CSE 3902: High-Quality Software

Justin Holewinski

The Ohio State University

What Makes Software “High-Quality”?

Does it work?

- Not good enough

Key metrics for this course:

- SOLID
- Coupling
- Cohesion

Key ideas:

- Simplicity
- Readability
- Maintainability
- Reusability

SOLID

- **S**ingle-Responsibility Principle
  - Every class has one responsibility/purpose
- **O**pen-Closed Principle
  - "Open for extension, closed for modification"
- **L**iskov Substitution Principle
  - aka Design By Contract
  - Users of a reference to a base class must be able to use a reference to a derived class
- **I**nterface Segregation Principle
  - Do not force clients to depend on interfaces they do not use
- **D**ependency Inversion Principle
  - "Depend upon abstractions, [not] concretions"

For more information: https://en.wikipedia.org/wiki/SOLID

Coupling

How much are components dependent on each other's details?

- If I want to draw the player sprite, do I need to know how the sprite rendering component is implemented?

High Coupling:

```
Player player = game.Player;
DrawSprite(player.X, player.Y, player.Width, player.Height,
           player.AnimationFrame);
```

Low Coupling:

```
ISprite playerSprite = game.GetPlayerSprite();
playerSprite.Draw();
```

Cohesion

How well does the code keep to its specific purpose?

High Cohesion:

```
public void MovePlayer(Vector2 change)
{
    this.Position += change;
}
```

Low Cohesion:

```
public void DoEverything()
{
    MovePlayer();
    ShootEnemy();
    PlaySound();
}
```

Simplicity

Software is inherently complex

- Lots of code
- Lots of features
- Lots of dependencies

Simplicity refers to individual components

- A simple component has one element, one purpose

Build *complex* software by combining *simple* components

Readability

Would a reasonable programmer be able to understand your code?

Code Style

- Is it easy to understand?
- Do you use self-explanatory variable names?
- Is it *consistent*?
- Are you decomposing the control flow in a logical, easy-to-understand way?

```
public class Something
{
    public void
  DoTheThing(Player abc) {
abc.foo();
            abc.Move();
}
}
```

Readability

How familiar is the team with the programming language?

Do not assume everyone on the team is an expert

- Some teammates may be learning the language for the first time
- Maybe avoid new, complicated language features in such cases

C# Examples

- Lambdas
- LINQ
- Pinning memory

Maintainability

How easy is it to fix bugs and extend your software?

- Add new feature for version N+1
- Replace component with updated version, with minimal code changes
- Adapt to new environment or platform
- Patch a security vulnerability

Reusability

Opt for creating and using reusable software components

- Design to a simple interface, not the specific program
- If you've already written a component for something, use it!
- Avoid Not Invented Here (NIH) syndrome

But also do not go too far out of your way for hypotheticals

- "I could spend an extra month to support this extra feature..."

Design Patterns

The purpose of *design patterns* is to help with all of these metrics

A *design pattern* provides a well-known template of a solution for a specific software problem

- Each pattern solves a particular type of problem
- The pattern name can be used to communicate with other engineers
    - "Let's use a *factory* for this!" instead of "Let's create a class that can construct instances of other classes by using ..."

We'll investigate several patterns during this course