SE464 Tutorial 1

Topics

- Introduction
- Questions
- System Modeling
- Decorator Pattern
- Observer Pattern
- Factory Pattern
- Tempate Pattern/NVI
- Visitor Pattern
- Measures of Design Quality
- Model-View-Controller

Introduction

- Your TA Ahmed El Shatshat
- Contact me on discord at ahmede3212

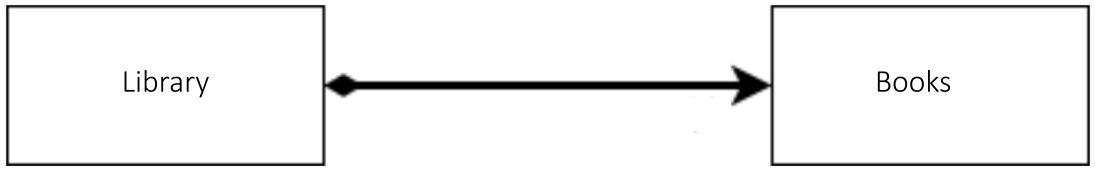
Questions?

System Modeling

Composition

- Known as an “owns a” relationship
- If A owns a B, then typically B has no existence outside of A
  - If A is destroyed, so is B
  - If A is copied, B is copied (deep copy)

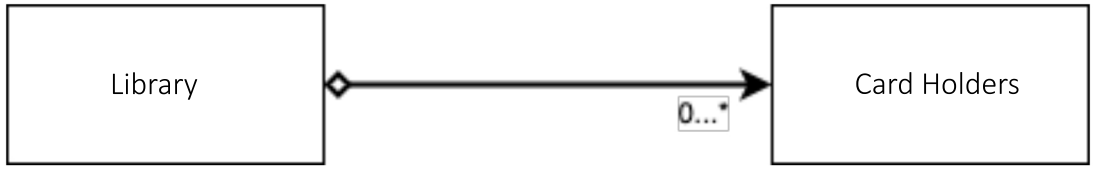
e.g. A library owns its books



Aggregation

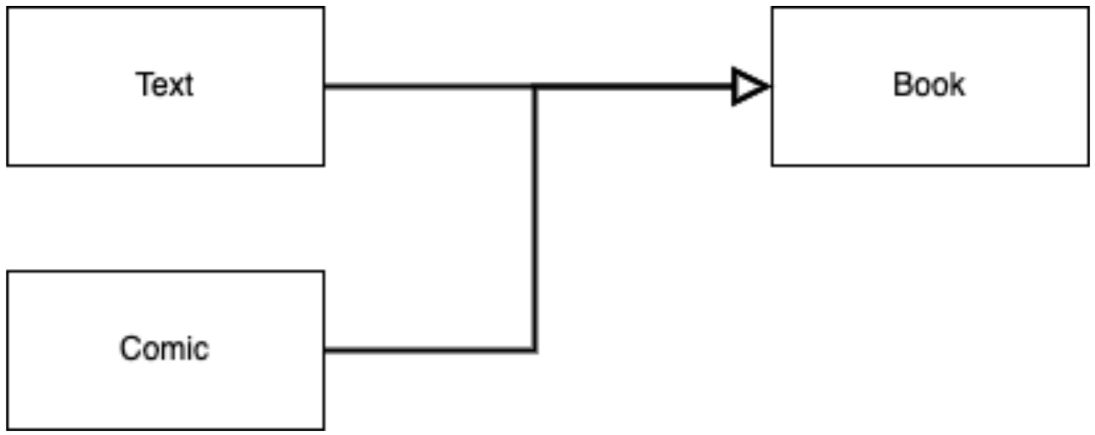
- Known as an “has a” relationship
- If A has a B, then typically B exists outside of A
  - If A is destroyed, B lives on
  - Copies of A are shallow copies (both share the same B)

e.g A library has library card holders



Inheritance Modeling

- Known as an “is a” relationship (a Text is a Book)

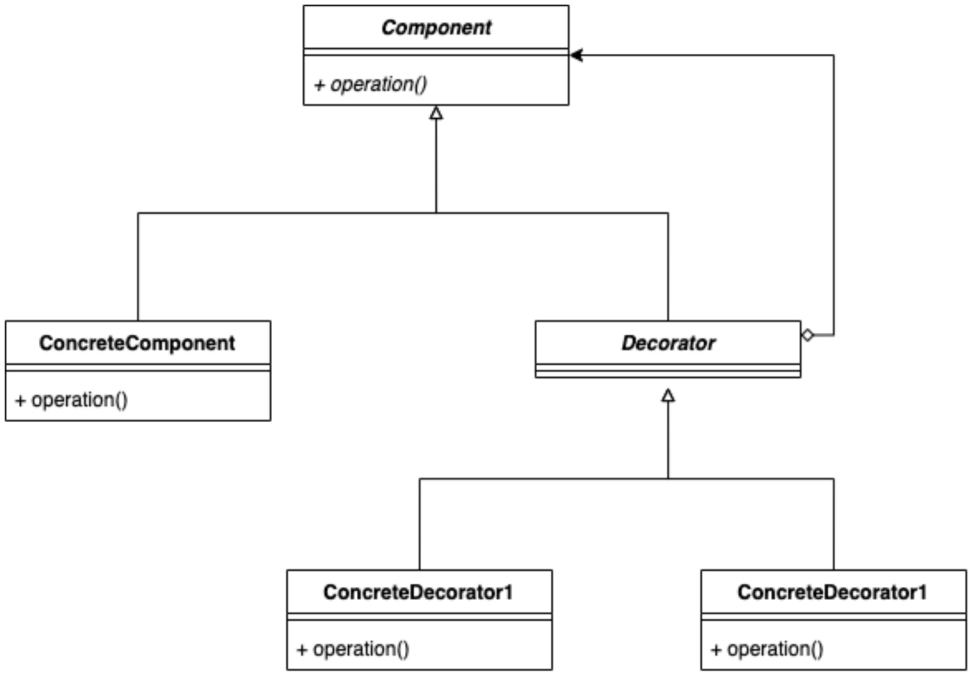


Decorator Pattern

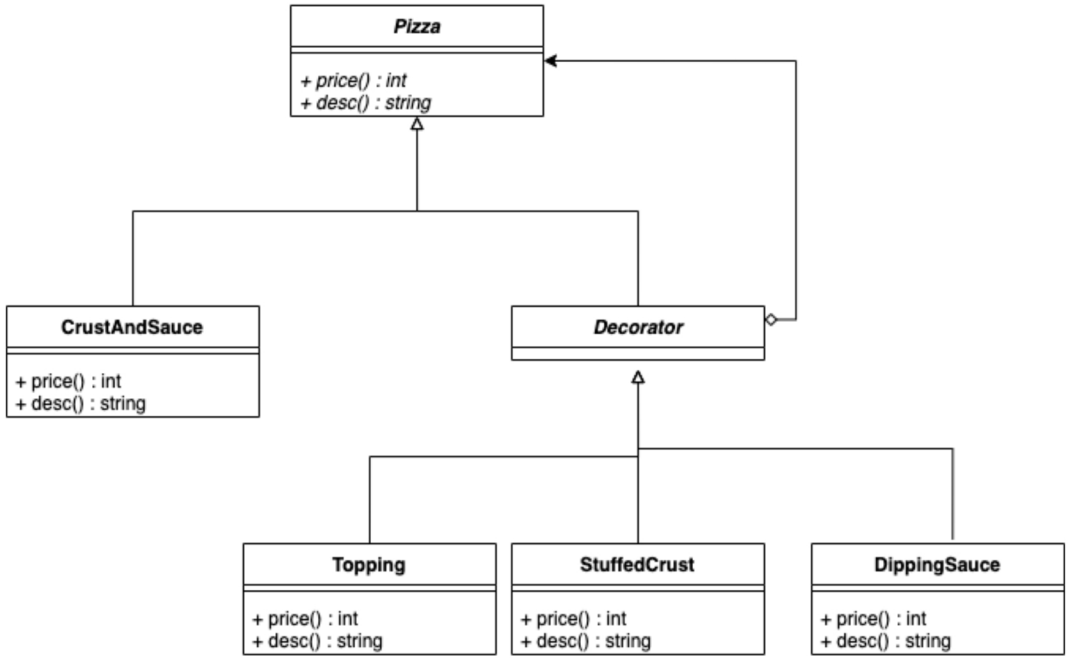
Decorator Pattern

- Goal is to add functionality or features to an object at runtime
- Class Component – defines the interface
- ConcreteComponent – implements the interface
- Decorators – all inherit from abstract Decorator, which inherits from Component
- Every Decorator is a Component and every Decorator has a pointer to the a component
- Whether it is "a has" a or "owns a" relationship depends on whether you consider the decorated object to be inseperable from its undecorated component
- Effectively a linked list

Decorator Pattern UML



Decorator Pattern UML



Decorator Pattern Example

```
class Pizza{
    public:
        virtual float price() const = 0;
        virtual string desc() const = 0;
        virtual ~Pizza(){}
};

class CrustAndSauce : public Pizza{
    public:
        float price() const override {return 5.99;}
        string desc() const override {return "Pizza :)";}
};
```

Decorator Pattern Example

```
class Decorator : public Pizza{
    protected:
            Pizza *component;
 private:
            Decorator(Pizza *p) : component{p}{}
            ~Decorator(){ delete component;}
};
class StuffedCrust : public Decorator{
 private:
   StuffedCrust(Pizza *p) : Decorator{p}{}
   float price() const override{ return component -> price() + 2.69;}
   float desc() const override{ return component -> desc() + " with stuffed
crust";}
};
```

Decorator Pattern Example

```
class Topping : public Decorator{
 string top;
    private:
StuffedCrust(Pizza *p, string t) : Decorator{p}, top{t}{}
float price() const override{ return component -> price() + 0.75;}
float desc() const override{ return top + " " + component -> desc();}
};

Client:
Pizza *p1 = new CrustAndSauce();
p1 = new Topping{"cheese", p1};
p1 = new Topping{"mushrooms", p1};
p1 = new StuffedCrust{p1};
cout << p1 -> desc() << " " << p1 -> price() << endl;
```

Observer Pattern

Observer Pattern

- Publish – subscribe model
- One class: Publisher/subject -> generates data
- One or more classes: subscriber/observer -> read and react to data
- Can be many different kinds of observer objects

Sequence of method calls:

1) Subject's state is updated
2) Subject::notifyObservers calls each observer's notify() function
3) Each observer calls ConcreteSubject::getState to query the state and update accordingly

Observer Pattern Example

Usage:

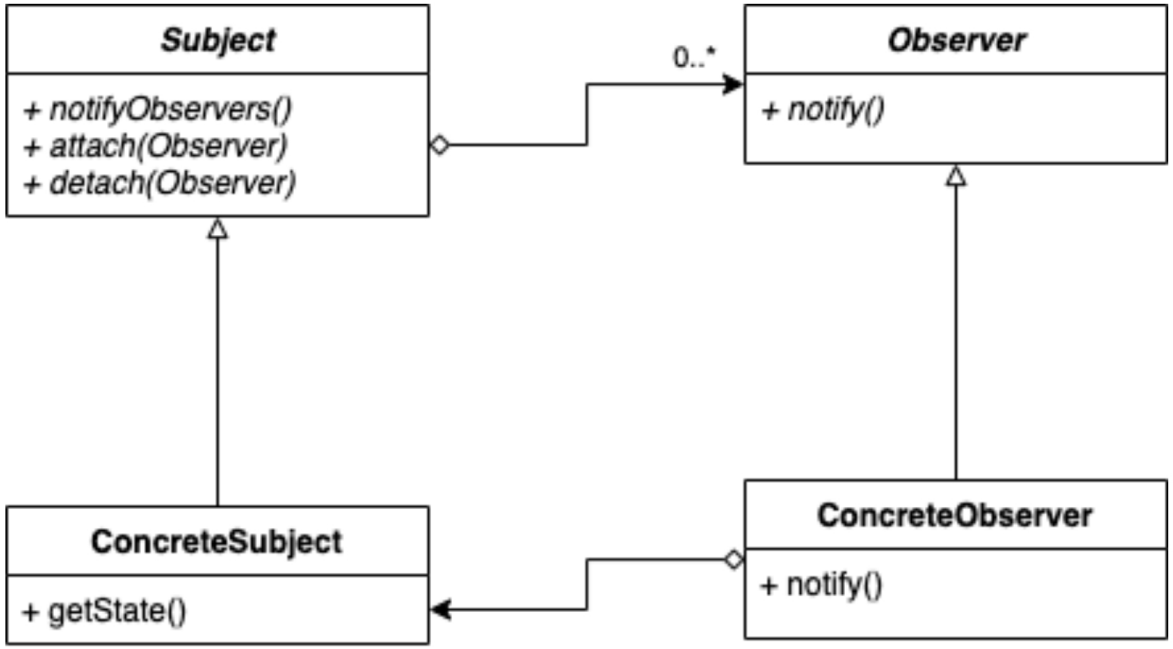
```
class Subject{
        vector<Observer*> observers;
    public:
        void attach(Observer *ob){ observers.emplace_back(ob);}
        void detach(Observer *ob){ //Remove from list}
        void notifyObservers(){
            for(auto ob : observers){
                ob.notify();
            }
        }
        virtual ~Subject() = 0;
};
Subject::~Subject{} //in .cc file
```

Observer Pattern Example

Usage:

```
class Observer{
        public:
            virtual void notify() = 0;
            virtual ~ Observer(){}
};
```

Observer Pattern UML



Factory Pattern

Factory Pattern

- Problem: We want a way to generate objects that can change dynamically
- Eg. Writing a video game with 2 different enemies, turtles and bullets
- System randomly generates either turtles or bullets
- Want more bullets to generate on harder levels

Factory Pattern Example

Usage:

```
class Level{
    public:
        virtual Enemy* createEnemy() = 0; //factory method
};

class Easy : public Level{
    public:
        Enemy *createEnemy() override{ //create mostly turtles}
};
class Hard : public Level{
    public:
        Enemy *createEnemy() override{ //create mostly bullets}
};
```

Client:

```
Level *l = new Easy;
Enemy *e = l->createEnemy();
```

Template Pattern

Template Pattern

- Problem: Want subclasses to override only **some** aspects of superclass behaviour
- Eg. Have red and green turtles, but should only override how shell is drawn

```
class Turtle{
      public:
         void draw(){ drawHead(); drawShell(); drawFeet();}
      private:
         void drawHead(){ //draw head }
         virtual void drawShell() = 0; //Overridden by subclasses
         void drawFeet){ //draw feet }
};
```

Template Pattern

- Subclasses then only override the templated method they have to

```
class RedTurtle : public Turtle{
        void drawShell() override{ //draw Red Shell}
};

class GreenTurtle : public Turtle{
        void drawShell() override{ //draw Green Shell}
};
```

Subclass can only change the way the shell is drawn, not the other parts

Non-Virtual Interface (NVI) Idiom

A generalization of the Template pattern -> puts every virtual function inside a template method

Public virtual methods have an issue. They are:

- Public, and thus an interface to the client
  - Indicates promised behaviour
  - Should guarantee pre/post conditions, as well as class invariants
- Virtual, and thus an interface to a subclass
  - Subclass can change this to anything, thus the promised behaviour could be broken
  - Pre/post conditions and class invariants could be ignored

A public virtual method can never guarantee that the promised behaviour occurs, as the subclass is always free to change everything about it

Non-Virtual Interface (NVI) Idiom

- NVI says: all public methods should be non-virtual (except constructors)
- All virtual methods should be private, or at least protected

```
class DigitalMedia{
     public:
          virtual void play() = 0;
};
```

vs

```
class DigitalMedia{
          virtual void doPlay() = 0;
     public:
          void play(){
                    doPlay();
          }
};
```

Non-Virtual Interface (NVI) Idiom

- By doing this we can enforce certain behaviour to occur either before or after the virtual method
- If we left the subclasses to handle it they might leave it out, either by accident or on purpose

```
class DigitalMedia{
          virtual void doPlay() = 0;
     public:
          void play(){
                    checkCopyright();
                    doPlay();
                    incrementPlaycount();
          }};
```

Visitor Pattern

Visitor Pattern

- Problem: Want to achieve double dispatch i.e be able to have a process depend on two different object types
- Eg. Want to have a strike method that depends on the Enemy being struck and the Weapon used to strike it

```
class Enemy{
    public:
        virtual void beStruckBy(Weapon &w) = 0;
        ...
};
```

Visitor Pattern

```
class Turtle : public Enemy{
    public:
         virtual void beStruckBy(Weapon &w ) {w.strike(*this);}
         ...
};
class Bullet : public Enemy{
     public:
         void beStruckBy(Weapon &w) {w.strike(*this);}
         ...
};
```

Visitor Pattern

```
class Weapon {
     public:
          virtual void strike(Turtle &t) = 0;
          virtual void strike(Bullet &b) = 0;
};

class Stick : public Weapon {
     public:
          void strike(Turtle &t){ /*strike turtle with stick */}
          void strike(Bullet &b){ /*strike bullet with stick */}
};
```

Visitor Pattern

```
Enemy *e = new Bullet{...};
Weapon *w = new Rock{...};

e -> beStruckBy(*w); //Bullet::beStruckBy runs (virtual method dispatch)
                     //Then calls Weapon::strike, *this is of type Bullet
                     //Bullet version is chosen at compile time
```

Visitor Pattern

- Problem 2: Add functionality to existing class without changing or recompiling it (for whatever reason)
- Could add a virtual function, but don't want to (for whatever reason)
- Eg. Add a visitor to the Book class to count author, topic, and hero

```
class Book {
     public:
          virtual void accept(BookVisitor &bv) {bv.visit(*this)}
};
class Text : public Book {
     public:
          void accept(BookVisitor &bv) {bv.visit(*this)}
};
```

Visitor Pattern

```
class BookVisitor{
     public:
          virtual void visit(Book &b) = 0;
          virtual void visit(Text &b) = 0;
          virtual void visit(Comic &b) = 0;
};
struct Catalogue : public BookVisitor{
          map<string, int> theCatalogue;
          void visit(Book &b){theCatalogue[b.getAuthour()]++}
          void visit(Text &t){theCatalogue[b.getTopic()]++}
          void visit(Comic &c){theCatalogue[b.getHero()]++}
};
```

Visitor Pattern Issue

- Main issue with Visitor pattern is having to define an “accept” method for every subclass
- If you want a new accept method or make a new subclass you have to make sure you do all the book keeping to get everything in order

Measures of Design Quality

Coupling

- How much distinct program modules depend on each other

LOW

Modules communicate via function calls with basic parameters/results

Modules pass arrays and structs back and forth

Modules affect each other's control flow

Modules have access to each other's implementation (friendship)

HIGH

- High coupling -> changes to one module require greater changes to the other module
- Makes it harder to reuse individual modules

Cohesion

- How closely elements of a module relate to reach other

LOW

An arbitrary grouping of unrelated elements (<utility>)

Elements share a common theme, are otherwise unrelated (<algorithm>)

Elements manipulate state over the life of an object (file manipulation)

Elements pass data to each other

Elements cooperate to perform exactly 1 task (Parse a string into a tree)

HIGH

- Low cohesion -> Poorly organized code, hard to understand
- Makes it harder to reuse the module

Model-View-Controller

Model-View-Controller

- Architectural model to achieve low coupling and high cohesion
- Separate the distinct notions of:
  - the data (or state – "model")
  - the representation of the data ("view")
  - and the control for the manipulation of the data ("controller")
- Model can have multiple views (e.g text and graphics)
  - Doesn't need to know about their details
- Controller mediates control flow through model and view
- By decoupling presentation and control, MVC promotes reuse

MVC Diagram


View
Controller
Model
LIBRARY
SUPER NINTENDO
SELECT
START
X
A
Y
B