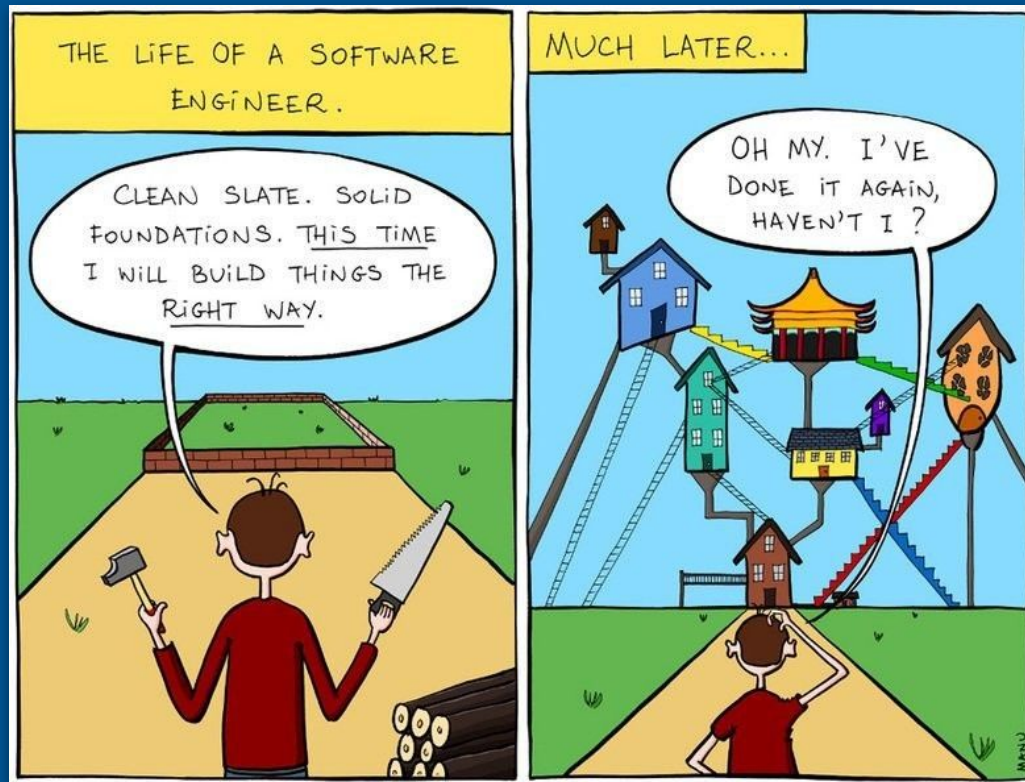


For personal use only,
please do not
distribute on
non-UBC domains.

05a -Patterns I

CPSC310

Nick Bradley & Kyle Chin



Module 1: Synthesis

Design in the ***small***: low-level software analysis & improvement

- Cost of change
- Sources of cost (coupling, cohesion, testability)
- Addressing cost (refactoring)

Module 1: Synthesis

Design in the ***small***: low-level software analysis & improvement

- Cost of change
- Sources of cost (coupling, cohesion, testability)
- Addressing cost (refactoring)

Module 2: Starting Now

Design in the ***medium***: higher-level software design & analysis

- Improving designs of software
- Tradeoffs in architectures
- External implications of designs

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: Adapter and Composite).
- Evaluate whether a pattern is worth its cost in a particular situation.

Designing for Change

Every system you build will be changed:

- New requirements: "Can we add a QA role to the org chart?"
- New dependencies: "We're switching payment providers."
- New scale: more users, more developers, more code

We can't predict what the change will be, but we can often predict where it will land.

When the change arrives, how many places do you have to edit, and how much unrelated code do you have to touch?

What is a Design Pattern?

A tried and true solution to a commonly encountered problem.

A design pattern is not:

- A class diagram you copy
- A framework or library
- A rule you must always follow

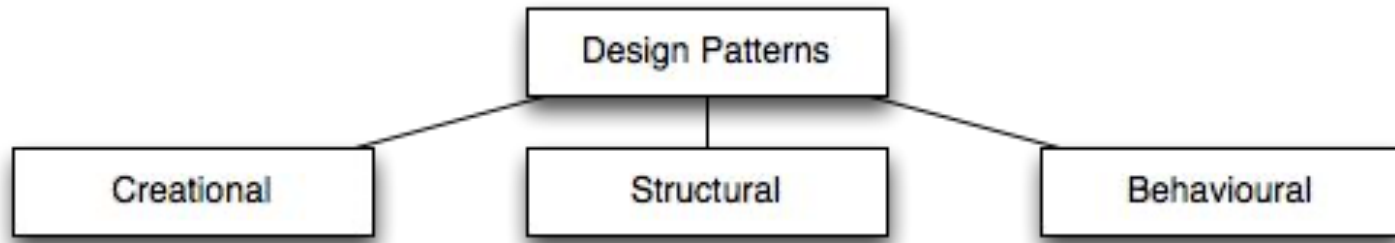
A design pattern is:

- A *reusable way of thinking* about a recurring design problem
- What emerges when you keep a likely change local
- A vocabulary for discussing trade-offs

The Value of a Name

- **Recognition:** you spot the problem shape earlier, sometimes before writing any code, instead of only after the pain shows up.
- **A destination:** refactoring is how you get there. Knowing the pattern tells you where you're heading, so the refactorings have a direction.
- **Known costs:** the pattern comes with known trade-offs, including when it isn't worth the indirection.
- **Shared vocabulary:** one word stands in for a paragraph in a design discussion or code review.

OO Design pattern categories



“ Creational design patterns are design patterns that deal with **object creation mechanisms**, trying to create objects in a manner suitable to the situation.

“ Structural design patterns are design patterns that ease the design by identifying a simple way to **realize relationships between entities**.

“ Behavioural design patterns are design patterns that identify **common communication patterns between objects** and realize these patterns.



There are LOTS of OO
design patterns!

Categories are useful for organization—not for
deciding when to use a pattern

How to understand a design pattern

- 1) **Change**: which decision is likely to vary?
- 2) **Pain**: where does that decision scatter (a coupling problem) or tangle (a cohesion problem) in the current code?
- 3) **Mechanism**: how does the pattern give that decision one home?
- 4) **Cost**: what indirection did we add, and was it worth it?

Focus on OO patterns in 310

OO Design Patterns help developers navigate object oriented solutions using:

Abstraction (OO view)

- Separating *what varies* from *what stays the same*
- Captured using **interfaces or abstract classes**
- Defines *roles*, not concrete behaviour

Polymorphism (OO view)

- Treating different concrete objects *uniformly* through abstraction
- Behaviour varies via **overriding and dynamic dispatch**
- Enables code to be **oblivious to concrete types**



Finding the Mechanism

1. **Abstraction:** what new interface or class gives the decision a single home?
2. **Polymorphism/Obliviousness:** what methods does that abstraction declare, and who calls them without knowing the concrete type? Can someone add a new implementation and override them safely?
3. **Trigger:** when are those methods called, and by whom?

Why / Why Not Design Patterns?

Easy to blindly ‘force’ a pattern into a context where it does not make sense.



Patterns help when:

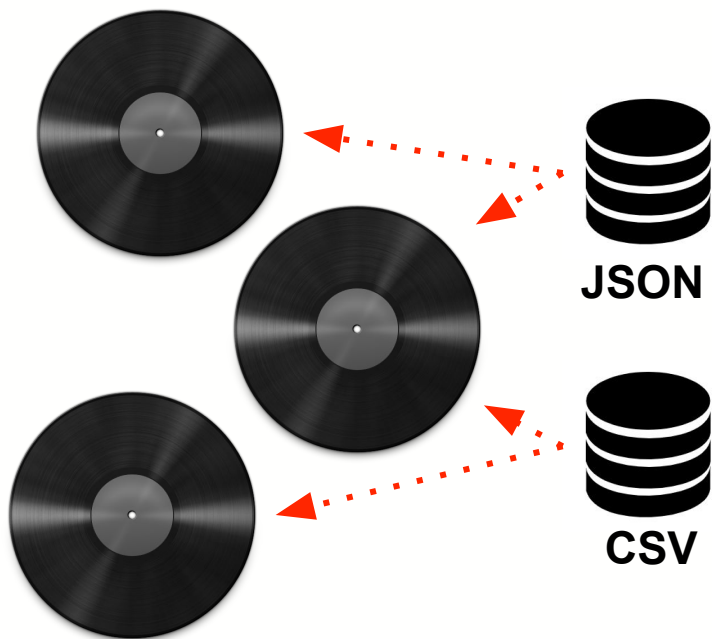
- Change is likely
- Variability already exists
- Coupling is becoming painful



Patterns hurt when:

- Requirements are stable
- There's only one concrete case
- The abstraction cost outweighs future benefit

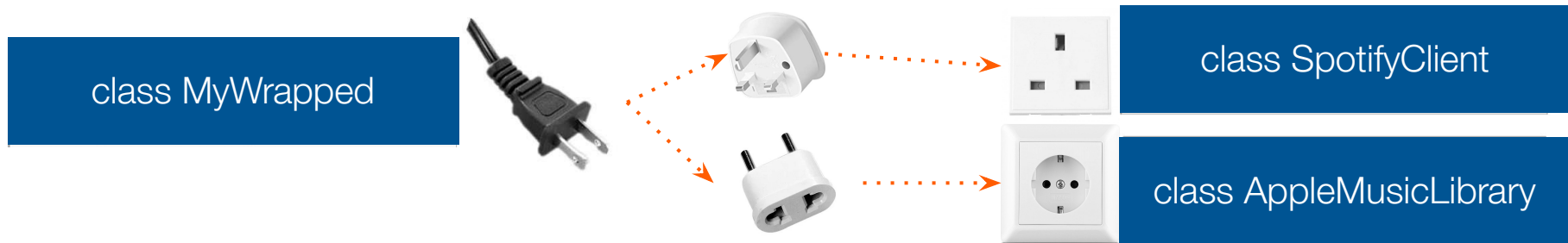
Scenario



You're building an app that generates a year-end "wrapped" summary of the songs you played across several music platforms. Each platform has its own interface, with its own methods and its own idea of what a play looks like.

Adapter Design Pattern

Intent: An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.



Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. **Wrap an existing class with a new interface.**

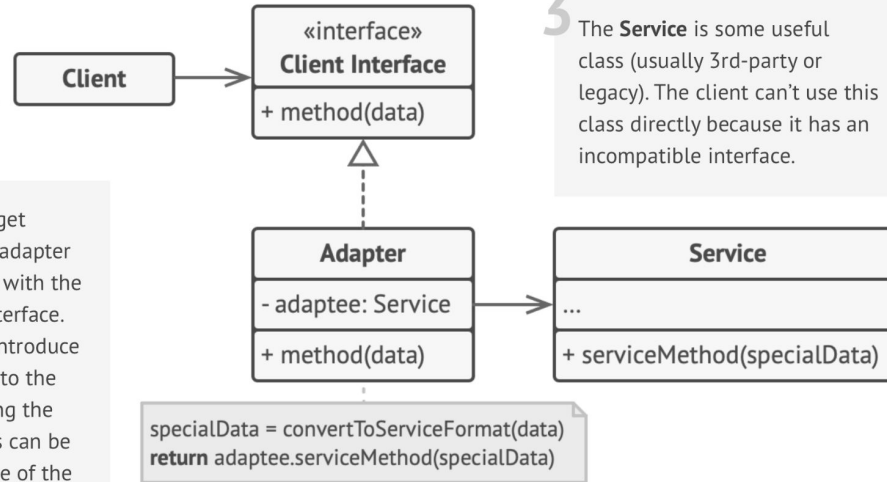
Adapter: Canonical Depiction

1 The **Client** is a class that contains the existing business logic of the program.

2 The **Client Interface** describes a protocol that other classes must follow to be able to collaborate with the client code.

3 The **Service** is some useful class (usually 3rd-party or legacy). The client can't use this class directly because it has an incompatible interface.

5 The client code doesn't get coupled to the concrete adapter class as long as it works with the adapter via the client interface. Thanks to this, you can introduce new types of adapters into the program without breaking the existing client code. This can be useful when the interface of the service class gets changed or replaced: you can just create a new adapter class without changing the client code.



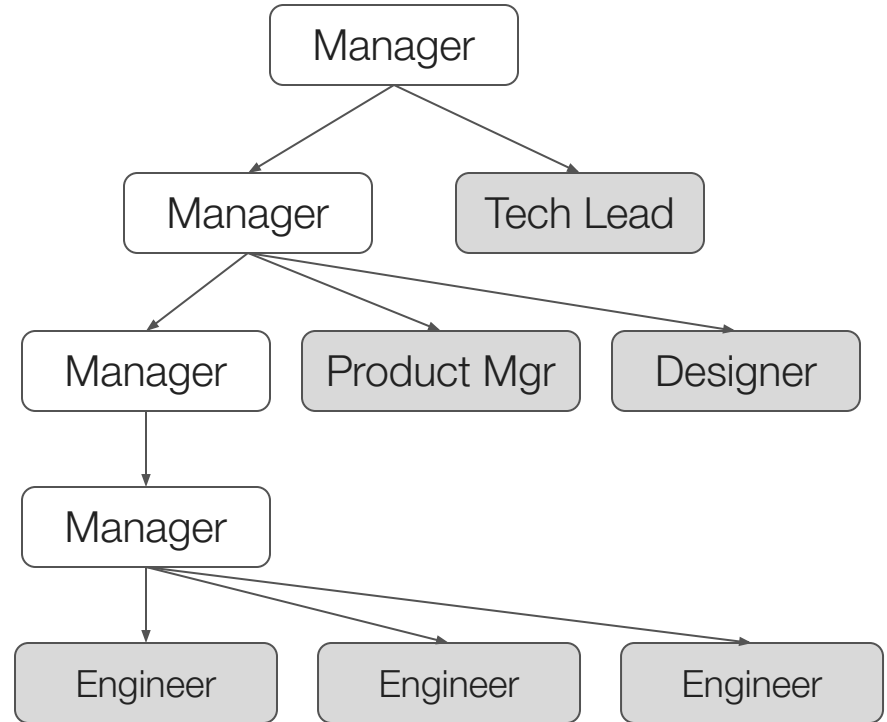
4 The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

Adapter: Analysis

1. **Change:** new platforms, each with its own interface
2. **Pain:** one loop per platform in every report (scattered); report logic mixed with calls to each platform's interface and field translation (tangled)
3. **Mechanism**
 - a. **Abstraction:** `MusicSource` interface $\rightarrow$ `getRecords(): MusicRecord[]`
 - b. **Oblivious caller:** `MyWrapped` only knows `MusicSource[]`; each adapter must return normalized records
 - c. **Trigger:** each report calls `getRecords()` on every adapter
4. **Cost:** one class per platform; `MusicRecord` only has fields every platform can supply

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

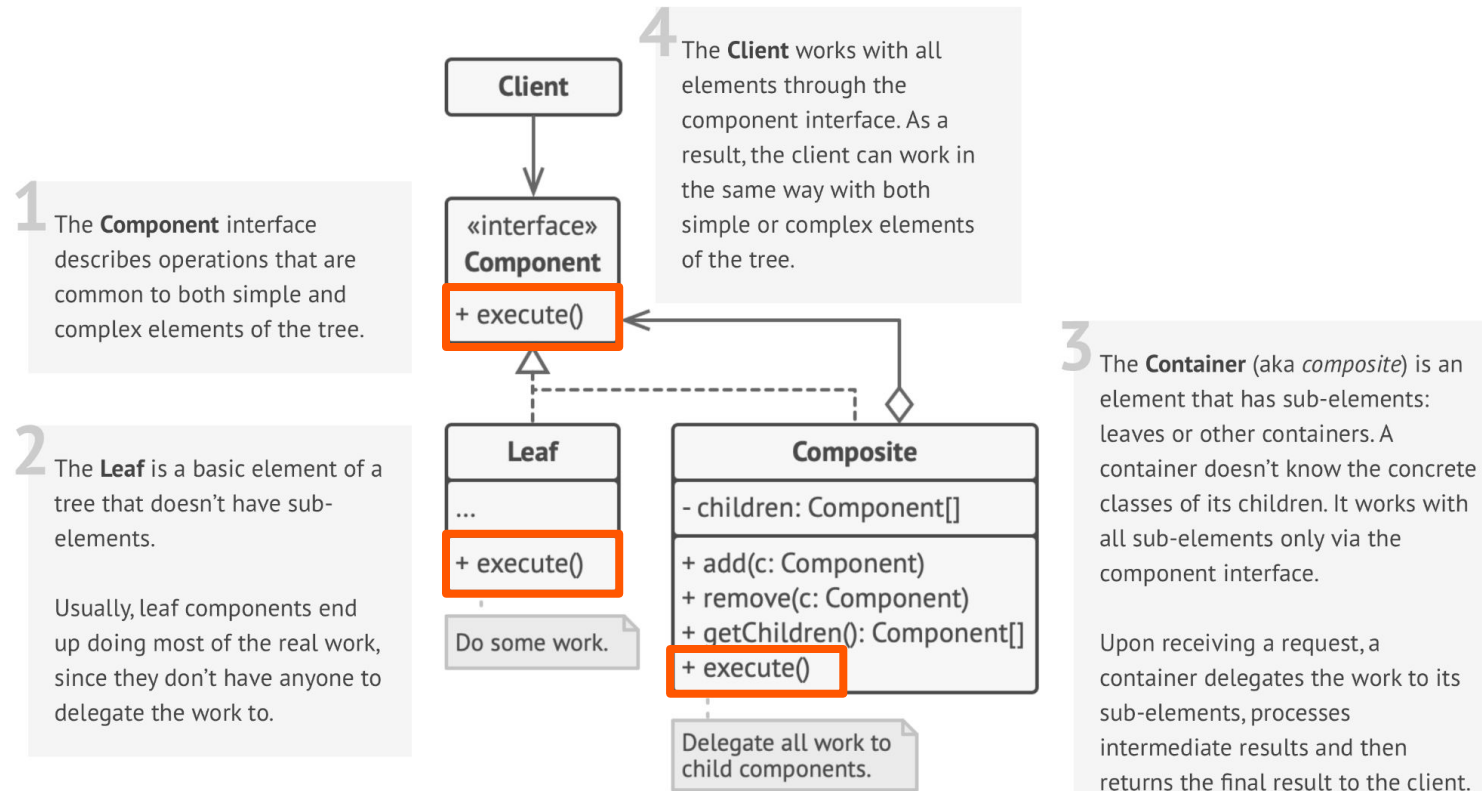
Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Composites contain entries, each of which could be a composite.

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