Design Patterns Overview

What Design Patterns are not

- They are not one size fits all solutions to a particular problem
- They do not solve the particular details of your particular problem
- They are not blocks that, when composed exactly right, can solve any and all software problems

What Design Patterns are

- They are reusable blueprints for software development.
- They are good places to start when solving a well known software interaction
- They are a good structure for thinking about how to develop a large software system
- They are filled with tradeoffs and answers that involve “it depends”

Patterns in the world around us

Why are design patterns so important?

- Useful Abstractions
  - Coding to an interface allows people to use a system without having to worry about the details, increasing reuse, decreasing development time, and allowing teams to work independently
- Maximize Reuse
  - Write **modular** code that can be used in many different places, saves on re-writing or copy-pasting code
- Minimizing Maintenance
  - Reused code creates a single reference, any code changes or bug fixes will then propagate to all references
- Maximize Extensibility
  - Create code that is easy to add new features to without changing what has already been made

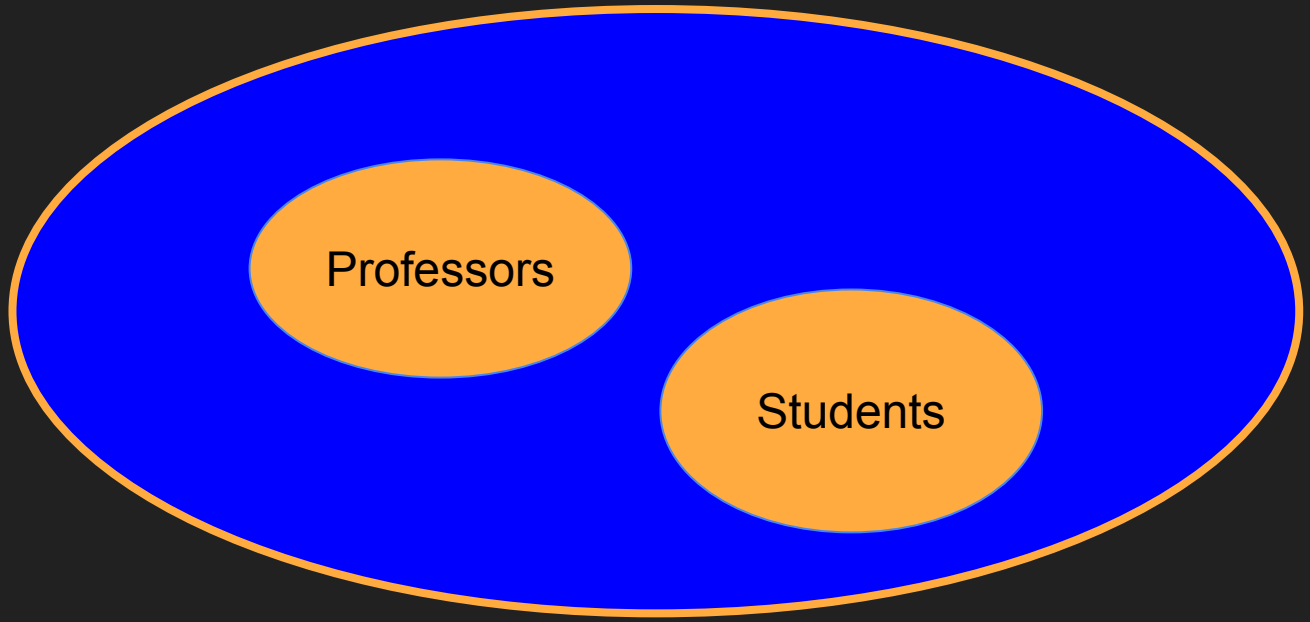
What do I need to implement a design pattern?

- Encapsulation - Objects encapsulate, or contain, data and procedures
- Inheritance/Interfacing - Multiple objects can be used and referenced interchangeably
  - Polymorphism - Dynamic binding of requests to objects based on run-time type
- Composition - Objects are composed, or formed together into a single object, to create new functionality

“Program to an interface, not an implementation”

-- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Class Inheritance - Type - Subtype

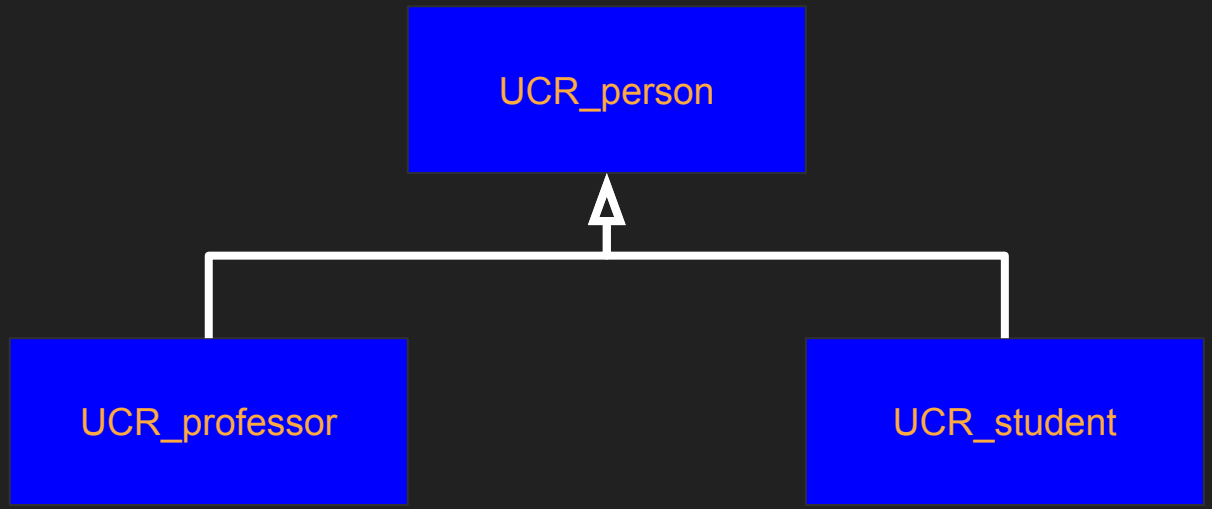


People at UCR

Type hierarchy

UCR_person contains:

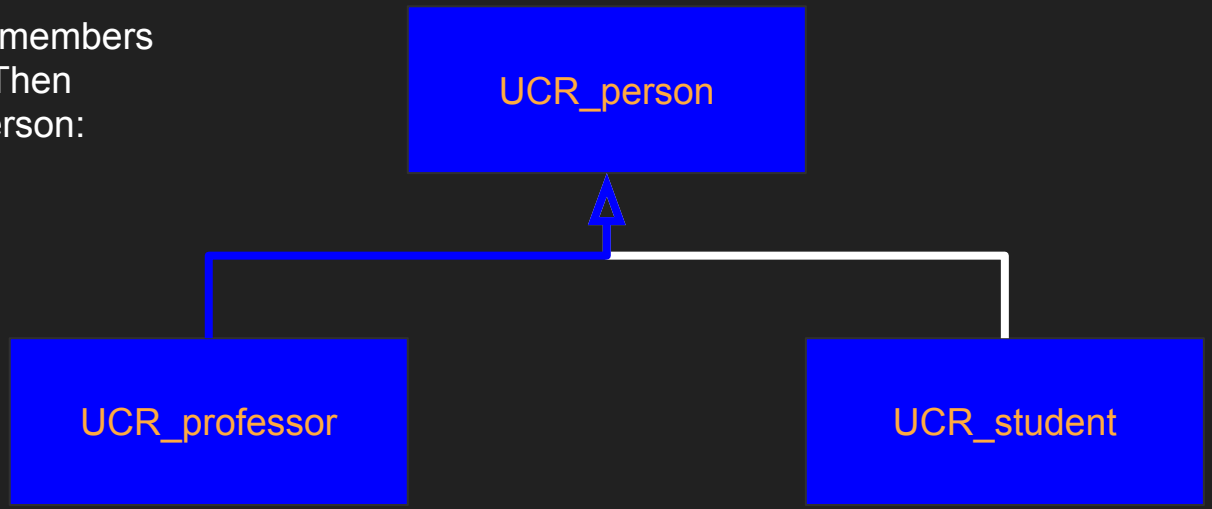
- Common data members
  - Name
  - ID
  - Address
- Common methods
  - Change_address
  - display



Type hierarchy

UCR_professor **inherits** the data members and methods from UCR_person. Then UCR_professor **extends** UCR_person:

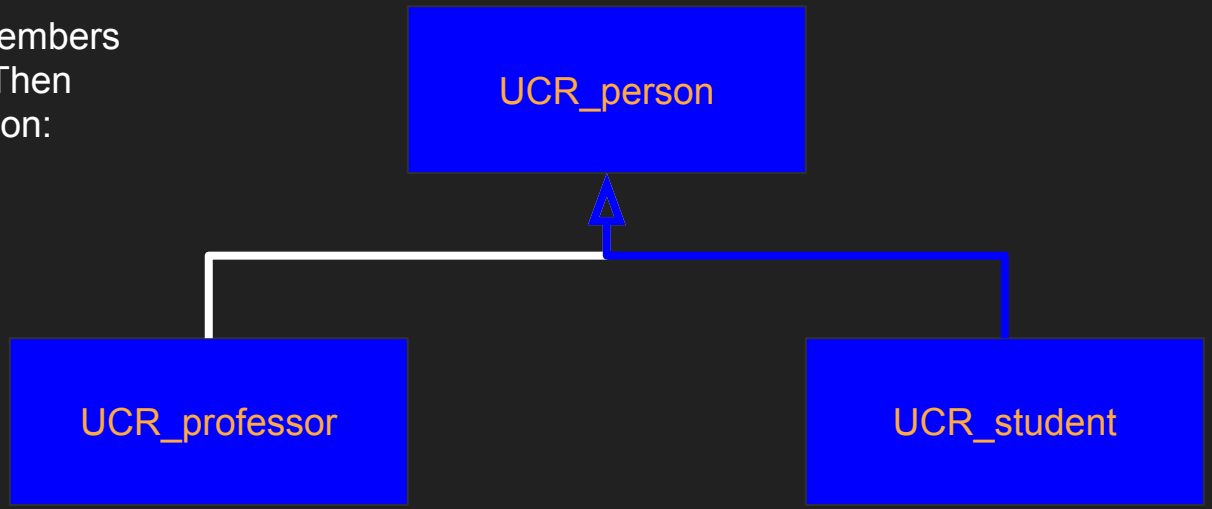
- Specific data members
  - Salary
  - Rank
- Specific methods
  - add_class
  - change_rank



Type hierarchy

UCR_student **inherits** the data members and methods from UCR_person. Then UCR_student **extends** UCR_person:

- Specific data members
  - major
  - Classes taken
  - year
- Specific methods
  - change_major
  - change_course



C++ Access Control

- Public:
  - Accessible by anyone
- Private:
  - Accessible only inside the class, **NOT** by subclasses
- Protected:
  - Accessible inside the class and from all subclasses

Polymorphism

```
Class UCR_person {
  private:
    string name;
    int ID;
  public:
    UCR_person(int,string);
    virtual void display() {
      cout << name << “\n” << ID << “\n”
           << address << endl;
    }
};
```

```
Class UCR_professor : public UCR_person {
  private:
    int salary;
    string rank;
  public:
    void display() {
      UCR_person::display();
      cout << rank << “\n” << salary << endl;
    }
};
```

```
UCR_person* jeff =
    new UCR_professor(1,”Jeff McDaniel”,);
jeff->display();
```

Declared type: UCR_person
Actual (resolved) type: UCR_professor

Abstract Base Class

When it doesn't make sense to have an object of that class

In C++ use **pure virtual functions**; virtual functions without implementation in the base class

```
class UCR_person {
    public:
     virtual void display() = 0;
}
```

Indicates that display is **pure** (no implementation)

Display is declared **virtual**

UCR_person can no longer be instantiated

Inheritance vs. Object Composition

- Inheritance is considered **white-box reuse**
  - Internals of parent class are visible to children
  - Redesign of parent causes redesign of children classes
  - Statically determined at compile time
- Object composition is **black-box reuse**
  - Internals of object in the composition are hidden from other objects
  - Redesign of internals of an object don’t require redesign of other objects
  - Redesign of interface of an object requires redesign of other objects
  - Dynamically determined at run-time

What is a design pattern?

A **design pattern** systematically names, explains, and evaluates an important and recurring design in object-oriented systems

Design patterns help a designer get a design “right” faster.

Structure of design patterns (Example)

- Pattern Name
- Problem

- Solution

- Consequences

- Composite
- Code treats primitive and container objects differently, even if the user treats them similarly
- An abstract class the represents *both* primitives and containers with the same interface
- Consequences
  - Makes the client simple
  - Defines class hierarchies
  - Makes adding new components easy

Design Pattern Classification

- Purpose
  - Creational
    - Concerned with object creation
  - Structural
    - Composition of classes or objects
  - Behavioral
    - Characterize interactions of classes or objects
- Scope
  - Class
    - Pattern applies primarily to classes
  - Object
    - Pattern applies primarily to objects

Design Pattern Classification

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• **Abstract Factory** • Builder • Prototype • Singleton	• Adapter (object) • Bridge • **Composite** • **Decorator** • Facade • Flyweight • Proxy	• Chain of Responsibility • **Command** • **Iterator** • Mediator • Memento • Observer • State • **Strategy** • **Visitor**

How to select a design pattern

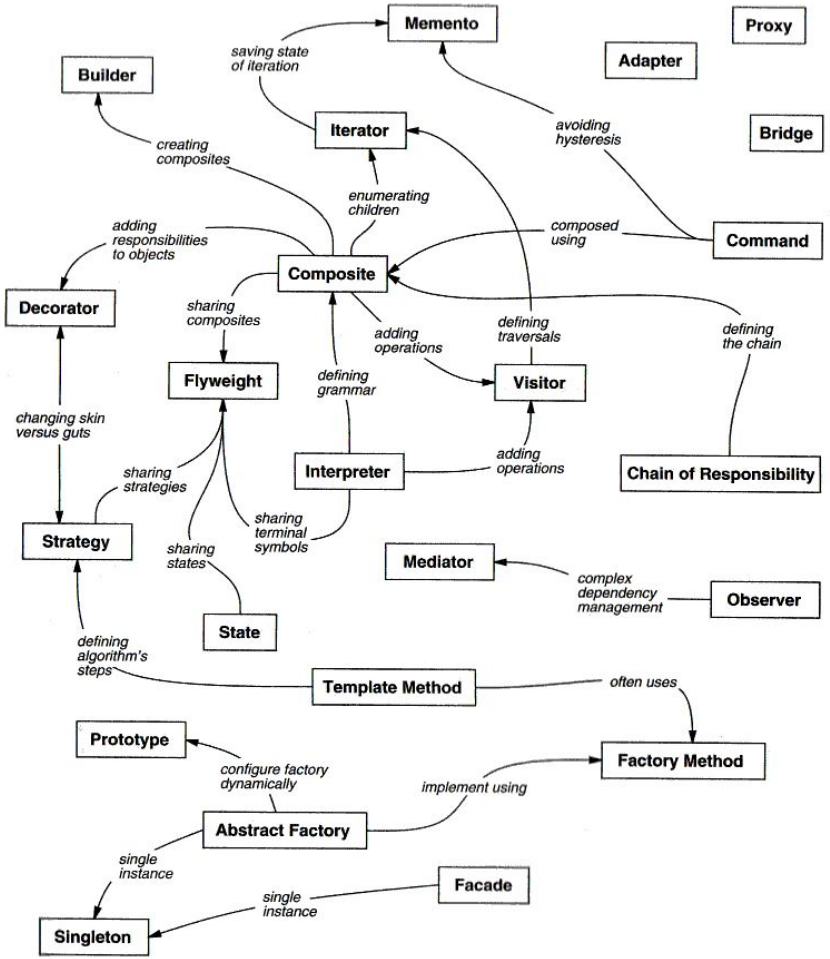


Figure 1.1: Design pattern relationships

How to select a design pattern?

- *Consider how design patterns solve design problems*
- *Review the **intent** of the design pattern*
- *Review how the patterns relate to one another*
  - *Select a pattern or a group of patterns*
- *Review similar patterns*
- *What might cause you to redesign your system?*
- *What do you want to be easy to change in your system?*

Course Overview

Creational Patterns

Structural Patterns

Behavioral Patterns

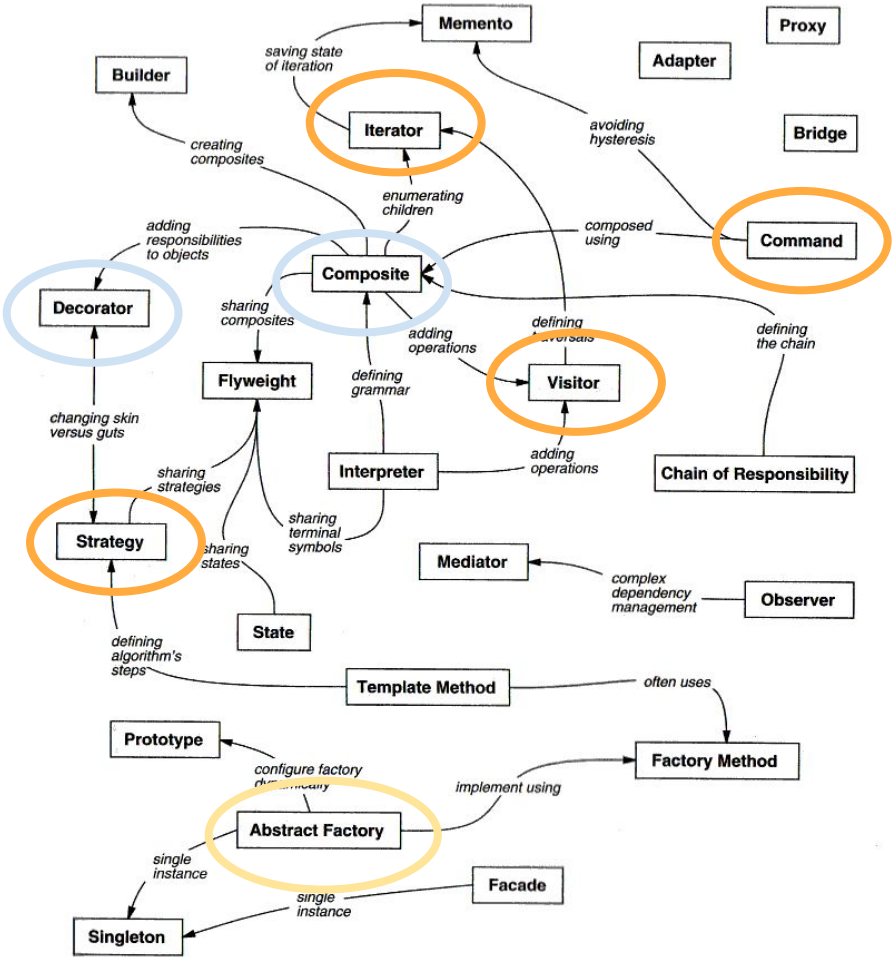


Figure 1.1: Design pattern relationships