Design Patterns in the 21st Century

29th May 2015

@SamirTalwar

What do you want
from me?

I want you to stop using design patterns.

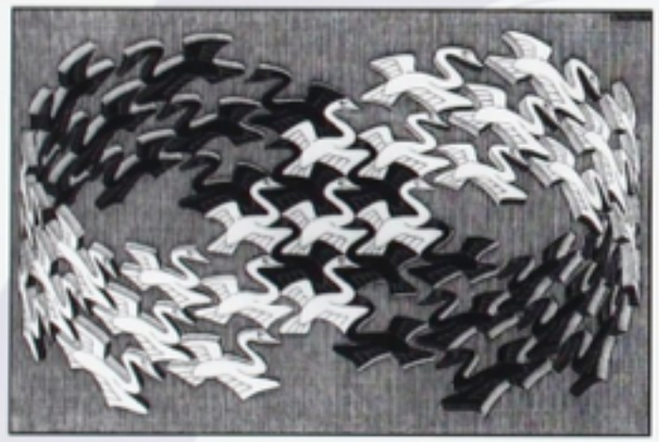
What do you want
from me?

I want you to stop using design patterns…
like it's 1999.

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

– Christopher Alexander

And now, an aside, on functional programming.

```
int courses = 3;

Course dessert =
    prepareCake.madeOf(chocolate);

Preparation prepareCake = new Preparation() {
    @Override
    public Course madeOf(Ingredient mmmmm) {
        return
            new CakeMix(eggs, butter, sugar)
                .combinedWith(mmmmm);
    }
};
```

```
Preparation prepareCake = new Preparation() {
    @Override
    public Course madeOf(Ingredient mmmmm) {
        return
            new CakeMix(eggs, butter, sugar)
                .combinedWith(mmmmm);
    }
};

@FunctionalInterface
interface Preparation {
    Course madeOf(Ingredient mmmmm);
}
```

```
Preparation prepareCake = new Preparation() {
    @Override
    public Course madeOf(Ingredient mmmmm) {
        return
            new CakeMix(eggs, butter, sugar)
                .combinedWith(mmmmm);
    }
};

Preparation prepareCake =
    mmmmm ->
        new CakeMix(eggs, butter, sugar)
            .combinedWith(mmmmm);
```

```
Preparation prepareCake =
      mmmmm ->
            new CakeMix(eggs, butter, sugar)
                  .combinedWith(mmmmm);
```

```
Mix mix = new CakeMix(eggs, butter, sugar);
Preparation prepareCake =
      mix::combinedWith;
```

```
Preparation prepareCake =
    mmmmm ->
        new CakeMix(eggs, butter, sugar)
            .combinedWith(mmmmm);

Mix mix = new CakeMix(eggs, butter, sugar);
Preparation prepareCake =
    mix::combinedWith;
```

```
Course combinedWith(Ingredient);

@FunctionalInterface
interface Preparation {
    Course madeOf(Ingredient mmmmm);
}
```

Well.

On to the Good Stuff

The Abstract Factory pattern

Abstract Factory

```
public interface Bakery {
    Pastry bakePastry(Topping topping);

    Cake bakeCake();
}

public class DanishBakery implements Bakery {
    @Override
    public Pastry bakePastry(Topping topping) {
        return new DanishPastry(topping);
    }

    @Override
    public Cake bakeCake() {
        return new Æblekage(); // mmmm, apple cake…
    }
}
```

Client

Abstract Factory

Bakery

Pastry

Product A

Concrete Factory 1

DanishBakery

Concrete Factory 2

BritishBakery

Product A 2

RhubarbTart

Product A 1

DanishPastry

Cake

Product B

Product B 2

VictoriaSponge

Product B 1

Æblekage

Abstract Factory

```
public interface Bakery {
    Pastry bakePastry(Topping topping);
}
```

```
public class DanishBakery implements Bakery {
    @Override
    public Pastry bakePastry(Topping topping) {
        return new DanishPastry(topping);
    }
}
```

Abstract Factory

```
public class DanishBakery implements Bakery {
    @Override
    public Pastry bakePastry(Topping topping) {
        return new DanishPastry(topping);
    }
}

Bakery danishBakery = topping ->
    new DanishPastry(topping);

Bakery danishBakery = DanishPastry::new;
```

Abstract Factory

```
package java.util.function;

/**
 * Represents a function that
 * accepts one argument and produces a result.
 *
 * @since 1.8
 */
@FunctionalInterface
public interface Function<T, R> {
    /**
     * Applies this function to the given
argument.
     */
    R apply(T t);

    ...
}
```

Abstract Factory

```
public class DanishBakery
        implements Function<Topping, Pastry> {
    @Override
    public Pastry apply(Topping topping) {
        return new DanishPastry(topping);
    }
}

Function<Topping, Pastry> danishBakery
    = topping ->
        new DanishPastry(topping);

Function<Topping, Pastry> danishBakery
    = DanishPastry::new;
```

Client

Abstract Factory

Bakery

Pastry

Product A

Product A 2

RhubarbTart

Product A 1

DanishPastry

Concrete Factory 2

BritishBakery

Product B

Cake

Product B 2

VictoriaSponge

Product B 1

Æblekage

The Adapter pattern

Adapter

```
interface Fire {
    <T> Burnt<T> burn(T thing);
}

interface Oven {
    Food cook(Food food);
}

class WoodFire implements Fire { ... }
```

```
class MakeshiftOven
        extends WoodFire
        implements Oven {
    @Override public Food cook(Food food) {
        Burnt<Food> nastyFood = burn(food);
        return nastyFood.scrapeOffBurntBits();
    }
}
```

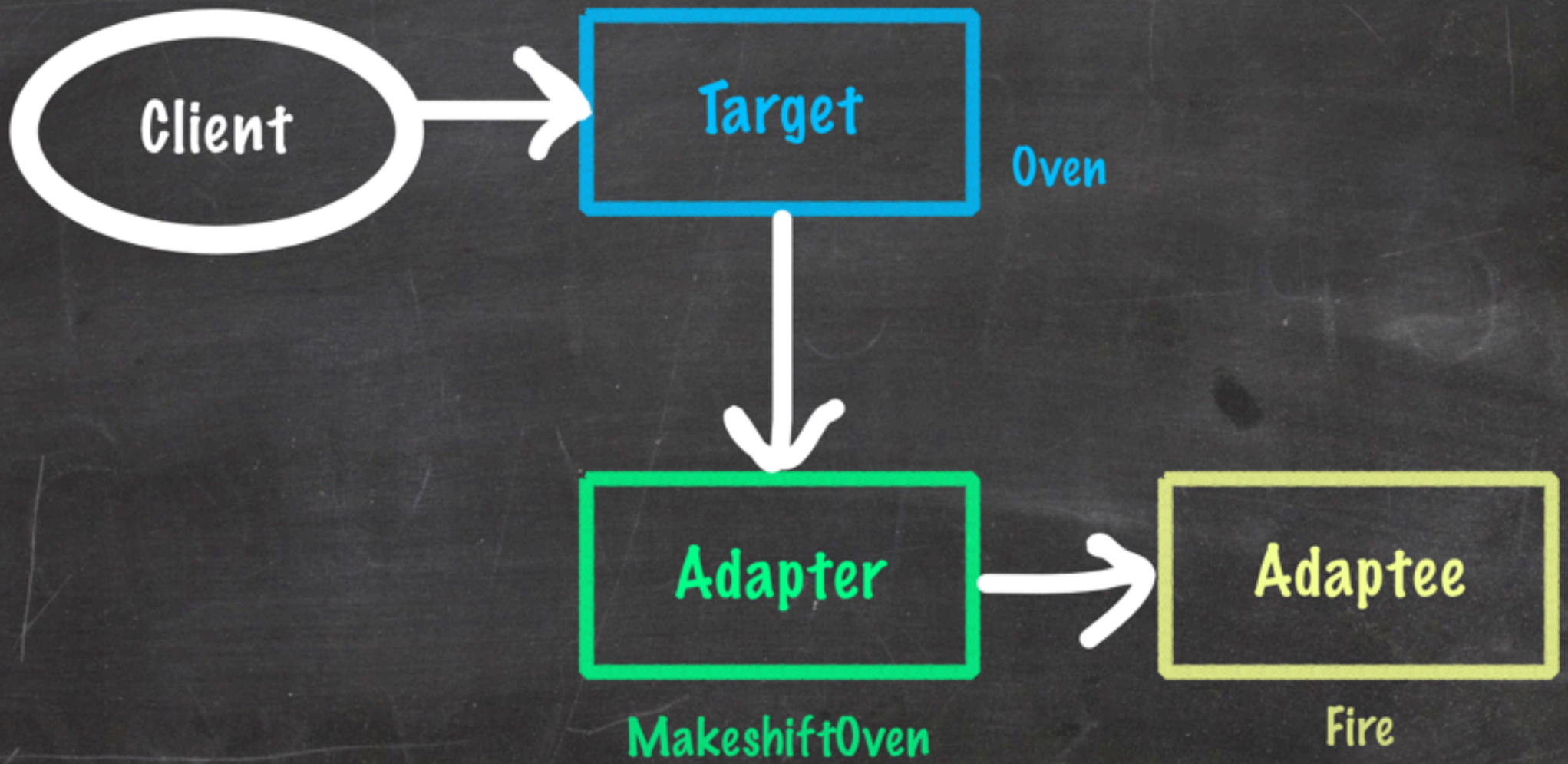
Adapter

```
interface Fire {
    <T> Burnt<T> burn(T thing);
}

interface Oven {
    Food cook(Food food);
}

class MakeshiftOven implements Oven {
    private final Fire fire;

    public MakeshiftOven(Fire fire) { /* ... */ }

    @Override public Food cook(Food food) {
        Burnt<Food> nastyFood = fire.burn(food);
        return nastyFood.scrapeOffBurntBits();
    }
}
```

Adapter

```
interface Oven {
    Food cook(Food food);
}
```

```
Oven oven = new MakeshiftOven(fire);
Food bakedPie = oven.cook(pie);
```


Client
Target
Oven
Adapter
MakeshiftOven
Adaptee
Fire

Adapter

```
interface Oven {
    Food cook(Food food);
}

class MakeshiftOven implements Oven {
    private final Fire fire;
    public MakeshiftOven(Fire fire) { /* ... */ }

    @Override public Food cook(Food food) {
        Burnt<Food> nastyFood = fire.burn(food);
        return nastyFood.scrapeOffBurntBits();
    }
}
```

Adapter

```
class MakeshiftOven implements Oven {
    private final Fire fire;

    public MakeshiftOven(Fire fire) { /* ... */ }

    @Override public Food cook(Food food) {
        Burnt<Food> nastyFood = fire.burn(food);
        return nastyFood.scrapeOffBurntBits();
    }
}

Oven oven = food -> {
    Burnt<Food> nastyFood = fire.burn(food);
    return nastyFood.scrapeOffBurntBits();
};
Food bakedPie = oven.cook(pie);
```

Adapter

```
Oven oven = food -> {
    Burnt<Food> nastyFood = fire.burn(food);
    return nastyFood.scrapeOffBurntBits();
};
```

```
Oven oven = food ->
    fire.burn(food).scrapeOffBurntBits();
```

Adapter

```
Oven oven = food ->
    fire.burn(food).scrapeOffBurntBits();



// Do *not* do this.
Function<Food, Burnt<Food>> burn
    = fire::burn;
Function<Food, Food> cook
    = burn.andThen(Burnt::scrapeOffBurntBits);
Oven oven = cook::apply;
Food bakedPie = oven.cook(pie);
```

Adapter

```
package java.util.concurrent;

/**
 * An object that executes
 * submitted {@link Runnable} tasks.
 */
public interface Executor {
    void execute(Runnable command);
}
```

Adapter

```
public interface Executor {
    void execute(Runnable command);
}
```

```
Executor executor = ...;
Stream<Runnable> tasks = ...;

tasks.forEach(executor);
```

Adapter

```
public interface Stream<T> {
    ...

    void forEach(Consumer<? super T> action);

    ...
}
```

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);

    ...
}
```

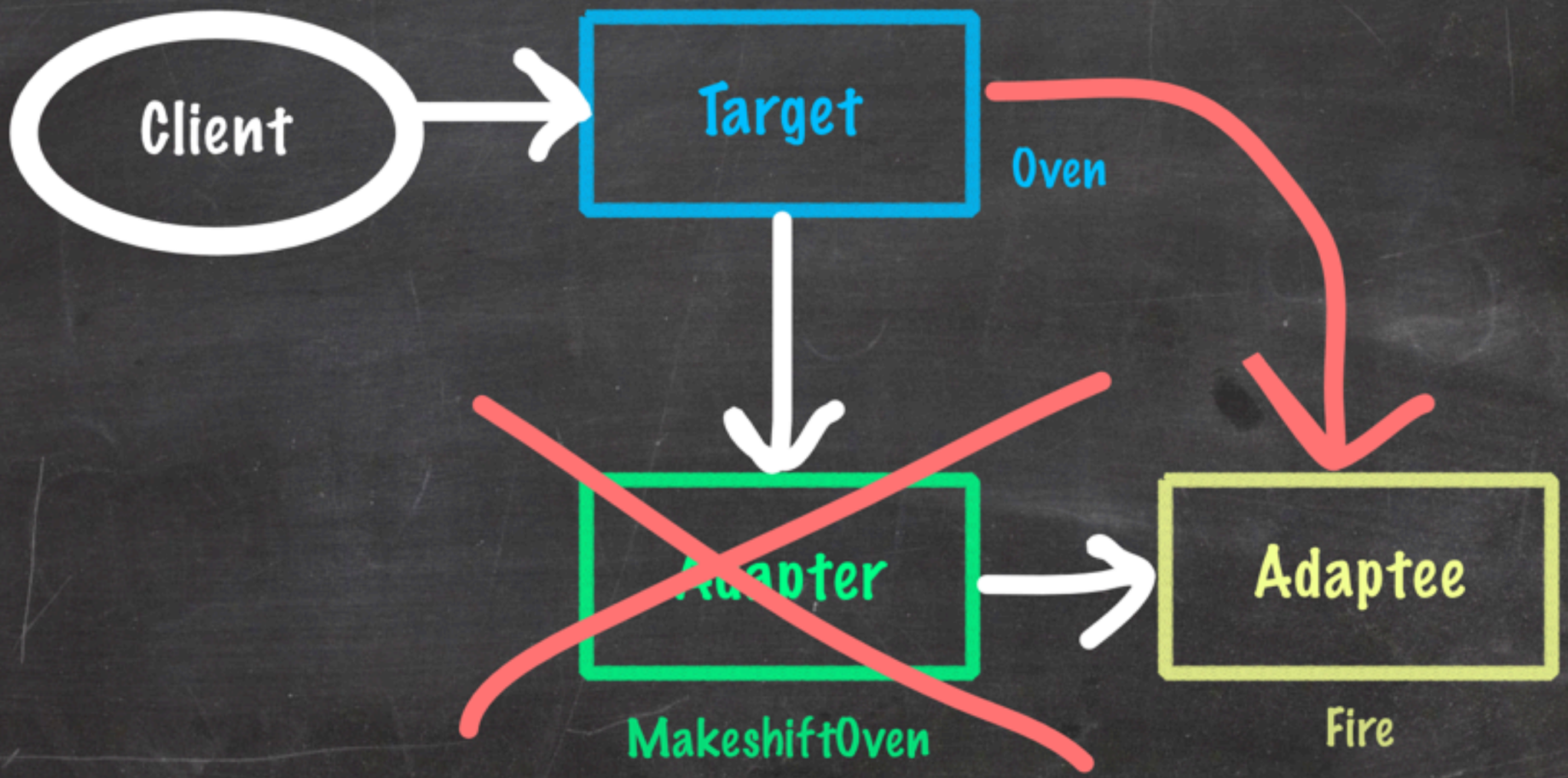
Adapter

```
public interface Executor {
    void execute(Runnable command);
}

@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

```
Executor executor = ...;
Stream<Runnable> tasks = ...;

tasks.forEach(task -> executor.execute(task));

tasks.forEach(executor::execute);
```

Adapter

`executor::execute`


Client
Target
Oven
Adapter
Adaptee
MakeshiftOven
Fire

The Chain of Responsibility pattern

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff alice = new PieChef();
    KitchenStaff bob = new DollopDistributor();
    KitchenStaff carol = new CutleryAdder();
    KitchenStaff dan = new Server();

    alice.setNext(bob);
    bob.setNext(carol);
    carol.setNext(dan);

    Patron patron = new Patron();
    alice.prepare(new Pie()).forPatron(patron);

    assertThat(patron, hasPie());
}
```

Chain of Responsibility

```
public final class HitCounterFilter implements Filter {
    private FilterConfig filterConfig = null;

    public void init(FilterConfig filterConfig) throws ServletException {
        this.filterConfig = filterConfig;
    }

    public void destroy() {
        this.filterConfig = null;
    }

    public void doFilter
            (ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        if (filterConfig == null)
            return;

        Counter counter =
            (Counter)filterConfig.getServletContext().getAttribute("hitCounter");

        StringWriter sw = new StringWriter();
        PrintWriter writer = new PrintWriter(sw);
        writer.println("The number of hits is: " + counter.incCounter());
        writer.flush();
        filterConfig.getServletContext().log(sw.getBuffer().toString());

        chain.doFilter(request, response);
    }
}
```

Chain of Responsibility

```
public final class HitCounterFilter
        implements Filter {
    // init and destroy

    public void doFilter(
            ServletRequest request,
            ServletResponse response,
            FilterChain chain) {
        int hits = getCounter().incCounter();
        log(“The number of hits is ” + hits);
        chain.doFilter(request, response);
    }
}
```

Chain of Responsibility

```
public final class SwitchEncodingFilter
        implements Filter {
    // init and destroy

    public void doFilter(
            ServletRequest request,
            ServletResponse response,
            FilterChain chain) {

        request.setEncoding(“UTF-8”);
        chain.doFilter(request, response);
    }
}
```

Chain of Responsibility

```
public final class AuthorizationFilter
        implements Filter {
    // init and destroy

    public void doFilter(
            ServletRequest request,
            ServletResponse response,
            FilterChain chain) {
        if (!user.canAccess(request))
            throw new AuthException(user);
        chain.doFilter(request, response);
    }
}
```

Handler

KitchenStaff

Client

Handler 1

PieChef

Handler 2

DollopDistributor

Handler 3

CutleryAdder

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff alice = new PieChef();
    KitchenStaff bob = new DollopDistributor();
    KitchenStaff carol = new CutleryAdder();
    KitchenStaff dan = new Server();

    alice.setNext(bob);
    bob.setNext(carol);
    carol.setNext(dan);

    Patron patron = new Patron();
    alice.prepare(new Pie()).forPatron(patron);

    assertThat(patron, hasPie());
}
```

Chain of Responsibility

Ick.

- So much mutation.
- Where's the start?
- What happens if we change the order?
- Why do we duplicate the chain logic in every element?

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff alice = new PieChef();
    KitchenStaff bob = new DollopDistributor();
    KitchenStaff carol = new CutleryAdder();
    KitchenStaff dan = new Server();

    alice.setNext(bob);
    bob.setNext(carol);
    carol.setNext(dan);

    Patron patron = new Patron();
    alice.prepare(new Pie()).forPatron(patron);

    assertThat(patron, hasPie());
}
```

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff dan = new Server();
    KitchenStaff carol = new CutleryAdder(dan);
    KitchenStaff bob = new DollopDistributor(carol);
    KitchenStaff alice = new PieChef(bob);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = alice.prepare(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff<WithCutlery<Meal>> dan
            = new Server();
    KitchenStaff<Meal> carol
            = new CutleryAdder(dan);
    KitchenStaff<CookedPie> bob
            = new DollopDistributor(carol);
    KitchenStaff<UncookedPie> alice
            = new PieChef(bob);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = alice.prepare(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```

Chain of Responsibility

```
public PieChef(KitchenStaff next) {
    this.next = next;
}
```

```
public PieChef(KitchenStaff<CookedPie> next) {
    this.next = next;
}
```

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff<WithCutlery<Meal>, Serving> dan
            = new Server();
    KitchenStaff<Meal, Serving> carol
            = new CutleryAdder().then(dan);
    KitchenStaff<CookedPie, Serving> bob
            = new DollopDistributor().then(carol);
    KitchenStaff<UncookedPie, Serving> alice
            = new PieChef().then(bob);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = alice.prepare(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```

Chain of Responsibility

```
interface KitchenStaff<I, O> {
    O prepare(I input);

    default <Next> KitchenStaff<I, Next>
        then(KitchenStaff<O, Next> next) {
          return input -> {
              O output = prepare(input);
              return next.prepare(output);
          };
    }
}
```

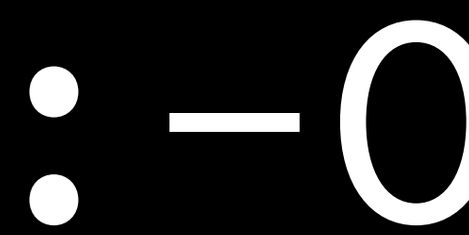
Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    KitchenStaff<UncookedPie, CookedPie> alice
            = new PieChef();
    KitchenStaff<CookedPie, Meal> bob
            = new DollopDistributor();
    KitchenStaff<Meal, WithCutlery<Meal>> carol
            = new CutleryAdder();
    KitchenStaff<WithCutlery<Meal>, Serving> dan
            = new Server();

    KitchenStaff<UncookedPie, Serving> staff =
            alice.then(bob).then(carol).then(dan);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = staff.prepare(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```

Chain of Responsibility

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);

    ...

    default <V> Function<T, V> andThen
      (Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    ...
}
```

Chain of Responsibility

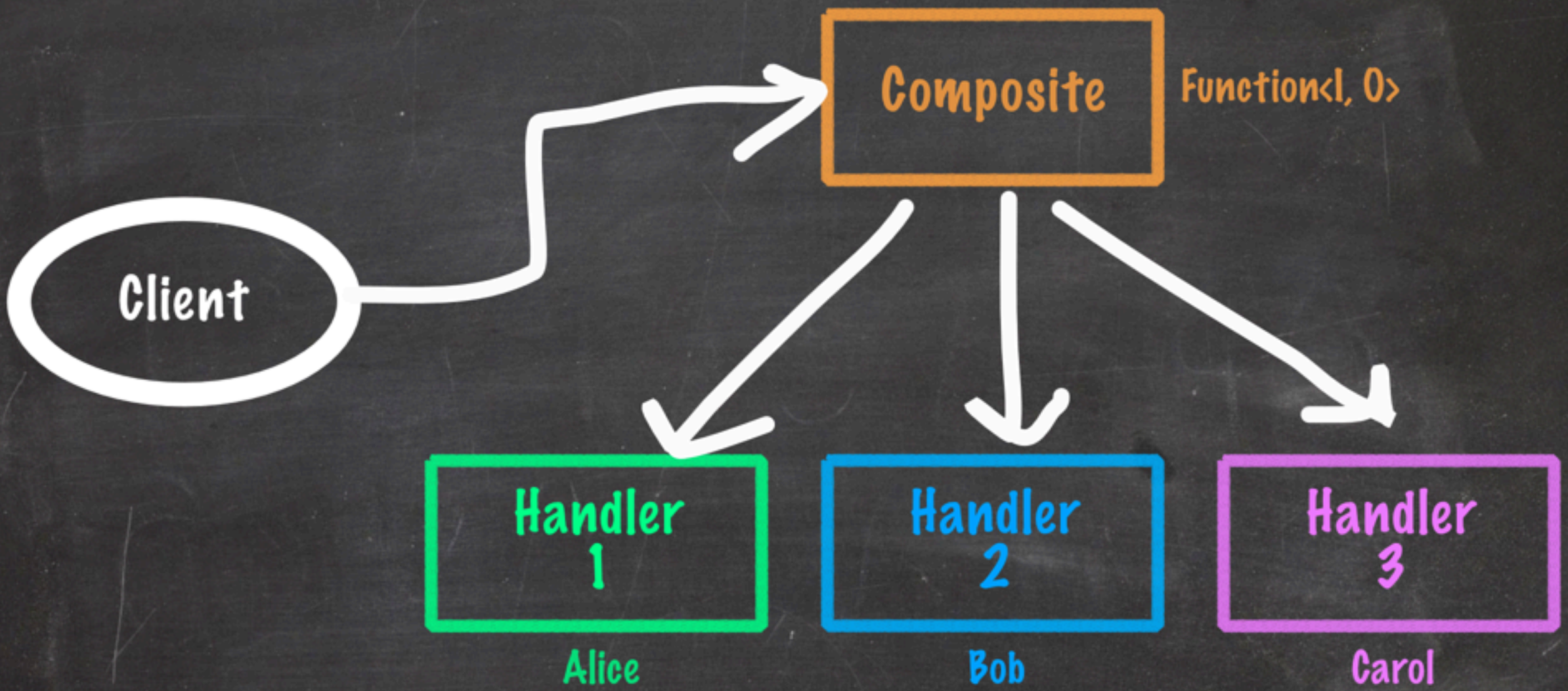


Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    Function<UncookedPie, CookedPie> alice
            = new PieChef();
    Function<CookedPie, Meal> bob
            = new DollopDistributor();
    Function<Meal, WithCutlery<Meal>> carol
            = new CutleryAdder();
    Function<WithCutlery<Meal>, Serving> dan
            = new Server();

    Function<UncookedPie, Serving> staff =
            alice.andThen(bob).andThen(carol).andThen(dan);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = staff.apply(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```

Chain of Responsibility

```
@Test public void hungryHungryPatrons() {
    Function<UncookedPie, CookedPie> alice = UncookedPie::cook;
    Function<CookedPie, Meal> bob = CookedPie::addCream;
    Function<Meal, WithCutlery<Meal>> carol = WithCutlery::new;
    Function<WithCutlery<Meal>, Serving> dan = Serving::new;

    Function<UncookedPie, Serving> staff =
            alice.andThen(bob).andThen(carol).andThen(dan);

    Patron hungryPatron = new HungryPatron();
    Patron happyPatron = staff.prepare(new Pie())
            .forPatron(hungryPatron);

    assertThat(happyPatron, hasPie());
}
```


Composite
Function<I, O>
Client
Handler 1
Handler 2
Handler 3
Alice
Bob
Carol

So.

So. What's your point?

I want you to use design patterns…
like it's 1958.

λOOP

Credits

Bakery, by Boris Bartels

Just baked, by Sergio Russo

cherry pie with crumble topping, by Ginny

endurance

craft at heart

talks.samirtalwar.com

Thank you.