# CSE 3902: Misc Software Topics

Justin Holewinski

The Ohio State University

# Amdahl's Law

How do we assess optimization potential of code?

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

| | |
|---|---|
| $S_{latency}(s)$ | Theoretical improvement ratio |
| $p$ | Ratio of program execution time for section being optimized |
| $s$ | Improvement ratio from optimization |

https://en.wikipedia.org/wiki/Amdahl%27s_law

# Amdahl's Law Example

Simple Program

```
Function1();  // 10ms
Function2();  // 25ms
Function3();  // 15ms
```

What if we double the execution speed of `Function2`?

$$
\begin{aligned}
p &= \frac{25}{10+25+15} \\
&= \frac{25}{50} \\
&= 0.5 \\
s &= 2 \\
S_{latency}(s) &= \frac{1}{(1-p)+\frac{p}{s}} \\
&= \frac{1}{(1-0.5)+\frac{0.5}{2}} \\
&= \frac{1}{0.5+0.25} \\
&= 1.33
\end{aligned}
$$

# Code Smells

A *code smell* is an indicator of a design problem in your code

- Does not affect functionality (not a bug), but may lead to future bugs
- Leads to many issues
  - Hard to extend code
  - Hard to debug code
  - Hard to reason about code

# Code Smells

Common smells

- *Mysterious Name*: names that do not convey a meaning
- *Contrived Complexity*: use of unnecessarily-complex design patterns
- *Large Class*: class that does too much
- *Long Method*: method that does too much
- *Magic Constants*: hard-coded immediates instead named constants
- *Too Many Parameters*: lots of function parameters

More information: https://en.wikipedia.org/wiki/Code_smell

# Technical Debt

*Technical debt* is the future cost associated with design and coding choices, often made due to time or business constraints. This debt usually has a direct impact on the maintainability and overall quality of a software project.

Causes:

- Time crunch
- Deferred refactoring
- Coupling
- Lack of engineer experience
- No clear leadership

Tackling technical debt:

- Fixing ineffective design patterns
- Taking longer to find and fix bugs
- Refactoring to make code more maintainable

# Refactoring

*Refactoring* is the process of cleaning up your code by making non-functional changes.

Examples:

- Replacing object creation with factory
- Introducing commands between initiator and receiver
- Modifying source formatting/style
- Splitting a large class into smaller classes
- Removing magic numbers

The goal of refactoring is to improve the maintainability of your code.

Warning: refactoring can lead to bugs! Ensure you have a good test plan for before and after refactoring to ensure functionality is unaffected.

# Refactoring

```
public void LoadLevel()
{
  // ...
  foreach (EnemyData data in level.EnemyData)
  {
    if (data.Type == "goomba")
    {
      Texture2D tex =
          Content.Load<Texture2D>(/*...*/);
      Vector2 position =
          data.StartPosition + WorldOffset;
      enemies.Add(new Goomba(tex, position));
    }
    else if (data.Type == "koopa")
    {
      Texture2D tex =
          Content.Load<Texture2D>(/*...*/);
      Vector2 position =
          data.StartPosition + WorldOffset;
      enemies.Add(new Koopa(tex, position));
    }
  }
}
```

```
public void LoadLevel()
{
  // ...
  foreach (EnemyData data in level.EnemyData)
  {
    Vector2 position = data.StartPosition + WorldOffset;
    enemies.Add(EnemyFactory.Get(data.Type, position));
  }
}
```

Why is this code “better”? Think about:

- Coupling
- Cohesion
- Maintainability
- Readability