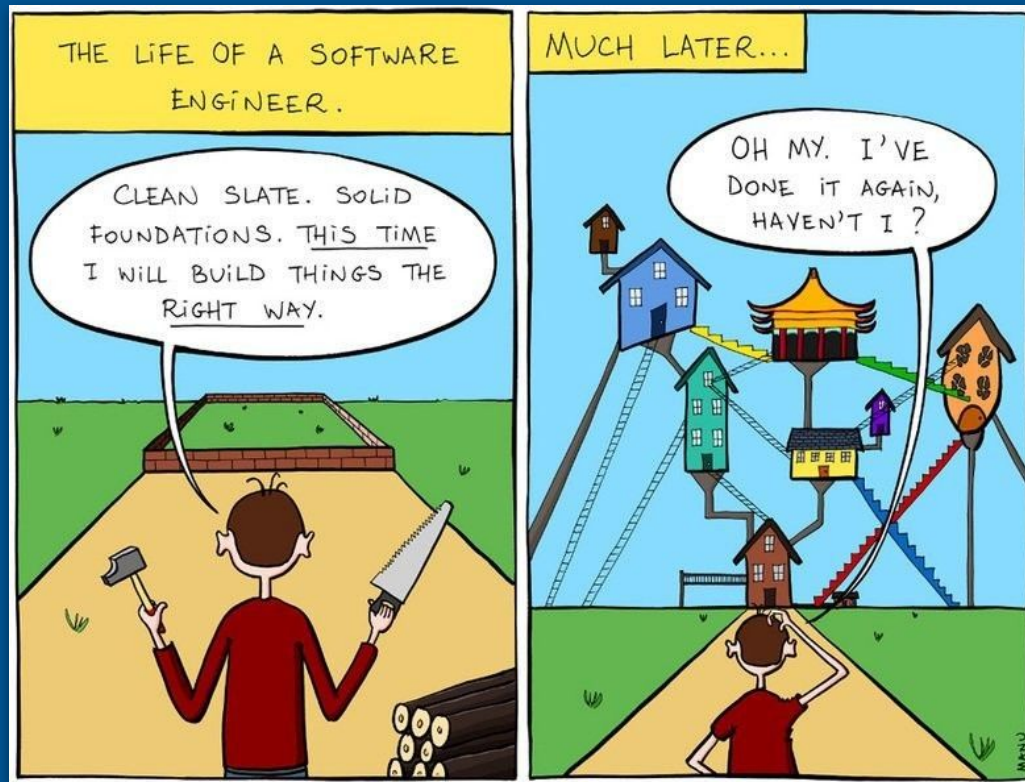


For personal use only,  
please do not  
distribute on  
non-UBC domains.

# 05b-Patterns II

## CPSC310

Nick Bradley & Kyle Chin



# Project

Reminder: D2 due next week!

Team formation for D3/D4 opens on Friday

- Instructions will be posted on Piazza
- Talk to your TA in lab if you need a partner
- Team repos will start being provisioned on Oct 19th
  - You should have your team entered by end of day

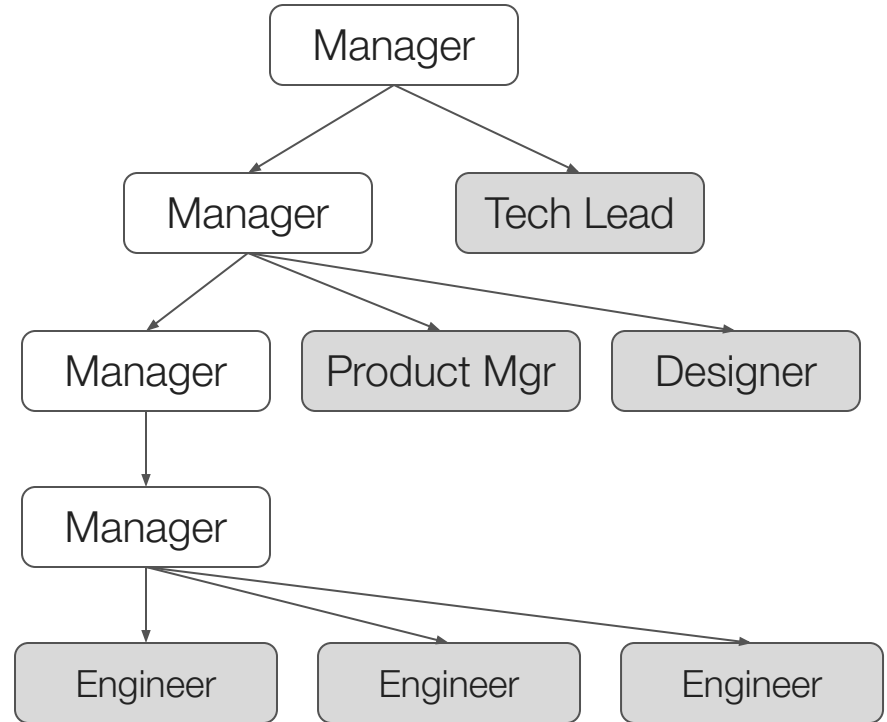
# Learning goals

By the end of this lecture, you should be able to:

- Identify a decision in existing code that is likely to change, and where knowledge of it is scattered or tangled.
- Explain a design pattern in terms of the change, the pain, the mechanism and the cost.
- Refactor unfamiliar code to a pattern in small steps that keep its behaviour the same (today: Composite and Decorator).
- Evaluate whether a pattern is worth its cost in a particular situation.

# Scenario

Workday has hired you to work on their organization management software! An org contains managers who supervise managers, engineers, designers, technical leads, and product managers.



# Composite Design Pattern

**Intent:** An application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to know the "type" of each object before attempting to process it is not desirable.

Component

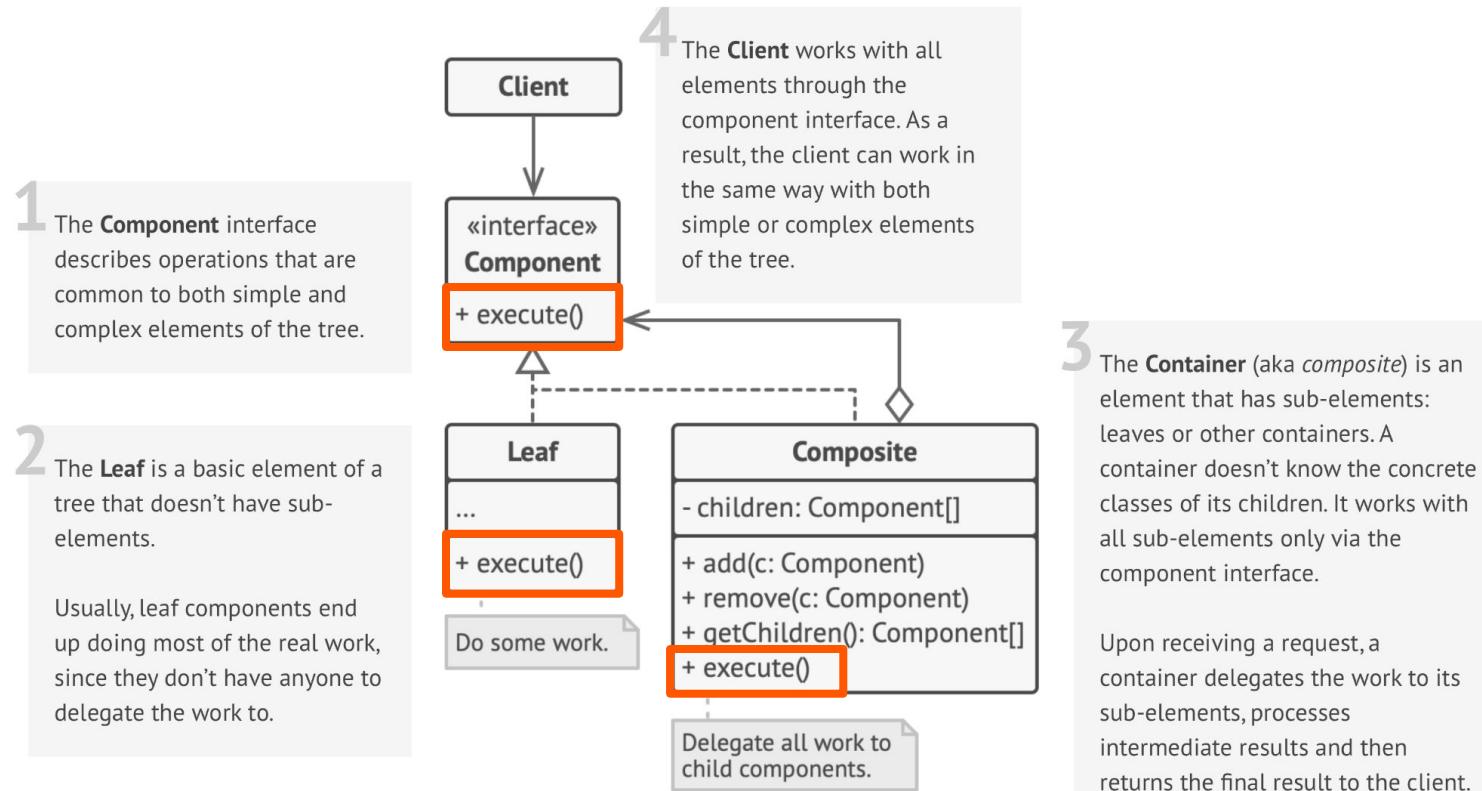
Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Composites contain entries, each of which could be a composite.

Composite

Leaf



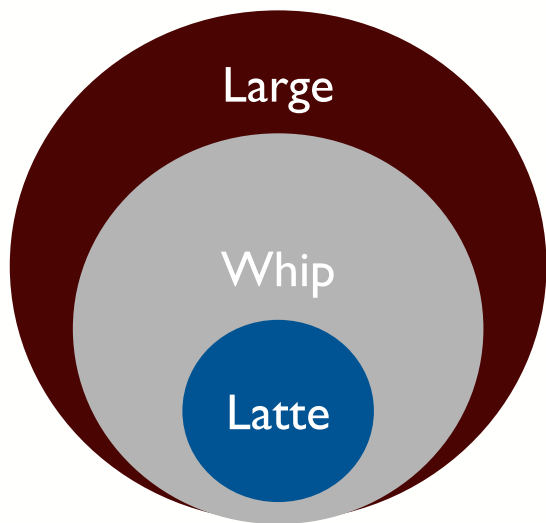
# Composite: Canonical Depiction



# Composite: Analysis

1. **Change:** new roles in the org, any of which might be one person or head a team of their own
2. **Pain:** one array and loop per role in every Manager operation (scattered); each operation mixed with checking whether a report is one person or a team (tangled)
3. **Mechanism:**
  - a. **Abstraction:** Employee interface  $\rightarrow$  getTotalSalary(), getHeadcount(), traverse()
  - b. **Oblivious caller:** Manager only knows Employee[]; each employee must answer for itself and everyone under it
  - c. **Trigger:** client calls a method on any node; each Manager calls it on each of its reports
4. **Cost:** one class per role; a new operation means a new method in the interface and in every class

# Scenario



You're working on a cash register app for a coffee cart. It sells Espresso, Latte, Cold Brew, and Matcha Latte. Any drink can have add-ons (an extra shot, whip, or a large size), including **more than one** of the same add-on.

# Decorator Design Pattern

**Intent:** You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

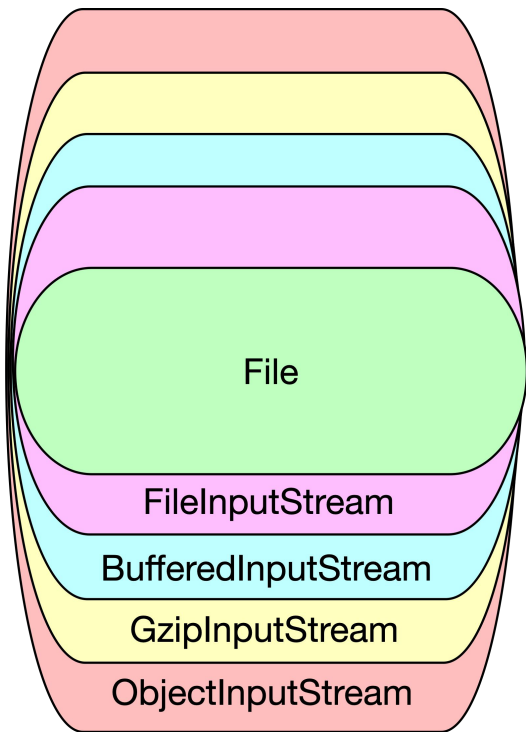
Each wrapper changes the behaviour...



...but it still looks like the original

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. Wrapping a gift, putting it in a box, and wrapping the box.

# Example in Wild: Java.io

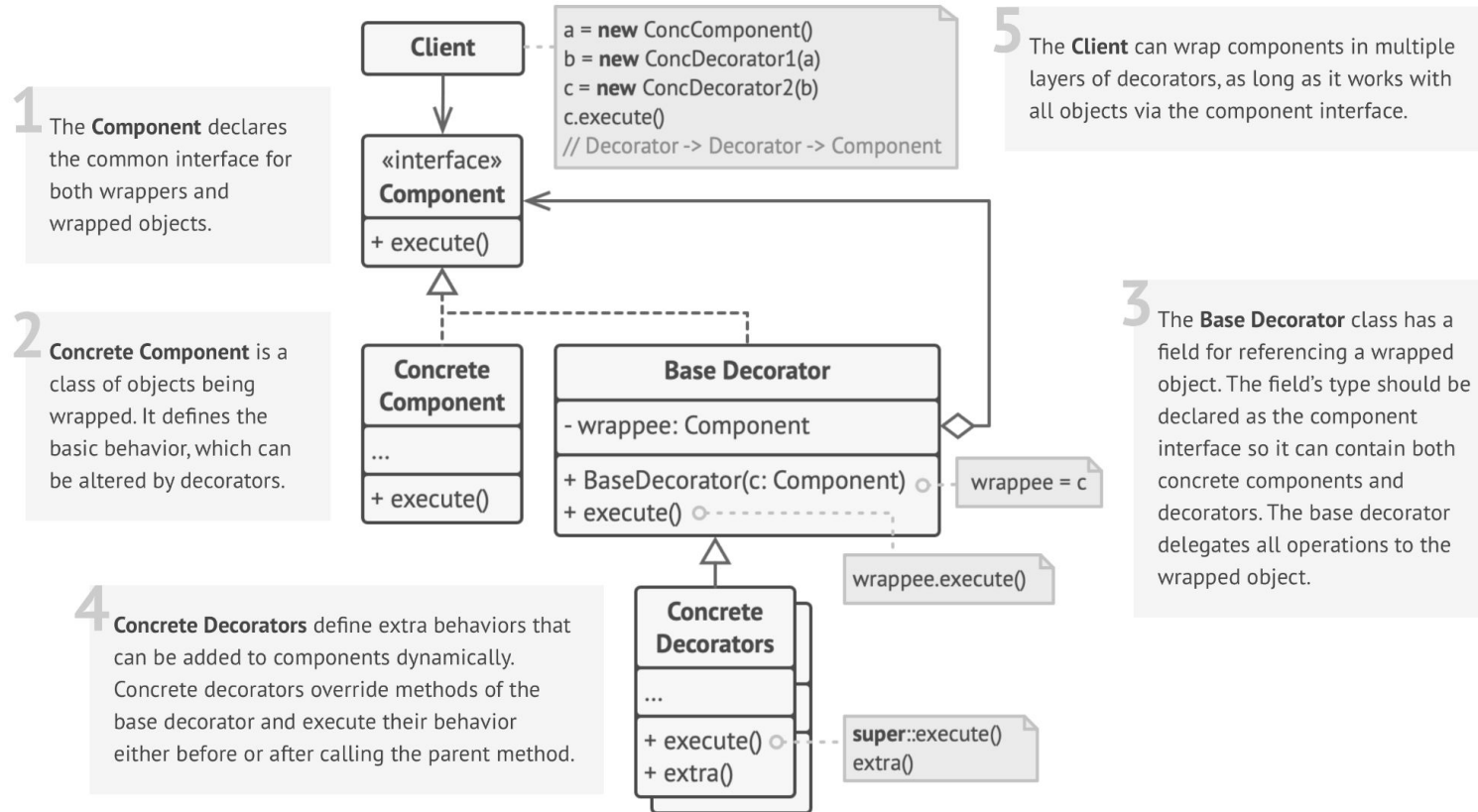


```
new ObjectInputStream(  
    new GzipInputStream(  
        new BufferedInputStream(  
            new FileInputStream(  
                new File('fname.gz')  
            ))))  
));
```

**Decorators**

**Base Object**

# Decorator: Canonical Depiction



# Decorator: Analysis

**Change:** new add-ons, each changing a drink's price and description in its own way

**Pain:** one case per add-on in every switch (scattered); CoffeeCart mixes tracking the order with every add-on's pricing and naming rules (tangled)

## Mechanism

- **Abstraction:** Drink interface → getDescription(), getPriceCents(); AddOn wraps one Drink
- **Oblivious caller:** CoffeeCart only knows Drink[]; each add-on must answer by changing the answer of the drink it wraps
- **Trigger:** the cart asks the outermost add-on, which asks the drink it wraps, all the way down to the menu drink

**Cost:** one class per add-on; wrapping order changes the result; long chains of wrappers are harder to read and debug