CS189A
Software Engineering: Concepts and Practices

Jianwen Su
Department of Computer Science
UC Santa Barbara

https://capstone.cs.ucsb.edu/cs189a/cs189a_sched.html
Today’s Lecture

- Progress checking and upcoming deadlines
  - Retrospective for the class

- Design technique
  - 2 weeks ago:
    Requirements engineering → PRDv1
  - Last week: techniques and tools → PRDv2

- Today: drill down—The Decorator Pattern
Overall Course Plan @Week 6

- Four 2-week sprints:
  - Oct 15-29 (PRD v1 – tools, technologies, design, terminology);
  - Oct 29-Nov 12 (use cases/user studies, prototyping, PRD v1);
  - Nov 12-26 (design, prototyping, testing, PRD v2);
  - Nov 26-Dec 10 (prototype demo/pres prep, prototyping and testing)

- Specify what the product will do
  - Vision statement
  - Product Requirements Document (PRD) (due Nov 6 and Dec 4)
  - Design tools, brainstorming, coding (tests and implementation)

- Build and test an initial prototype
  - Typically teams iterate on these activities until they converge to a working prototype!

- Course presentations with demo:
  - Friday, December 11, 3:30-5:30pm, zoom (details to come)
This Week’s Plan

Team activities
- Scrum: Completes sprint 2 & starts Sprint 3
- Retrospective for Sprints 2 (and 1), plan for Spring 3

Section: Scrum, TA meetings:
- Sprint 2 reports/demo (trello board and burndown)
  Sprint 3 starts

Upcoming deadlines:
- Nov 12: Sprint 2 ends (Sprint 3 starts)
  - Product Requirements Document v2
- Nov 16: 2-slide presentation in during lecture:
  - Retrospective for Sprint 2 (1 good, 1 bad, and changes)
  - Plans for Spring 3 & PRDv2
Sprints 3 and 4

- Sprint 2 ends and Sprint 3 starts, this Thursday
- Break down stories into tasks & components associated with design
  - Prioritize stories
  - Assign timings to stories/use cases AND store/use-case tasks
  - Specify acceptance/test that can be used to verify a story is finished

- Sprint: Prototype tasks (primary implementation before demo)
  - Prioritize tasks
  - Assign timings to tasks
  - Specify what test(s) are to be used as evidence of task completion/acceptance (use case post condition OR user story acc test)
  - Each member/developer chooses task, implements, and tests task
  - Another member does code review/test and accepts the pull request
    - Test is the one specified above (Acceptance)
  - When store/case is complete, some member performs story test/acceptance
PRDv2: Your Living Requirements Document: A Shared Google Doc

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance (3+ pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1+ pages) – ie detailed design
- Requirements (functional and non-functional)
  - User stories or use cases (links) → 20+ for PRDv2 prioritized w/acc. tests
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed
Your Project Design: PRDv2

- **Architecture (hardware/software)**
  - Evolve your overview picture from PRDv1 to provide significantly more detail and any updates or changes

- **Detailed design**
  - UML diagrams of primary data structures that comprise the system architecture connected via their associations (if any)
    - Ensure that each "class" is balanced in terms of cohesion & coupling
    - Annotate with pre/post conditions when appropriate
  - Sequence diagrams
    - Synchronous and asynchronous for key interactions between classes
      - At least 3 different interactions
    - User interactions with the system
      - At least 3 different interactions
      - Can be a human user or a machine user (API) interaction
      - Event response, updated application state
      - If you have a user interface: Provide mockups for primary UIs
PRDv2 User Stories / Use Cases

- Revise spec to add detail to the functional specification to match your design
- Add user stories and break up the stories you have into finer grained stories
  - Provide UML, sequence diagrams, dataflow diagrams
  - Goal: a CS senior should be able to take your doc and implement the project
- For each fine-grained story, provide a description and acceptance test
  - Provide time estimates (1 person-hours) for each story implementation
    - Ensure you can finish the implementation in the time you have (this/next quarter)
  - Prioritize tasks to have a complete prototype by the end of this quarter
    - Focus on the externally facing interfaces, mock out what you cannot get to
  - Write unit tests to implement tasks for mandatory tasks
    - Document these tasks (autogen the documentation/usage)
  - Add trello/pivotal task links (titles must match) to PRDv2 for each story
- Prototype designed mandatory tasks; add github commit ID/link to PRDv2
  - Github must have unit tests, documentation (for anything without unit tests), and prototyping implementations for each story in Sprint
- If you have a user interface
  - Provide mockups that are tied to the functionality described in 1+ components
Retrospective (Class)

- **Positives:**
  - Teams-Projects working well, including Brian-Jianwen
  - Mentors contributing

- **Negatives:**
  - Communications could be better
    - Improvements: team meetings and mentor meetings
  - Management not as technical
    - Improvements: next round of team meetings
Today’s Lecture

- Progress checking and upcoming deadlines
  - Retrospective for the class

- Design technique
  - 2 weeks ago:
    Requirements engineering → PRDv1
  - Last week: techniques and tools → PRDv2

- Today: drill down—The Decorator Pattern
An Example Application Scenario

- A coffee shop starts simple:

  + steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk

- Success means adding more varieties:
What’s wrong? A maintenance nightmare!

Examples:
- milk price goes up → a lot of changes
- Add a new topping → too many new combinations to add
Another Solution: Using Fields

- Boolean fields to record condiments
- `cost()` no longer abstract
- Subclasses need to use superclass’ `cost()` to calculate their costs
- A reasonable solution but …
  - Price change?
  - Add new condiments?
  - How about double Mocha?
  - New beverages?

- The Open-Closed Principle: Classes should be open for extension but closed for modification

```
Beverage
- description
- milk
- soy
- mocha
- whip
+ getDescription()
+ cost()
+ hasMilk()
+ setMilk()
+ hasSoy()
+ setSoy()
+ hasMocha()
+ setMocha()
+ hasWhip()
+ setWhip()
```

```
HouseBlend
+ cost()

DarkRoast
+ cost()

Decaf
+ cost()

Espresso
+ cost()
```
Key Idea Of The Decorator Pattern

- Start with a beverage object, and “decorate” it with condiments at runtime

- Example:
  - Create a **DarkRoast** object
  - Decorate it with a **Mocha** object
  - Decorate it with a **Whip** object
  - Call the `cost()` method and add the condiment costs

```
Whip: 1.29
Mocha: 1.19
DarkRoast: 0.99
```
The Decorator Pattern

- Allows to attach additional responsibilities to an object dynamically
- **Component**: basic elements
- **Decorator**: additional behaviors as flexible alternative

- **ConcreteComponent**: actual objects to be added new behaviors
- **ConcreteDecorator**: actual behaviors
- Multiple **ConcreteComponent** and **ConcreteDecorator** classes are possible
In our example, component is **Beverage** and decorator is **CondimentDecorator**.

```java
public abstract class Beverage {
    private String description = "Unknown Beverage";
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
    protected Beverage beverage;
    public abstract String getDescription();
}
```

**Diagram:**
- **Beverage**
  - + cost()
  - ...
- **CondimentDecorator**
  - beverage: Beverage
  - + cost()
  - ...

Requires that all condiment decorators reimplement the `getDescription` method.
Implement Beverages

- Two implementations in our example:

```java
public class Espresso extends Beverage {
    public Espresso() {
        description = "Espresso";
    }
    public double cost() {
        return 1.99;
    }
}
```

```java
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }
    public double cost() {
        return .89;
    }
}
```

- The prices of the beverages are defined here, only once!
  – Easy for future price changes
Implement Condiments

- Mocha as an example condiment, others are similar

```java
public class Mocha extends CondimentDecorator {
    public Mocha (Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.getDescription() + ", Mocha"
        // beverage inherited from CondimentDecorator
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
```

- Note:
  - `getDescription()` appends ", Mocha" for the condiment
  - `cost()` adds .20 to reflect the cost of the condiment
public class StarbuzzCoffee {
    public static void main( String args[] ) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription() + " $" + beverage.cost());
        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " $" + beverage2.cost());
        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription() + " $" + beverage3.cost());
    }
}

Expresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
What The Decorator Pattern Gives Us?

- One class per type of bare beverages
  - One class per type of condiments
    - One description value for each of them
    - One price value for each of them
- Each for future price/description changes
- For each actual order, there may be multiple objects:
  - One for the bare beverage
  - One for each condiment
- Clean conceptual modeling
Benefits

- Useful for applications when you want the capabilities of inheritance with subclasses, but you need to add functionality at runtime
- Allows to modify an object dynamically
- More flexible than inheritance
- Simplifies code
- Adding new code rather than modifying the old
- Not much redundant data