GOALS & DESIGN PRINCIPLES
What characteristics of software would customers desire?

Of these, which work well together, which are in opposition?
GOALS

- **Correct**: does what it is supposed to (no bugs)
- **Robust**: secure, dependable, resilient
- **Useable**: UI, compatibility, ...
- **Many features**: does lots of "good" stuff
- **Good performance**: fast, low energy usage, ...
- **Scalable**
- **Maintainable/extensible**: Can add and change features
- **Quick delivery**: low development time
- **Low cost**: less spending is preferred
GOALS

- **Correct**: does what it is supposed to (no bugs)
- **Robust**: secure, dependable, resilient
- **Useable**: UI, compatibility,…
- **Many features**: does lots of "good" stuff
- **Good performance**: fast, low energy usage
  - Scalable
- **Maintainable/extensible**: Can add and change features
- **Quick delivery**: low development time
- **Low cost**: less spending is preferred

Must have

Scope: What we must build
- Goals are in opposition with each other
- Can’t have all 3
- Software engineering may support more of each
- We have **10** developers to build our software product
  - Estimated time **1 year**
  - Developer salary: $125,000/year
  - Total cost: $1,250,000
- Can we just hire **20** developers and do it in **6 months**?
  - Each person does half as much work
  - Total cost still $1,250,000
  - Seems better, right?
- Could we take this further?
  - 40 developers in 3 months?
  - 3,650 developers in 1 day?
  - 87,600 developers in 1 hour?
  - 5,256,000 developers in 1 minute?
EMBARRASSINGLY PARALLEL PROBLEM

LITTLE OR NO EFFORT IS REQUIRED TO SEPARATE THE PROBLEM INTO A NUMBER OF PARALLEL TASKS
SOFTWARE DEVELOPMENT IS NOT EMBARRASSINGLY PARALLEL

- Significant communication & synchronization overheads
- 20 developers accomplish in 9 months
- 40 developers accomplish in 8 months
- Adding person power to a project often makes it later
“HOW DOES A SOFTWARE SYSTEM GET TO BE ONE YEAR LATE? ONE DAY AT A TIME”

- FREDERICK P. BROOKS, PHD
COROLLARY

- **Effective parallelism** between developers requires **independent tasks**

- Independent tasks are achieved by clearly **specified interfaces** that one developer can use while another implements
OBJECT-ORIENTED DESIGN FOUNDATIONS

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation
DESIGN PRINCIPLES

- Least Surprise/Astonishment
- Don’t Repeat Yourself (DRY)
- Low Coupling/High Cohesion
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Separation
- Dependency Injection
- Design for testability

We will keep coming back to these throughout the semester
Reduce how much another developer is surprised by what happens

- No unexpected side-effects
  - getSomething() should not change state
- No reliance on hidden knowledge to make it work
  - "You have to call methodA right before methodB or it crashes"
  - "It got stuck in an infinite loop? Yeah, that happens if you pass a negative number to methodC…"
- Names make sense.
DON’T REPEAT YOURSELF (DRY)

- Same code should only appear once
- If code needed more than once, use a function
- If constant needed more than once, give it a name
- **Gut Check**: If you change in place 1 and that requires you to make the same change in place 2, then you’ve violated DRY
LOW COUPLING/HIGH COHESION

- **Coupling**: interdependence and coordination between entities (i.e., classes, functions, modules)
- **Cohesion**: how related are things inside one entity
SINGLE RESPONSIBILITY PRINCIPLE

- Software should do "one thing"
“UNCLE BOB”

- “Gather together the things that change for the same reasons. Separate those things that change for different reasons.”
Quick Preview: UML Class Diagrams

- We will discuss more in a forthcoming lecture
- Simplified class diagram to visually describe a system’s organization
- Inheritance
  - Generalization (class extends another class)
  - Realization (class implements an interface)
QUICK PREVIEW: UML CLASS DIAGRAMS (2)

- Interface representations

<table>
<thead>
<tr>
<th>&lt;&lt;Interface&gt;&gt;</th>
<th>Class &lt;&lt;interface&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>attributes</td>
</tr>
<tr>
<td>Operations/behaviors</td>
<td>Operations/behaviors</td>
</tr>
</tbody>
</table>
Does the following class adhere to the Single Responsibility Principle?
SINGLE RESPONSIBILITY PRINCIPLE

- For each method $X$, ask, "Does the Automobile have primary responsibility for $X$-ing?"
- If the answer is no, the method doesn't belong.
AUTOMOBILE: REFACTORED

**For space, focusing on behaviors**

Automobile:
- start()
- stop()
- receiveOil()

CarWash:
- wash(Automobile)

Mechanic:
- changeTires()
- checkOil()

Driver:
- drive(Automobile)
IF YOUR DESCRIPTION OF A CLASS HAS MULTIPLE “ANDS” THEN IT PROBABLY NEEDS TO BE REFACTORED.
OPEN/CLOSED PRINCIPLE

- Software entities should be open to extension but closed to modification
- We should be able to use software entities in new ways, without changing the code
  - Can "lock code in a vault"
  - But use for things we did not think of when we wrote it
class Bird {
  public:
  virtual void setLocation(double longitude, double latitude) = 0;
  virtual void setAltitude(double altitude) = 0;
  virtual void draw() = 0;
};

VERSION 1: IS A SUCCESS!
BUT WHAT IF WE USED IF/ELSE FOR BIRDS?

- What if we want to check the bird type in 100 places?
- What if we want to add Hawk with this structure?

```java
void someMethod(Bird b) {
    if (b.getType() == Bird.PIGEON) {
        // pigeon code
    }
    else if (b.getType() == Bird.PARROT) {
        // parrot code
    }
    else if (b.getType() == Bird.EAGLE) {
        // eagle code
    }
    // ...
}
```

If/Else or switch are the enemies of Open/Closed
WHAT SHOULD WE DO INSTEAD?

- Dynamic dispatch and polymorphism
VERSION 2: ADDS A HAWK AND 10 OTHER BIRDS WITH NO ISSUE

Open/Closed Principle Satisfied
VERSION 3: ADDS A PENGUIN

DO YOU EXPECT ANY ISSUE?
Bird behavior when you run the program

Pigeon

Parrot

Penguin
WHY ARE PENGUINS FLOPPING?

Penguins don’t fly! All the penguin objects ignore the attempts of the user to make them to fly.

```cpp
#include <iostream>

class Bird {
public:
    virtual void setLocation(double longitude, double latitude) = 0;
    virtual void setAltitude(double altitude) = 0;
    virtual void draw() = 0;
};
```
Because the Penguin subclass violates the flying assumption, it does not satisfy the Liskov Substitution Principle for the Bird superclass.
LISKOV SUBSTITUTION

- Ability to replace any instance of a superclass with an instance of one of its subclasses without negative side effects
- Not the same as subtype polymorphism because “not breaking the program” is a requirement
### Preconditions & Post-conditions

<table>
<thead>
<tr>
<th><strong>Preconditions</strong>: things that are supposed to be true before the method is called</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Must not be strengthened in a subclass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Post-conditions</strong>: things that are true after the method (assuming preconditions were satisfied)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Must not be weakened in a subclass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Invariants</strong>: must be valid before and after a method call</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Invariants of the superclass must be preserved in the subclass</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>History</strong>: observable states of the object</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Any history must be valid for the superclass</td>
</tr>
</tbody>
</table>
Subclass should not strengthen preconditions

Stated Differently: Requirements on inputs to a subclass cannot be stricter than its base class

If attempt bird.eat(worm), then raises an error

Violates LSP because Pigeon expectations for input (only bread) stricter than Bird

- eat()
- layEgg()
- fly()

---

//override eat method, to only eat bread

Pigeon
- Postconditions must not be weakened in a subtype.
- Stated Differently: **Possible outputs from a subclass must be more than or equally as restrictive as from the base class.**
- If attempt `bird.hatch()` exception raised because no return in alien subclass.
- Violates LSP because there are less possible outputs. So post conditions are weakened.
INVARIENTS MUST ALWAYS BE TRUE

Example Bird Invariant:
If $isFlying()$ is true, then $get\ Altitude() > 0$

All subclasses must ensure this relationship too!
substitutable

NOT substitutable

Source: Baniassad, 2018
Egg  →  Hatching  →  Baby

Adult  →  Parenting  →  Egg

A sub-class that can go from Adult->Baby Violates LSP
BARBARA LISKOV

MIT INSTITUTE
PROFESSOR AND
HEAD OF THE
PROGRAMMING
METHODOLOGY GROUP
A client shouldn’t be forced to depend on interfaces they don’t use

Better to have many small interfaces than a few very large interfaces

“Too many is better than too few”
<<Automobile>>

- start()
- stop()
- receiveOil()
- topDown()
- airControl()

Ferrari

Fiat 500
<<Automobile>>

- start()
- stop()
- receiveOil()
- topDown()
- airControl()
- turbo() { //override turbo so it’s slow }

Ferrari

Fiat 500
<<Automobile>>

- start()
- stop()
- receiveOil()

<<Convert>>

- topUp()
- topDown()

<<Speed>>

- turbo()
DEPENDENCY INVERSION PRINCIPLE

- HIGH LEVEL OBJECTS SHOULD NOT DEPEND ON LOW LEVEL IMPLEMENTATIONS
- Higher classes are not dependent on the lower classes instead depend upon the abstraction of the lower classes
Manager always need to have concern over who employees are and what they do.
The manager doesn’t have an idea beforehand about all the type of workers that may come under him/her.
DEPENDENCY INJECTION

ONE OBJECT SUPPLIES THE DEPENDENCIES OF ANOTHER OBJECT
Does FlyingCreatureHouse follow Dependency Inversion?
  - Does it depend on an interface or a concrete implementation?

What about Open/Closed Principle?
Does BirdHouse follow Dependency Inversion? Depends on a concrete implementation (CardinalBird)
- Does it depend on an interface or a concrete implementation?
- What about Open/Closed Principle? No: cannot make this work with other bird types!
Does BirdHouse follow Dependency Inversion?  Depends only on FlyingCreature interface

Does it depend on an interface or a concrete implementation?

What about Open/Closed Principle?  Yes, can pass in any type of FlyingCreature
DESIGNING FOR TESTING

- **Controllability**: how well can my test code control the state of the module I want to test?

- **Observability**: how well can my test code inspect the state (to see if correct) of the module I want to test?

- **Isolatability**: how well can I separate out the module I want to test it by itself?
TESTING AND SRP

HOW DOES FOLLOWING THE SINGLE RESPONSIBILITY PRINCIPLE HELP WITH TESTING?