More Design Heuristics

Hard stop at 10:50 so we can play Design Jeopardy!!!!

Let's recap

- There is no "right answer" with design
- Applying effective **heuristics** can provide insights and lead to a good design

Code Complete 1-9

1. Identify objects
2. Form consistent abstractions
3. Encapsulate implementation details
4. Favor composition over inheritance
5. Hide information
6. Keep coupling loose and cohesion strong
7. Identify areas likely to change
8. Use design patterns
9. Consider testability
10. Other common principles

We'll explore 4 today

1. Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface
  - Clients should not be forced to depend on methods that they do not use
  - Example: Dogs jump but don't sing

ISP Example: Timed door

```
class Door
{
  public:
  virtual void Lock() = 0;
  virtual void Unlock() = 0;
  virtual bool IsDoorOpen() = 0;
};
```

TimedDoor needs to sound an alarm when the door has been left open for too long. To do this, the TimedDoor object communicates with another object called a Timer.

ISP Example: Timed door

```
class Timer
{
   public:
   void Register(int timeout, TimerClient* client);
};


class TimerClient
{
   public:
   virtual void TimeOut() = 0;
};
```

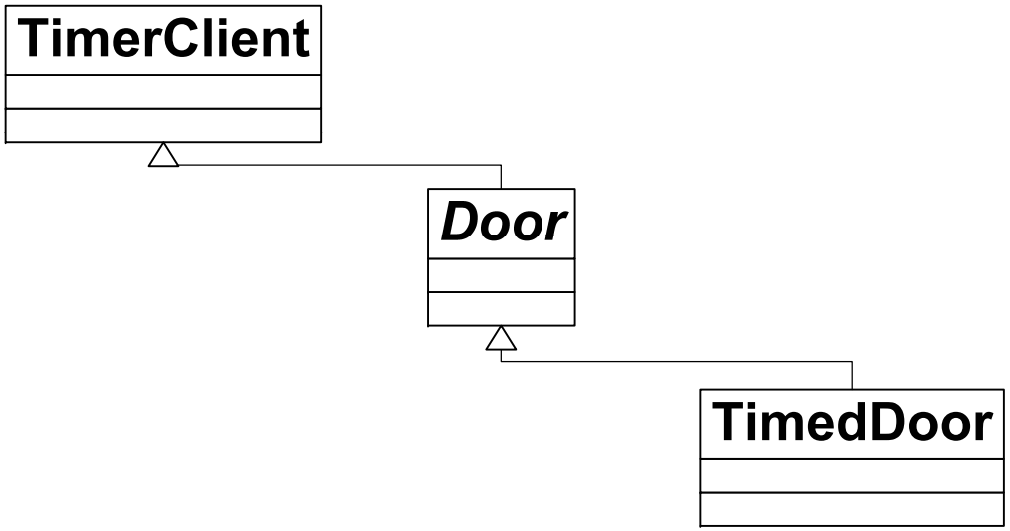
time of timeout

object to invoke TimeOut() on when timeout occurs

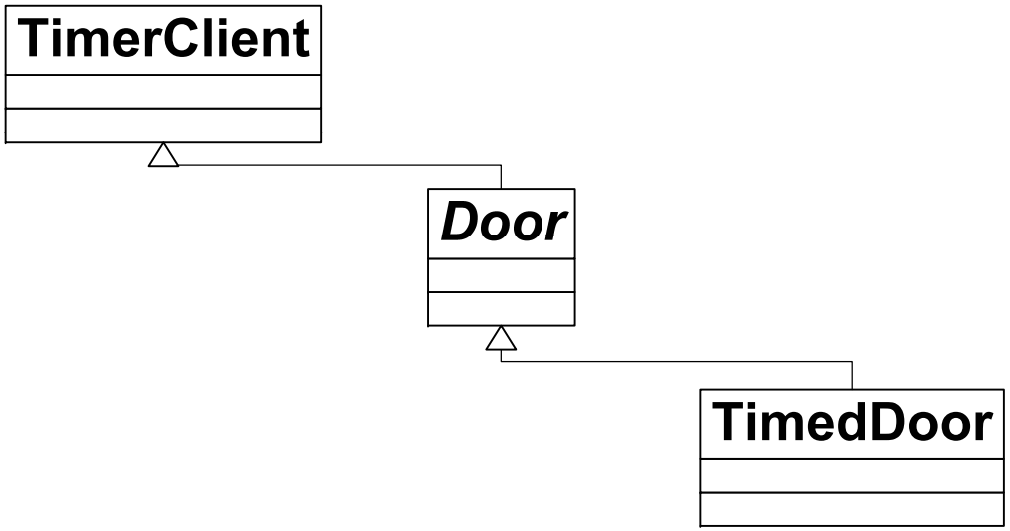
TimeOut method

How should we connect the TimerClient to a new TimedDoor class so it can be notified on a timeout?

InterfaceSegregationPrinciple Solution: yes or no?

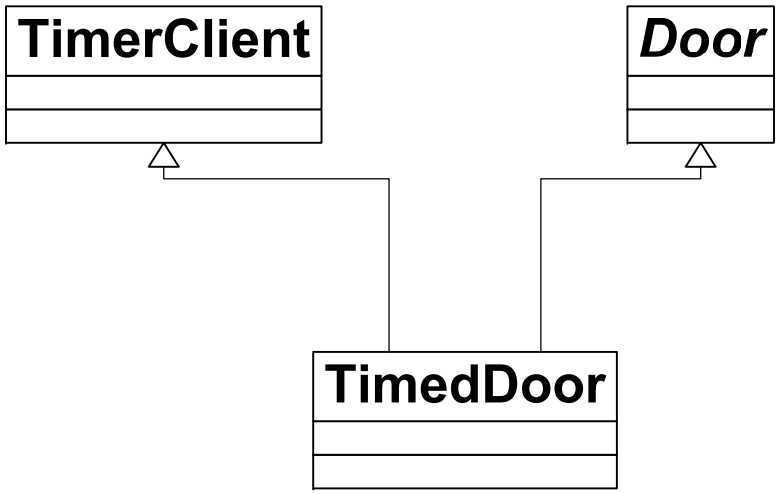


InterfaceSegregationPrinciple Solution: yes or no?

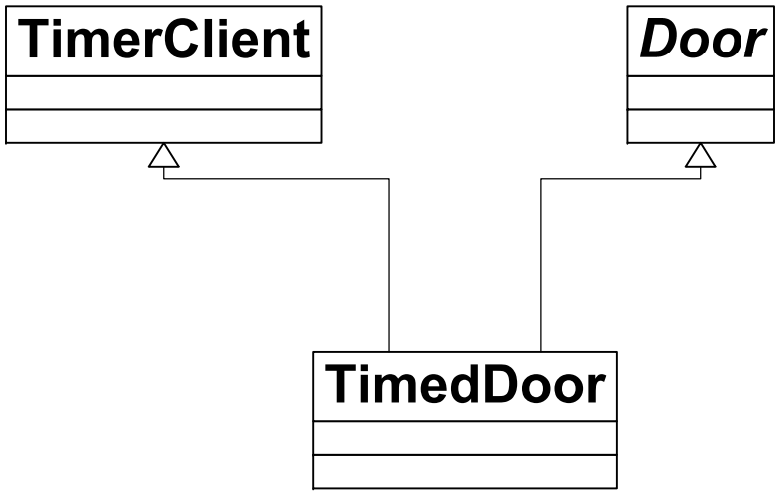


No, as it's polluting the Door interface by requiring all doors to have a TimeOut() method

ISP Solution: yes or no?



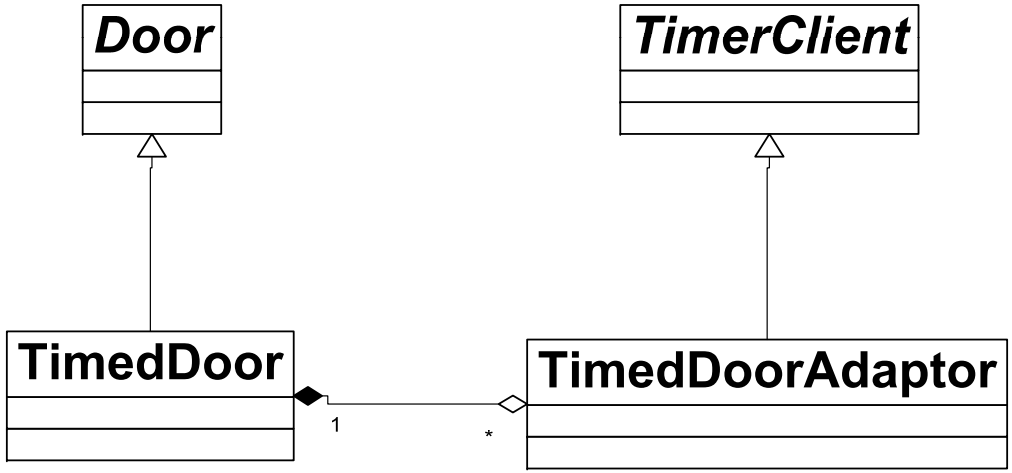
ISP Solution: yes or no?



Yes, separation through multiple inheritance

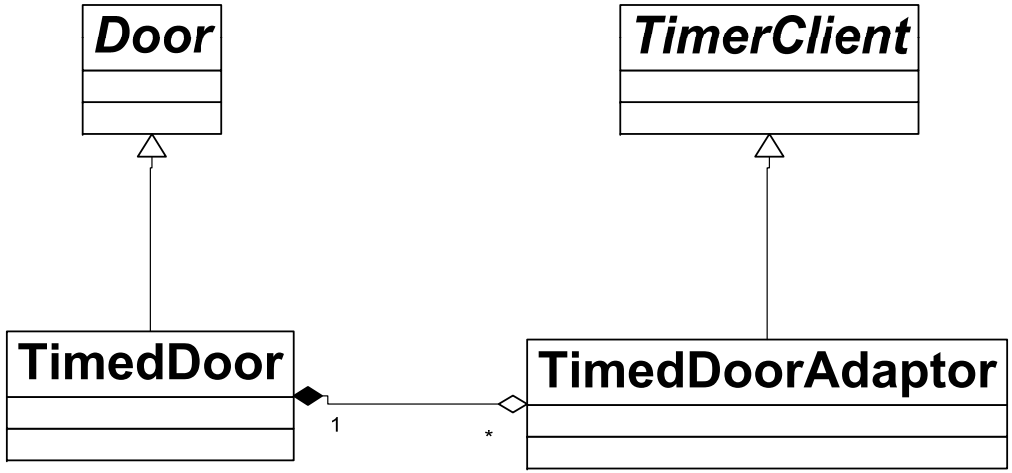
ISP solution: yes or no?

When the Timer sends the TimeOut message to the TimedDoorAdapter, the TimedDoorAdapter delegates the message back to the TimedDoor.



ISP solution: yes or no?

When the Timer sends the TimeOut message to the DoorTimerAdapter, the DoorTimerAdapter delegates the message back to the TimedDoor.



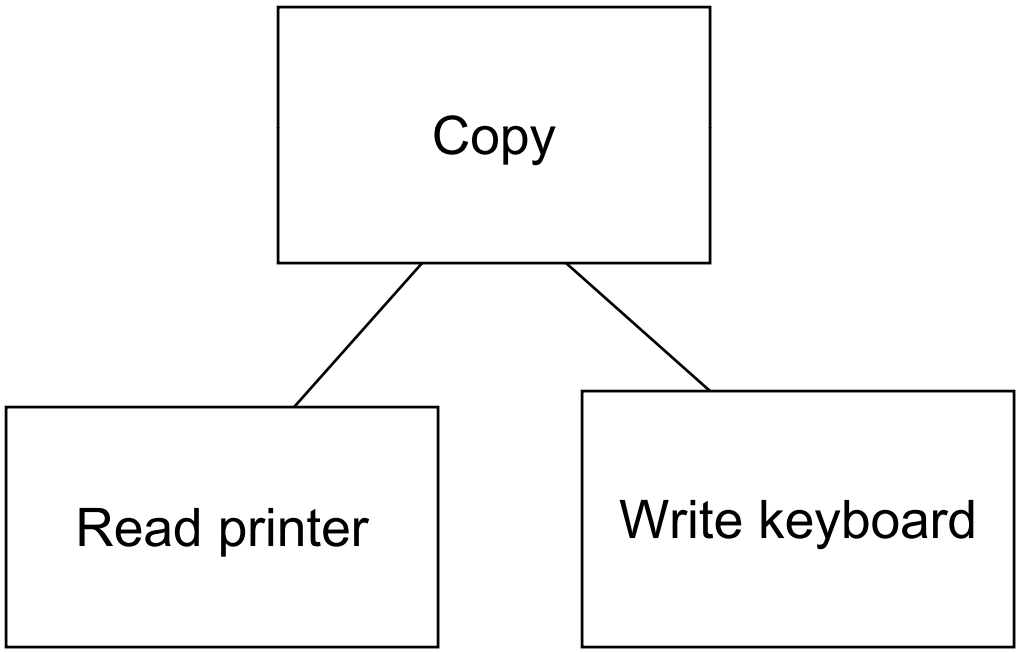
Yes, separation through delegation

2. Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- Example: Separation of policy and mechanism

DependencyInversionPrinciple: Example

Copy characters from a printer to a keyboard



DIP: yes or no?

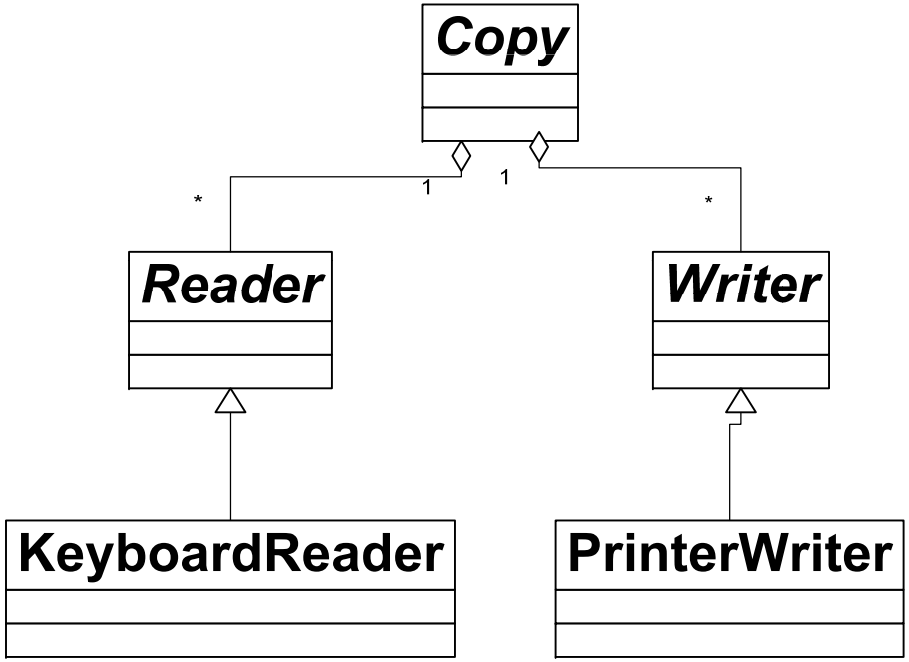
```
void copy()
{
   int c;
   while ((c = ReadKeyboard()) != EOF)
       WritePrinter(c);
}
```

DIP: yes or no?

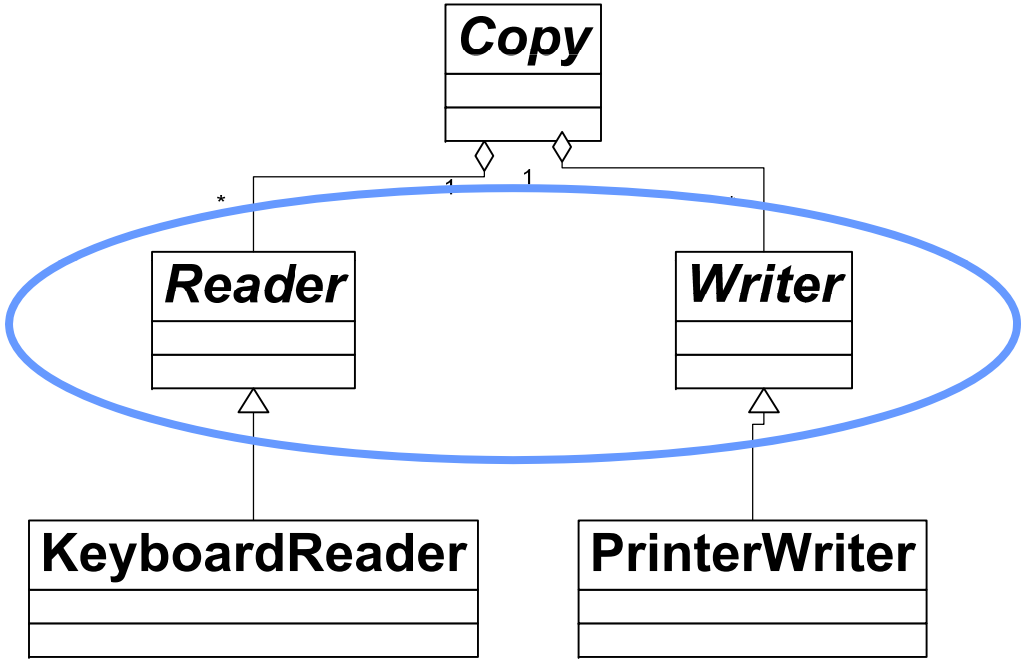
```
void copy()
{
   int c;
   while ((c = ReadKeyboard()) != EOF)
       WritePrinter(c);
}
```

Not really. Copy (high level) is quite dependent on the keyboard and printer (low level). It cannot be easily reused for another domain.

DIP: yes or no?



DIP: yes or no?



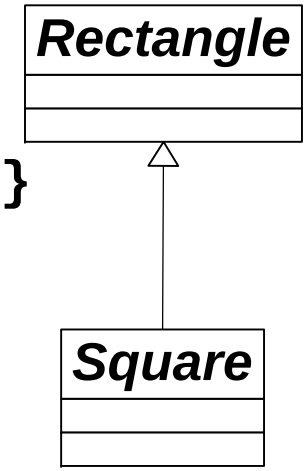
Yes. Reader and Writer provide an abstraction through which the high and low modules interact. Copy is now more widely usable.

3. Liskov Substitution Principle

- Subtypes must be substitutable for their base types
- Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it
- Example: Anyone? Do we need a translator for this one?

LSP: Example – Rectangle & Square

```
class Rectangle
{
public:
   void SetWidth(double w) {itsWidth=w;}
   void SetHeight(double h) {itsHeight=w;}
   double GetHeight() const {return itsHeight;}
   double GetWidth() const {return itsWidth;}
private:
   double itsWidth;
   double itsHeight;
};
```



Reasonable to derive a square from a rectangle, right?

LSP: any problems with this?

```
class Square : public Rectangle {
…
};

void Square::SetWidth(double w)
{
   Rectangle::SetWidth(w);
   Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
   Rectangle::SetHeight(h);
   Rectangle::SetWidth(h);
}
```

LSP: what about now?

```
void morph(Rectangle& r)
{
   r.SetWidth(32);//calls Rectangle::SetWidth
}
```

Could we call **`morph`** with a **`Square`**?
What would happen?

Liskov Substitution Principle:
Subtypes must be substitutable for their base types

LSP: but, but, but

- Couldn't we fix this with virtual functions, requiring setWidth and setHeight to be implemented by the subclasses?

```
void g(Rectangle& r)
{
      r.SetWidth(5);
      r.SetHeight(4);
      assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

4. Open Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Reasonable? Why?

OpenClosedPrinciple: yes or no?

Resource allocator allocates different objects, based on their type.

```
class ResourceAllocator {
   public:

   int Allocate(int resourceType) {
      int resourceId;
      switch (resourceType) {
        case TIME_SLOT:
            resourceId = FindFreeTimeslot();
            MarkTimeslotBusy(resourceId); break;
         case SPACE_SLOT:
            resourceId = FindFreeSpaceSlot();
            MarkSpaceslotBusy(resourceId); break;
         default:
            Trace(ERROR, "Attempted to allocate
                    invalid resource\n"); break;
      }
      }
      …
```

OpenClosedPrinciple: yes or no?

```
class ResourcePool {
    public:
      virtual int FindFree() = 0;
      virtual int MarkBusy() = 0;
      virtual Free(int resourceId) = 0;
};

class TimeslotPool : public ResourcePool{…};
class SpaceslotPool: public ResourcePool{…};


class ResourceAllocator {
  ResourcePool *rpool[MAX_RESOURCE_POOLS];

 public:
  int Allocate(int resourceType) {
    int resourceId;
    resourceId= rpool[resourceType]->FindFree();
    rpool[resourceType]->MarkBusy(resourceId)];
}
```

The list goes on…

We could spend a whole quarter talking about design

- Single responsibility principle (a class should have only 1 reason to change)
- No redundancy principle
- Make the common case fast and the uncommon case correct
- Fail early, gracefully, and transparently

Follow good practices, read good code, read good books, get feedback on your designs, iterate, iterate, iterate!

Today's references

- http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

Design Jeopardy

Ground rules

- Project teams act as teams (min 4 players)
- All try to answer
- First with answer up times out everyone
- All with correct answer get points

- Grand winner gets
  - Ice Cream Bars
  - Winning impaired teams get
    - Popsicles

Practice with Presenter

- Write your names in the Answer box
- Submit it

Answer

Team:

Design for 100

- One 403 project is clearly implementing a FAÇADE

- Which one is it?

Answer

Team:

Design for 100

- Describe 2 distinct advantages of providing an iterator for your data structure?

Answer

Team:

Design for 200

1. A database is a reasonable example of what pattern?
2. What characteristic of the pattern is not typical with the database?

Answer

Team:

Design for 200

- The public switched telephone network is an example of what type of design pattern?

- Focus on the fact that there are many resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers.

Answer

Design for 200

Team:

```
void process(String[] words) {
    // Loop through the array of Strings
    for(int i=0; i<words.length; i++) {
        String argument = "";
        // Reverse the characters in each String
        for(int j=words[i].length(); j>0; j--)
            { argument += words[i].substring(j-1,j); }
        System.out.println(argument); }
        // Test for two particular Strings
        if(words.length == 2){
            if(words[0].toLowerCase().equals("mighty") &&
                words[1].toLowerCase().equals("mouse"))
                    System.out.println("...here he comes to save the day."); }
}
```

1. Why is the process() method not “cohesive”?
2. What changes would you make to make it so?

Answer (+ draw changes on code above)

Team:

Design for 200

- From a design perspective, would it be appropriate to implement a class House via inheritance from class Building in order to reuse the implementation of the many useful methods already defined in Building even if Building has other methods that have no meaning in the context of House?

  1. What is your answer?
  2. What pattern/principle/heuristic guided you?

Answer

Design for 500!!!!!!

Just kidding …. who's our winner?

Applause, applause, applause – great game everyone!