[CS3704] Software Engineering

Dr. Chris Brown
Virginia Tech
10/23/2023

Announcements

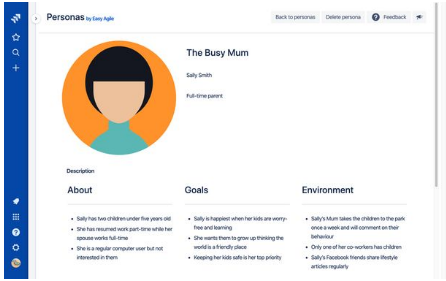
- **PM3 out today (due 11/10)**
- **Discussion Presentation on Design Friday (10/27) in class**
- **Workshop change**
    - **10/20: [Workshop] Course Review**
        - Overview of course content so far
        - Assess knowledge for exam
    - **10/23 (today): Low-level design**

User stories

- Scenarios with explicit acceptance criteria
- Try not to make assumptions
- Definitive, implementable, and specific
- User story is determined to be finished when it meets acceptance criteria.

Writing a User Story

- Written from the point of view of the end user
  - Often on index cards
- Epics: larger user stories
- User personas to describe interactions



- *"As a* `[persona type]`*, I want to* `[action]` *so that* `[benefit]`*."*

Running Example

From the Course Project Ideas List

- **Standup Bot:** A software bot to automatically schedule standup meetings between teammates.
    - *Scrum master*: Professional to ensure scrum processes for development team

Not **a story about a user using your system, example review comment by a user, etc.…**

Example: Stand-Up Bot

As a *Scrum Master*, I want to *find availability with a list of usernames* so that I can *schedule team daily stand-ups*.

<u>Acceptance criteria:</u>

Given Scrum Master has admin permissions
When Scrum Master types team member usernames
And <User can perform multiple actions here>
Then System finds list of available times between users

Low-Level Design

Design Concepts
Object-Oriented Design
Design Patterns

Learning Outcomes

By the end of the course, students should be able to:

- **Understand software engineering processes, methods, and tools used in the software development life cycle (SDLC)**
- Use techniques and processes to create and analyze requirements for an application
- **Use techniques and processes to design a software system**
- Identify processes, methods, and tools related to phases of the SDLC
- Explain the differences between software engineering processes
- Discuss research questions and current topics related to software engineering
- Create and communicate about the requirements and design of a software application

Design

Goal: decide the structure of the software and the hardware configurations that support it.

  – The *how* of the project

- How individual classes and software components work together in the software system.
  – Programs can have 1000s of classes/methods
- *Software Artifacts:* design documents, class diagrams (i.e. UML)

Design Engineering

- The process of making decisions about HOW to implement software solutions to meet requirements.
- Encompasses the set of concepts, principles, and practices that lead to the development of high-quality systems.

Design Practices

- **High-level: Architecture design**
  - Define major components and their relationship
- **Low-level: Detailed design**
  - Decide classes, interfaces, and implementation algorithms for each component

Low-Level Design

- **Goal:** To decompose subsystems into modules
- Two approaches:

1. Procedural:

- system is decomposed into functional modules which accept input data and transform it to output data
- achieves mostly procedural abstractions

2. Object-oriented

- system is decomposed into a set of communicating objects
- achieves both procedural + data abstractions

Software Design Concepts

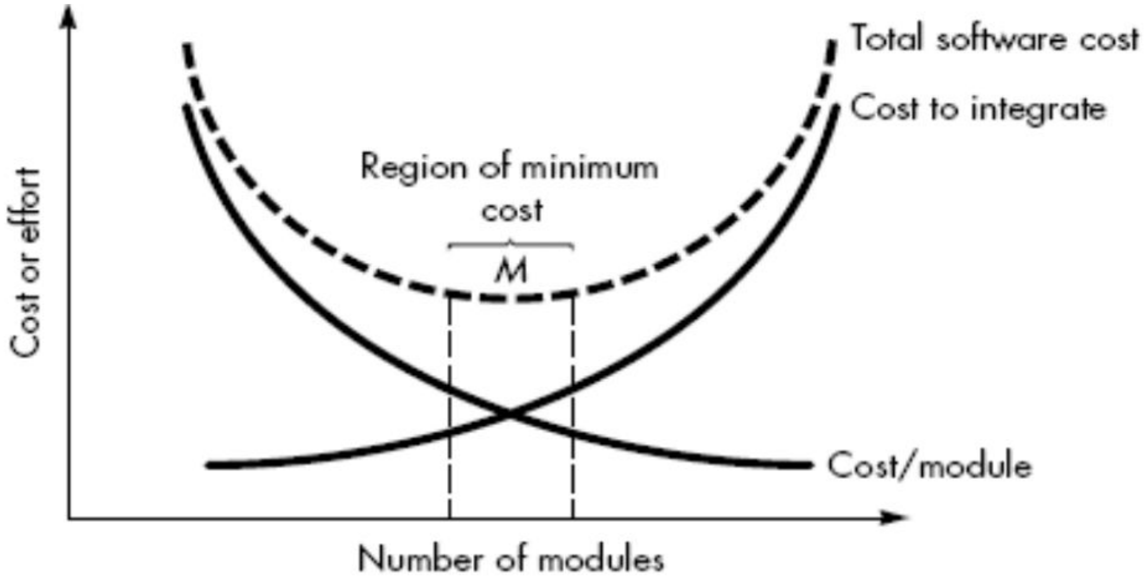
- Modularity
- Cohesion & Coupling
- Information Hiding
- Abstraction & Refinement
- Refactoring

Modularity

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.
    - Divide and conquer

Modularity and Software Cost

Very difficult to get the modules in software design right!



Cohesion and Coupling

- Cohesion
  – The degree to which the elements of a module belong together
  – A cohesive module performs a single task requiring little interaction with other modules
- Coupling
  – The degree of interdependence between modules

- **High cohesion and low coupling!**

Information Hiding

- Do not expose internal information of a module unless necessary
  - E.g., private fields, getter & setter methods
  - Object-oriented design (i.e. Java)

Abstraction and Refinement

- **Abstraction:** Focus on important, inherent properties while suppressing unnecessary details to manage complexity and anticipate changes.
    - *Abstraction Reduces Complexity!*
        - Easier to read, maintain code
    - *Abstraction Anticipates Changes!*
        - i.e. polymorphism, ability to create new types, objects, solutions
- **Refinement:** A top-down strategy to reveal low-level details from high-level abstraction as design progresses.
    - Classes & objects
    - Algorithms
    - Data

Refactoring

"...the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure" – Martin Fowler

- **Goal:** to make software easier to integrate, test, and maintain.
- *More on this with Maintenance lecture (11/8)!*

S.O.L.I.D. Principles of OOD

- **S** – Single-responsibility principle
- **O** – Open-closed principle
- **L** – Liskov substitution principle
- **I** – Interface segregation principle
- **D** – Dependency Inversion Principle

S.O.L.I.D. Running Example

```
class Circle {
    public float radius;

    public Circle(float radius) {
        this.radius = radius;
    }
}
class Square {
    public float length;

    public Square(float length) {
        this.length = length;
    }
}
```

Single-responsibility Principle

- A class should have only one job.
  - Modularity, high cohesion, low coupling
- Ex) Sum up the areas for a list of shapes

```
class AreaCalculator {
    protected List<Object> shapes;
    public AreaCalculator (List<Object> shapes) {
        this.shapes = shapes;
    }
    public float sumArea() {
        // logic to sum up area of each shape
    }
}
```

Open-closed Principle

- Objects or entities should be open for extension, but closed for modification.
- Ex) Add a new kind of shape, i.e. Triangle

```
interface Shape {
    public float area();
}
class Triangle implements Shape { … }
…
class AreaCalculator {
    protected List<Shape> shapes;
    public float sumArea() {
        float sum = 0;
        for (Shape s : shapes) {   sum += s.area(); }
        …
    } …
}
```

Liskov Substitution Principle

- Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.
- Every subclass/derived class should be substitutable for their base/parent class.

```
class Triangle implements Shape {
    …
    public float area () { return -1;}
}
```

Interface Segregation Principle

- A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

```
interface Shape{
    …
    public int numEdges();
}
```

- Circle?

Dependency Inversion Principle

- Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

```
class AreaCalculator{
    protected float radius;
    protected float length;
    public AreaCalculator(…,
            float param) {
        …
        if (…//is a square)
            this.length = param;
        else // is a circle
            this.radius = param;
    }
}
```

X

Design Patterns (i.e. Low-Level Design)

Design patterns are descriptions of *communicating objects and classes* that are <u>customized</u> to solve a general design problem in a particular context.

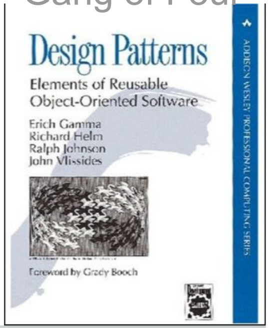
The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

Design Patterns (cont.)

Why design patterns?

- Appy working solutions to approaches
- Based on the implementations of many systems
- Capture and pass on the knowledge of experienced designers
  - Useful for inexperienced
  - Communicating about design

Gang of Four



Design Patterns (cont.)

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [Alexander]

Design Pattern Families

Creational

Concerned with the process of object creation

- Increases flexibility and reuse of code

Structural

Deal with the composition of classes or objects

- Organizing different classes and modules to form larger structures or add new functionality

Behavioral

Characterize the ways in which classes or objects interact and distribute responsibility

- Algorithms and assignment of responsibilities between objects

Creation Patterns

- **Abstract Factory**: Creates an instance of several families of classes
- **Builder**: Separates object construction from its representation
- **Factory Method**: Creates an instance of several derived classes
- **Object Pool**: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype**: A fully initialized instance to be copied or cloned
- **Singleton** A class of which only a single instance can exist

Structural Patterns

- **Adapter**: Match interfaces of different classes
- **Bridge**: Separates an object's interface from its implementation
- **Composite**: A tree structure of simple and composite objects
- **Decorator**: Add responsibilities to objects dynamically
- **Facade**: A single class that represents an entire subsystem
- **Flyweight**: A fine-grained instance used for efficient sharing
- **Private Class Data**: Restricts accessor/mutator access
- **Proxy**: An object representing another object

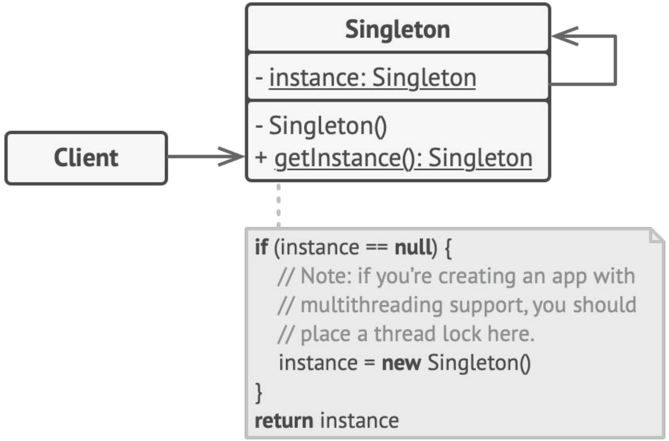
Behavioral Patterns

- **Chain of responsibility**: A way of passing a request between a chain of objects
- **Command**: Encapsulate a command request as an object
- **Interpreter**: A way to include language elements in a program
- **Iterator**: Sequentially access the elements of a collection
- **Mediator**: Defines simplified communication between classes
- **Memento**: Capture and restore an object's internal state
- **Null Object**: Designed to act as a default value of an object
- **Observer**: A way of notifying change to a number of classes
- **State**: Alter an object's behavior when its state changes
- **Strategy**: Encapsulates an algorithm inside a class
- **Template method**: Defer the exact steps of an algorithm to a subclass
- **Visitor**: Defines a new operation to a class without change

Creation: Singleton

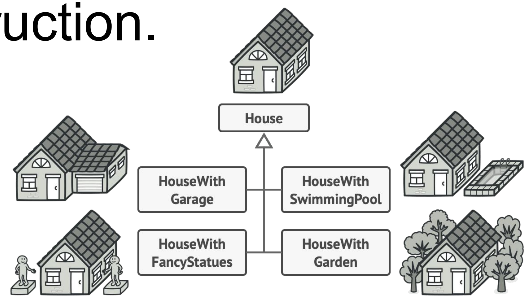
- Ensure a class only has one instance and provide a global point of access to it.

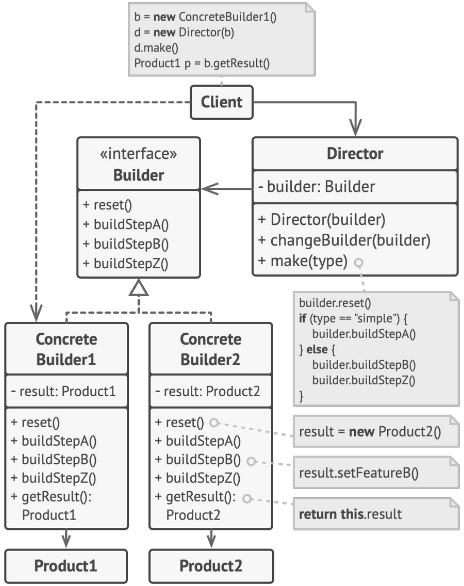
- Use when your program should have one instance available to all clients or stricter control over global variables.
  - i.e. single database shared in different parts of the program



Creation: Builder

- Separates object construction from its representation, allows the production of different types of objects using same code.
- Use when you need to construct objects step by step, create different representations of the same product, or defer construction.





Factory vs. Abstract Factory

Factory: create one object through inheritance.

```
class A {
    public void doSomething() {
        Foo f = makeFoo();
        f.whatever();
    }

    protected Foo makeFoo() {
        return new RegularFoo();
    }
}

class B extends A {
    protected Foo makeFoo() {
        //subclass is overriding the factory method
        //to return something different
        return new SpecialFoo();
    }
}
```

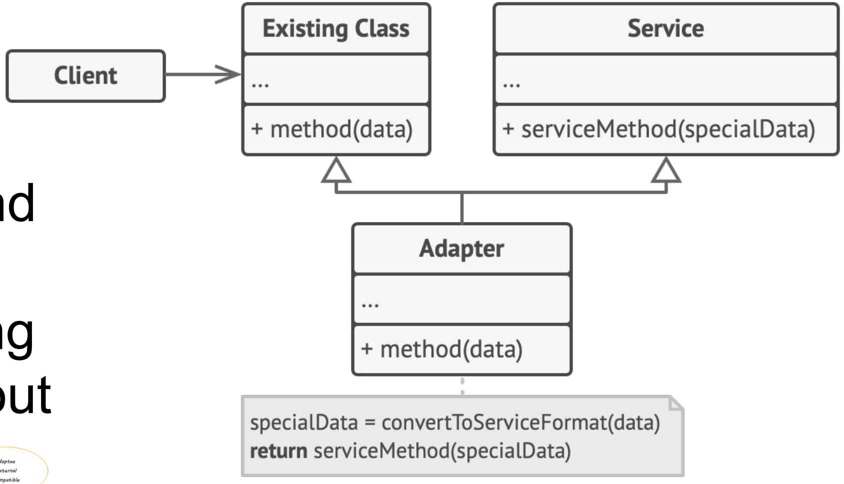
Abstract factory: create families of objects with abstraction.

- **Abstraction:** Reduce complexity by suppressing unnecessary details.

```
class A {
    private Factory factory;

    public A(Factory factory) {
        this.factory = factory;
    }

    public void doSomething() {
        //The concrete class of "f" depends on the concrete class
        //of the factory passed into the constructor. If you provide a
        //different factory, you get a different Foo object.
        Foo f = factory.makeFoo();
        f.whatever();
    }
}

interface Factory {
    Foo makeFoo();
    Bar makeBar();
    Aycufcn makeAmbiguousYetCommonlyUsedFakeClassName();
}
```

Structural: Adapter

- Allows objects with incompatible interfaces to collaborate.
  - Use when you need to create a middle-layer that serves as a translator between your code and another class (i.e. legacy code, 3rd-party library, etc.). Separating data conversion from logic without breaking client code.



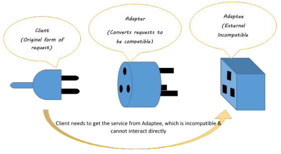
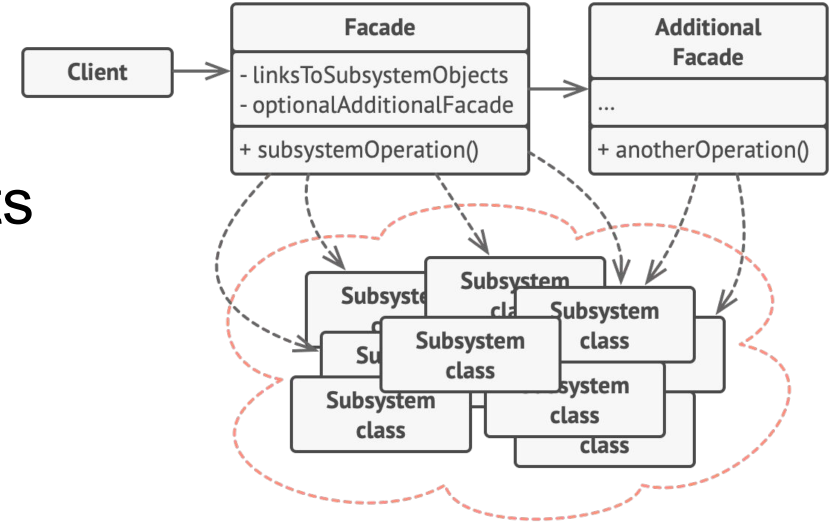
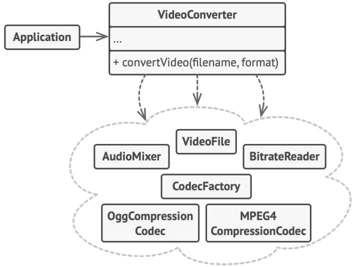


Figure 1-Adapter Pattern Concept

Structural: Facade

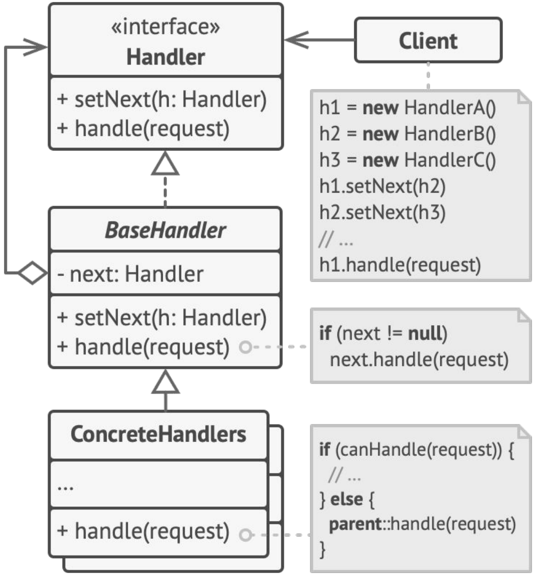
- Provides a simplified interface to represent an entire library, framework, or subset of classes.
- Hides system complexity system, client only interacts with facade.





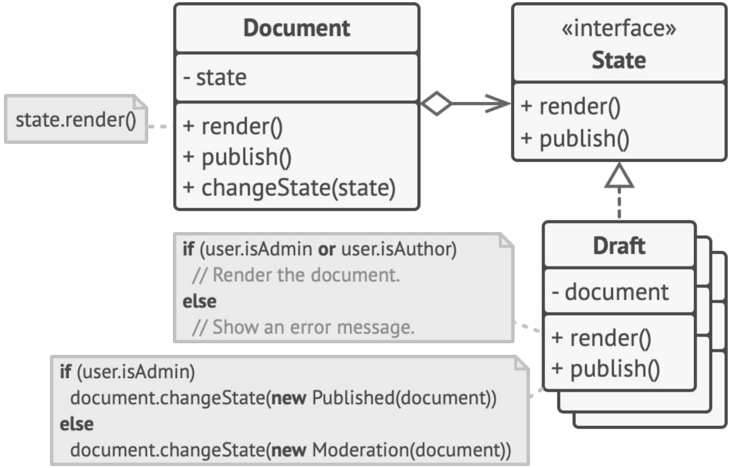
Behavioral: Chain of Responsibility

- Requests are passed through a chain of handlers, each handler decides on processing or passing.
- Useful if different types of requests are handled in various ways, unknown sequences or particular order.



Behavioral: State

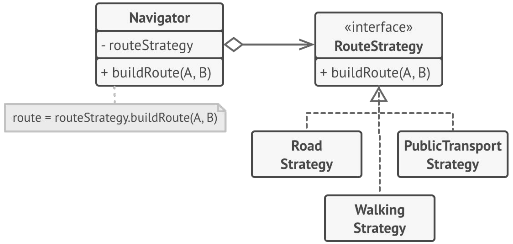
- Allows objects to alter their behavior when the internal state changes.
- Use if program behaves differently depending on the current state. Avoids massive conditionals in your code.
    - i.e. mobile development

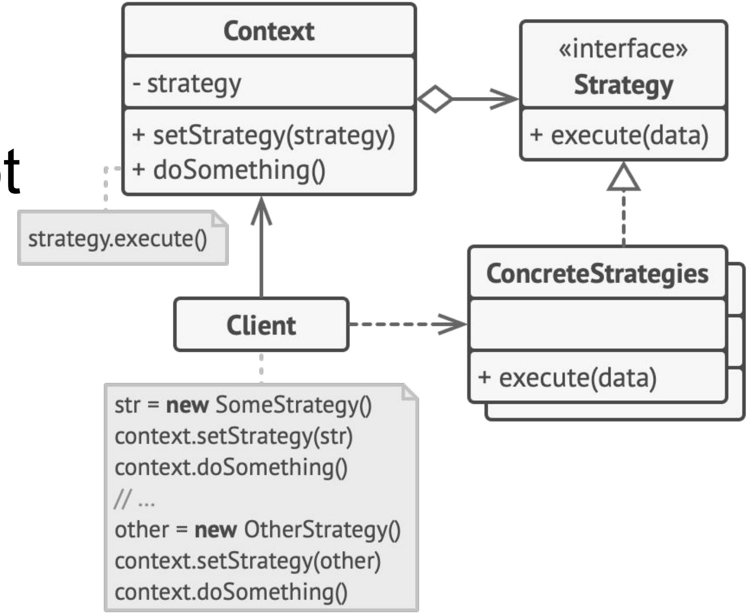


Behavioral: Strategy

- Separate algorithms into interchangeable classes.

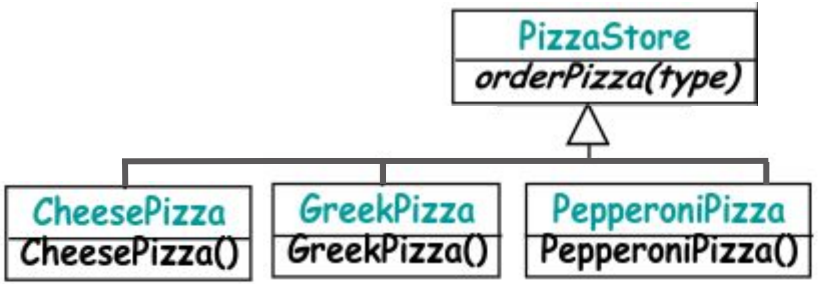
- Use when you want different variants of an algorithm or a lot of similar classes that differ in execution behavior.





Design Patterns to Code

Creating software for a Pizza Store 🍕



Discuss: What is wrong with this design?

Design Patterns to Code (cont.)

Problems…

– Client (PizzaStore) invokes different pizza constructors among other responsibilities

– New pizza types may be added

• Clam, Veggie, etc.

– New order types may be added

• In Store, Delivery, etc.

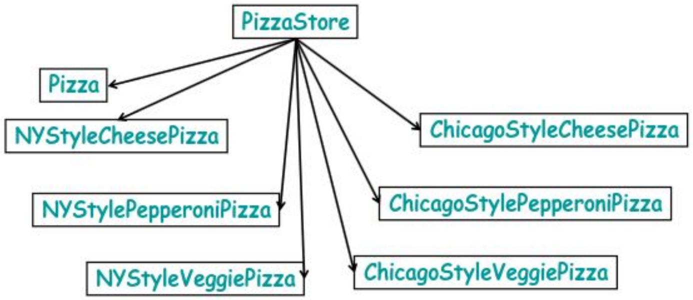
– Original pizza types may be removed

• Greek, etc.

– Different styles of pizza

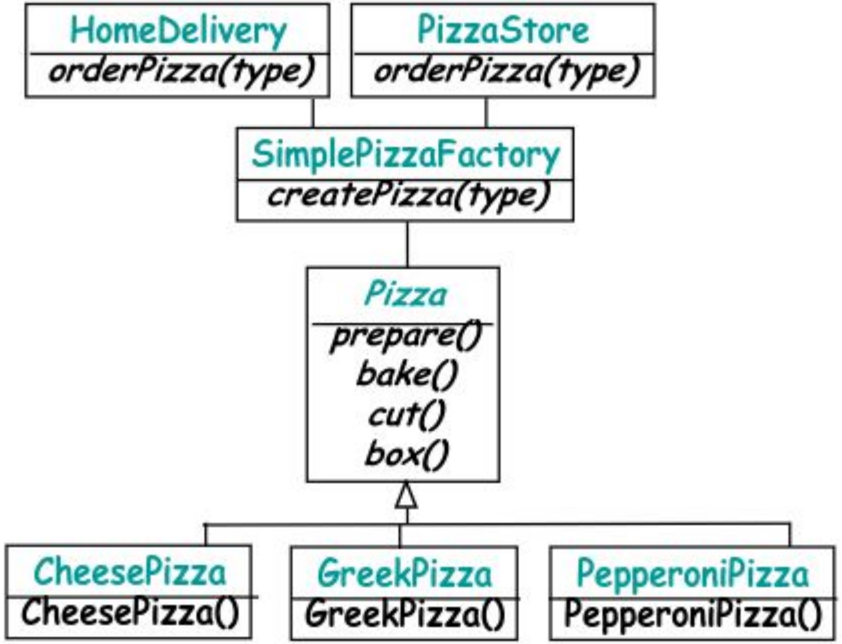
• Chicago, NY, Stuffed crust, etc.

– …



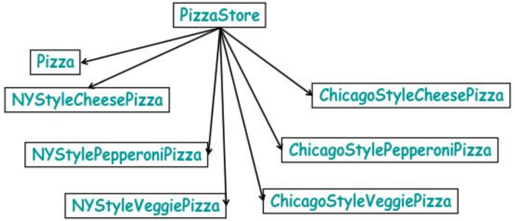
Solution: Factory (Pizza Type)

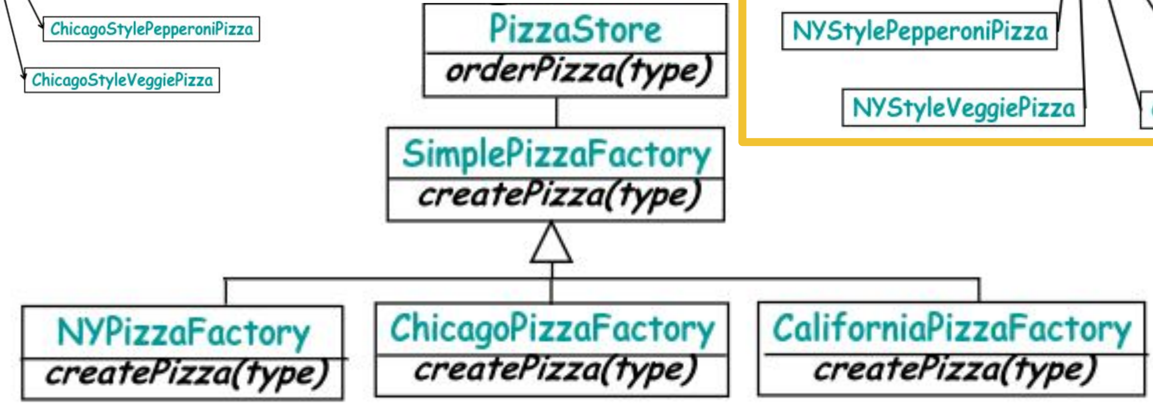
- Encapsulate object creation

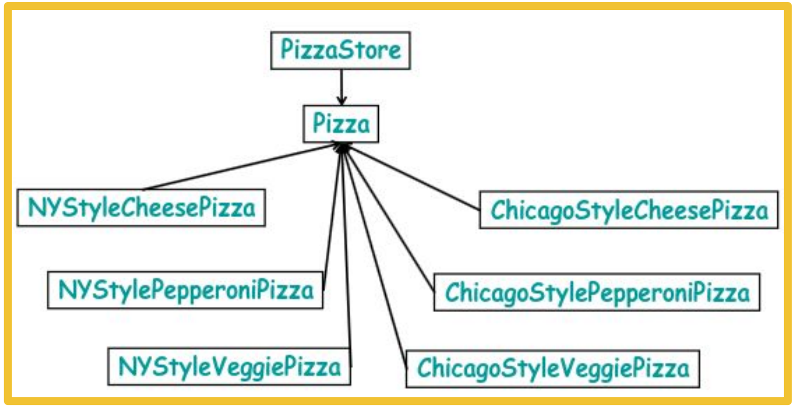


What about adding Pizza Styles?

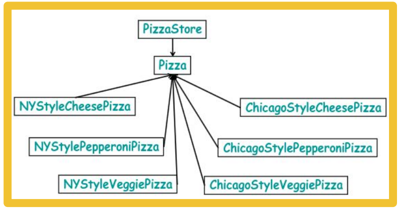
Factory can also support additional extensions.







Design Principles



Single-Responsibility: PizzaStore responsibilities reduced

Open-Closed: PizzaFactory open for new types/styles/etc., PizzaStore closed

Dependency Inversion Principle:

- Depend upon abstractions instead of concretizations.
- Use the pattern when:
  - A class cannot anticipate the class of objects it will create
  - A class wants its subclasses to specify the objects to create

Design Disclaimer

- No silver bullet for choosing design patterns.
    - For example, would the Builder pattern work for PizzaStore?
        - Yes

```
/** "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
        return pizza; }
    public void createNewPizzaProduct() {
        pizza = new Pizza(); }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
 }
```

Next Time…

- Continuing with Low-Level Design
- **Design Discussion Presentations Friday (10/27) in class**

References

- RS Pressman. *“Software engineering: a practitioner's approach”.*
- Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *“A Pattern Language”*. Oxford University Press, New York, 1977.
- Emil Vassev, Joey Paquet. *“Java Design Patterns”*. 2006-2012. Concordia University
- Refactoring Guru. “*Design Patterns”.* <https://refactoring.guru/design-patterns>
- Na Meng and Barbara Ryder
- Chris Parnin
- Sarah Heckman