More Design Heuristics
Let’s recap

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

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

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

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

Yes, separation through multiple inheritance
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
Dependency Inversion Principle: Example

Copy characters from a printer to a keyboard

- Read printer
- Write keyboard
DIP: yes or no?

```c
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?

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?
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?

```cpp
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?

```c
void g(Rectangle& r)
{
    rsetWidth(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;
            }
        }

        ...

CSE 403, Spring 2007, Alverson
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[MAXRESOURCE_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

Beta Release Highlights

- Functionality wise, we’d like to see all the main components integrated in a very basic way. Be able to have a tracer bullet go from your front end through to your back end, and back.

- Binary distribution
  - Generally a client package and a server package (2 tar files) with installation instructions
  - Don’t package the client if it’s simply a web page
  - Don’t package the database
  - We’d like to install the client (or use your web page – provide your url) and connect it to your live server

- Source distribution
  - Again, a client source package and a server source package, with building instructions for a developer
  - Don’t package the database

- Updated SRS and SDS docs with change tracking on (focus is on content changes vs polish at this stage)
Beta Release Highlights (2)

- Joel-test artifacts:
  - Source control
  - Automated build (directions, scripts, output)
  - Bug database (reasonable use/assignments)

- Application of one design principle that we’ve talked about (other than iterator)

- While you should be working on these, we won’t ask for: test results, more than trivial user documentation (so we know what to do with the beta), until the Final Release
Beta Release Highlights (3)

- We’d like to schedule 15 minute beta demos with you:
  - 3 during Friday’s class
  - 3 later Friday afternoon at a mutually convenient time
    (12-2:30, 3:30-6) – please email me your options

- What’s your preference for demo’ing with your customer group?
  - Together with the exec demo – sounds like the ideal
  - At a separate time – second choice if necessary due to scheduling

- No reading summary next week (but still read the readings!) - to allow you to focus on your beta
Next week

- Quality Assurance – Testing and beyond!
- Wednesday – guest lecturer, Marty Stepp