Lecture 13
Decorator, continued

- Decorator applies to interfaces (not classes).
  - Caveat: as I am teaching it here.
  - It’s the interface that is being decorated.
    - Extending an interface is not the same as subclassing
    - This is why we encounter statements like:
      - Decorators provide a flexible alternative to subclassing for extending functionality.
      - See Decorator in Java tutorial from Abhi On Java link in readings.
  - This is why we needed Song to have an associated SongInterface
Decorator Illustrated

```java
interface I {
    void do_something();
}

class C implements I {
    void do_something() {
        ...
    }
}

interface DI extends I {
    void do_more();
}

class DC implements DI {
    private I wrapped_i;
    public (I i_obj) {
        wrapped_i = i_obj;
    }
    void do_something() {
        wrapped_i.do_something();
    }
    void do_more() {
        ...
    }
}
```
Revisiting Why Decorator

• Localized need for extra functionality.
  – Objects are already created by someone else.
  – Assignment 5
    • Consider Belt’s methods:
      – public Sushi getSushiAt(int position)
      – public Sushi takeOffSushiAt(int position)
      – public int placeNearestTo(Sushi sushi, int target_position)
      – public Iterator<Sushi> iterator()  
    • Don’t want to change operations on Belt  
      – Other parts of system will be creating Sushi and adding it to the Belt.
        » Which method is the “entry” point to Belt?
Revisiting Why Decorator

• Localized need for extra functionality.
  – Objects are already created by someone else.
  – Assignment 5
    • Consider Belt’s methods:
      – public Sushi getSushiAt(int position)
      – public Sushi takeOffSushiAt(int position)
      – public int placeNearestTo(Sushi sushi, int target_position)
      – public Iterator<Sushi> iterator()
    • Don’t want to change operations on Belt
      – Other parts of system will be creating Sushi and adding it to the Belt.
        » Which method is the “entry” point to Belt?
Assignment 5

• In Assignment 5
  – Sushi is the interface that you will need to decorate.
    • Which classes implement sushi?
    • Why would using subclassing to add expiration functionality be harder?
Decorator for A5 Illustrated

interface Sushi {
...
}

class Nigiri implements Sushi {
...
}
class Sashimi implements Sushi {
...
}
class Roll implements Sushi {
...
}
class Plate implements Sushi {
...
}

interface TimedSushi extends Sushi {

}

class BeltedSushi implements TimedSushi {
private Sushi wrapped_sushi;
public (Sushi s) {
    wrapped_sushi = sushi;
}
// Sushi interface methods delegated
// to wrapped_sushi object.
}
Timed Sushi Functionality

• Illustration doesn’t specify any new functionality for TimedSushi interface.
  – What do we need?
Decorator for A5 Illustrated

interface Sushi {
    void do_something();
}

class Nigiri implements Sushi {
   ...
}

class Sashimi implements Sushi {
   ...
}

class Roll implements Sushi {
   ...
}

class Plate implements Sushi {
   ...
}

interface TimedSushi extends Sushi {
    boolean isExpired();
    int countdown();
}

class BeltedSushi implements TimedSushi {
    private Sushi wrapped_sushi;
    private int count;
    public (Sushi s, int expiration) {
        wrapped_sushi = s;
        count = expiration;
    }
    // Sushi interface methods delegated
    // to wrapped_sushi object.
}
Unwrapping the Decorator

• When you decorate an existing object, you are creating a new object.
  – Existing object is inside.
• Might need to have the ability to “undecorate” the object.
  – For example, if we need to give it back to someone who expects the original.
  – Back to Belt:
    – public Sushi getSushiAt(int position)
    – public Sushi takeOffSushiAt(int position)
    – public int placeNearestTo(Sushi sushi, int target_position)
    – public Iterator<Sushi> iterator()
  – When would we want to unwrap?
Unwrapping the Decorator

• When you decorate an existing object, you are creating a new object.
  – Existing object is inside.

• Might need to have the ability to “undecorate” the object.
  – For example, if we need to give it back to someone who expects the original.
  – Back to Belt:
    – public Sushi getSushiAt(int position)
    – public Sushi takeOffSushiAt(int position)
    – public int placeNearestTo(Sushi sushi, int target_position)
    – public Iterator<Sushi> iterator()
  – When would we want to unwrap?
Undecorating

- Provide method to get back original in decorated interface.

```java
interface I {
    void do_something();
}

class C implements I {
    void do_something() {
        ...
    }
}

interface DI extends I {
    void do_more();
    I getWrapped();
}

class DC implements DI {
    private I wrapped_i;
    public (I i_obj) {
        wrapped_i = i_obj;
    }
    public I getWrapped() {
        wrapped_i;
    }
    void do_something() {
        wrapped_i.do_something();
    }
    void do_more() {
        ...
    }
}
```
A5 and undecorating

• A5 does not ask for Sushi objects to be undecorated on the way out.
  – This will be part of the extra credit TBA.