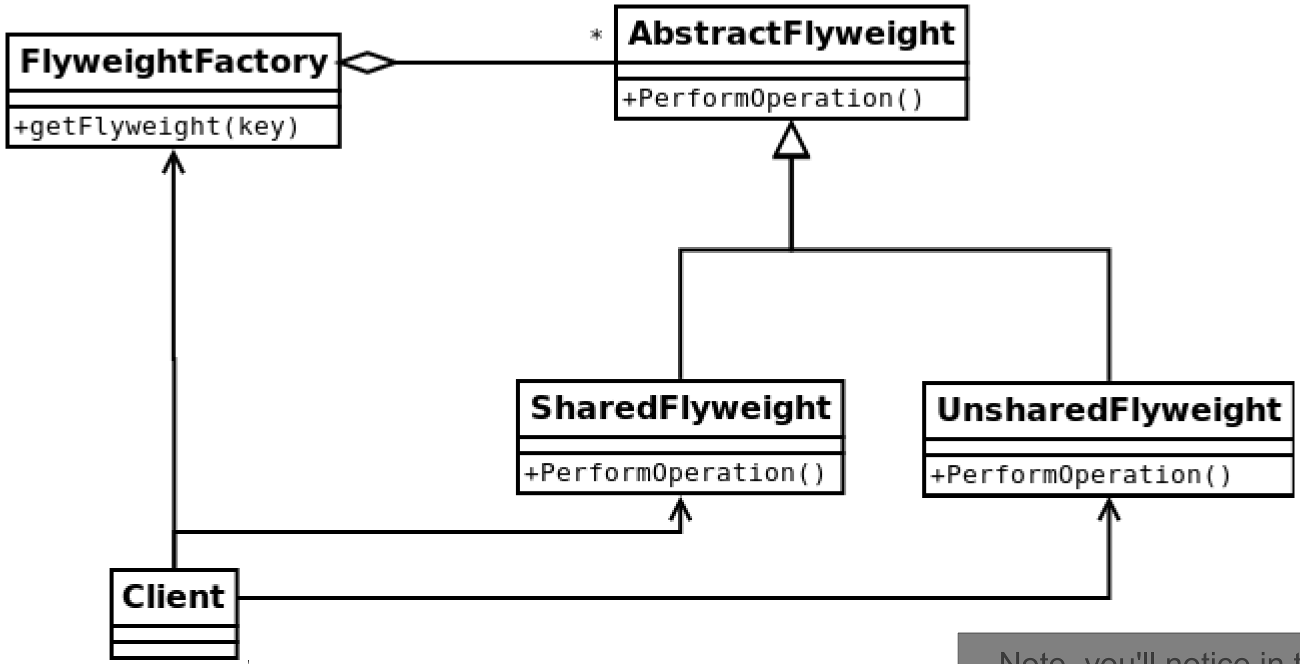

Greg
Hey man, don't give up hope yet. Object orientation is still here for us. The flyweight pattern gives us a way to reduce our memory footprint. Yeah, really!
Thank goodness, our Patterns guru has returned!!

Flyweight

- Introducing the flyweight pattern:

Instead of creating new objects yourself, you ask this factory guy here to give you an object of the type you want.

This is our SubSandwich



This is just the class you need the objects in

Note, you'll notice in this example we won't have an abstract class, but we're still using the flyweight pattern – the ability subclass just gives additional decoupling power to the pattern that we aren't going to use.

Flyweight

- Uses and advantages:
  - Most design patterns are used to increase the maintainability of your code.
  - Flyweight improves performance by reducing memory usage!
  - Flyweight is useful when you are creating very large numbers of objects that have very similar properties.

Flyweight

Check it out guys. We'll create a SubSandwichFactory –
that will be our “FlyweightFactory” - that will hold ONE sandwich object
for each sandwich type. The “extras” and “address” we'll pass
as parameters to methods instead of store them in the
object. See, let me show you:

Flyweight

- The SubSandwichFactory class using the flyweight pattern:

```
public class SubSandwichFactory {
   private HashMap<SandwichType, SubSandwich> sandwiches
      = new HashMap<SandwichType, SubSandwich>();

   public SubSandwich getSandwich(SandwichType type) {
      if ( !sandwiches.containsKey(type) ) {
         sandwiches.put(type, new SubSandwich(type));
      }
      return sandwiches.get(type);
   }
}
```

Flyweight

- And the new flyweight pattern SubSandwich class:

```
public class SubSandwich {
   private SandwichType sandType;

   public SubSandwich(SandwichType t) {
      this.sandType = t;
   }

   public void prepare(String sandwichExtras) {
      //System.out.println("Preparing sandwich with " + sandwichExtras + "
extras...");
   }

   public void deliver(String destinationAddress) {
      System.out.println("Delivering sandwich to " + destinationAddress + "...");
   }
}
```

Flyweight

- Known modern uses:
  - Java Strings: Java Strings are flyweights!
  - Borders in Swing
  - Java Swing Tree nodes