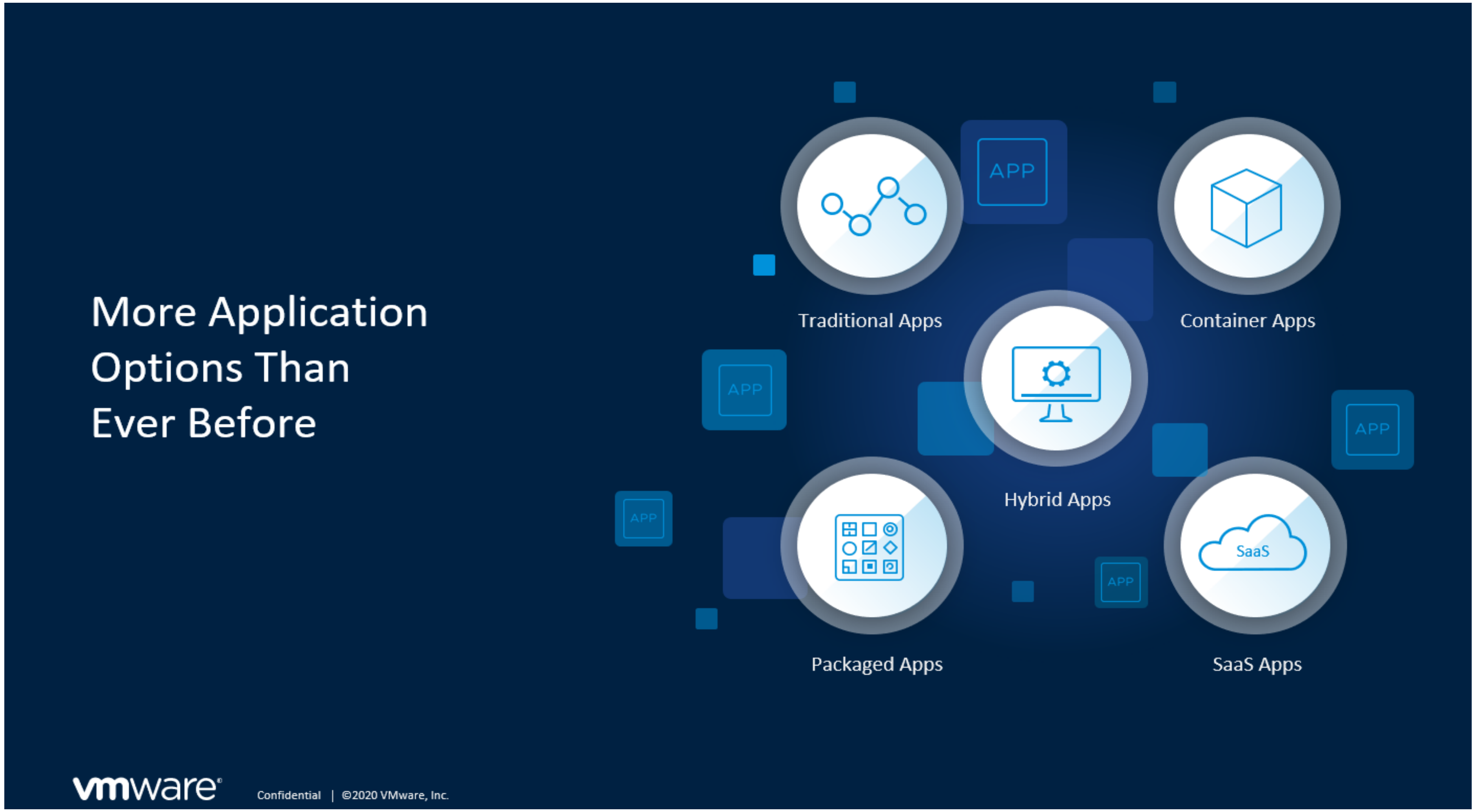
VMWare Launch - Working with Cloud

Application Options - SOLID

Allen R. Sanders
Senior Technology Instructor

Class Roadmap



More Application Options Than Ever Before

APP
Traditional Apps
Container Apps
Hybrid Apps
Packaged Apps
SaaS
SaaS Apps
vmware
Confidential | ©2020 VMware, Inc.

Architecting for the Future

- When we architect and build an application at a “point in time”, we hope that the application will continue to be utilized to provide the value for which it was originally built

- Since the “only constant is change” (a quote attributed to Heraclitus of Ephesus), we have to expect that the environment in which our application “lives and works” will be dynamic

- Change can come in the form of business change (change to business process), the need to accommodate innovation and ongoing advancement in technology

- Often, the speed at which we can respond to these changes is the difference between success and failure

Architecting for the Future

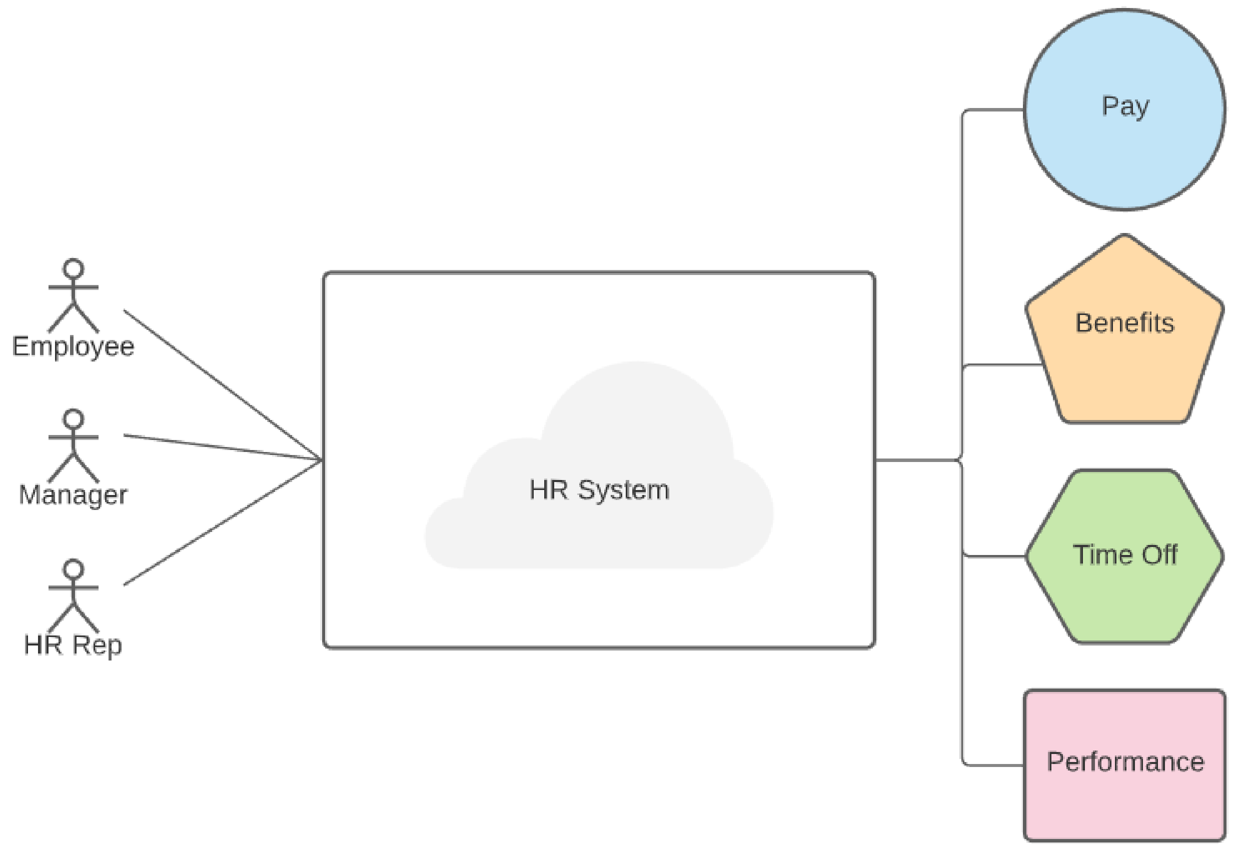
In order to ensure that we can respond to change "at the speed of business", we need to build our systems according to best practices and good design principles:

- ➢ Business-aligned design
- ➢ Separation of concerns
- ➢ Loose coupling
- ➢ Designing for testability

Business-Aligned Design

- Build systems that use models and constructs that mirror the business entities and processes that the system is intended to serve
- Drive the design of the system and the language used to describe the system based around the business process not the technology
- AKA Domain Driven Design
- Results in a system built out of the coordination and interaction of key elements of the business process – helps to ensure that the system correlates to business value
- Also helps business and technology stakeholders keep the business problem at the forefront

Exercise – Domain Driven Design



- Exercise 1 (10 min) – List as many business (i.e. non-technical) terms that you can think of to define this system
- Exercise 2 (10 min) – List the technology components that you would expect to see included in the solution

Separation of Concerns

- Break a large, complex problem up into smaller pieces
- Drive out overlap between those pieces (modules) to keep them focused on a specific part of the business problem and minimize the repeat of logic
- Logic that is repeated, and that might change, will have to be changed in multiple places (error prone)
- Promotes high cohesion and low coupling (which we will talk about in a minute)
- Solving the problem becomes an exercise in "wiring up" the modules for end-to-end functionality and leaves you with a set of potentially reusable libraries

Loose Coupling

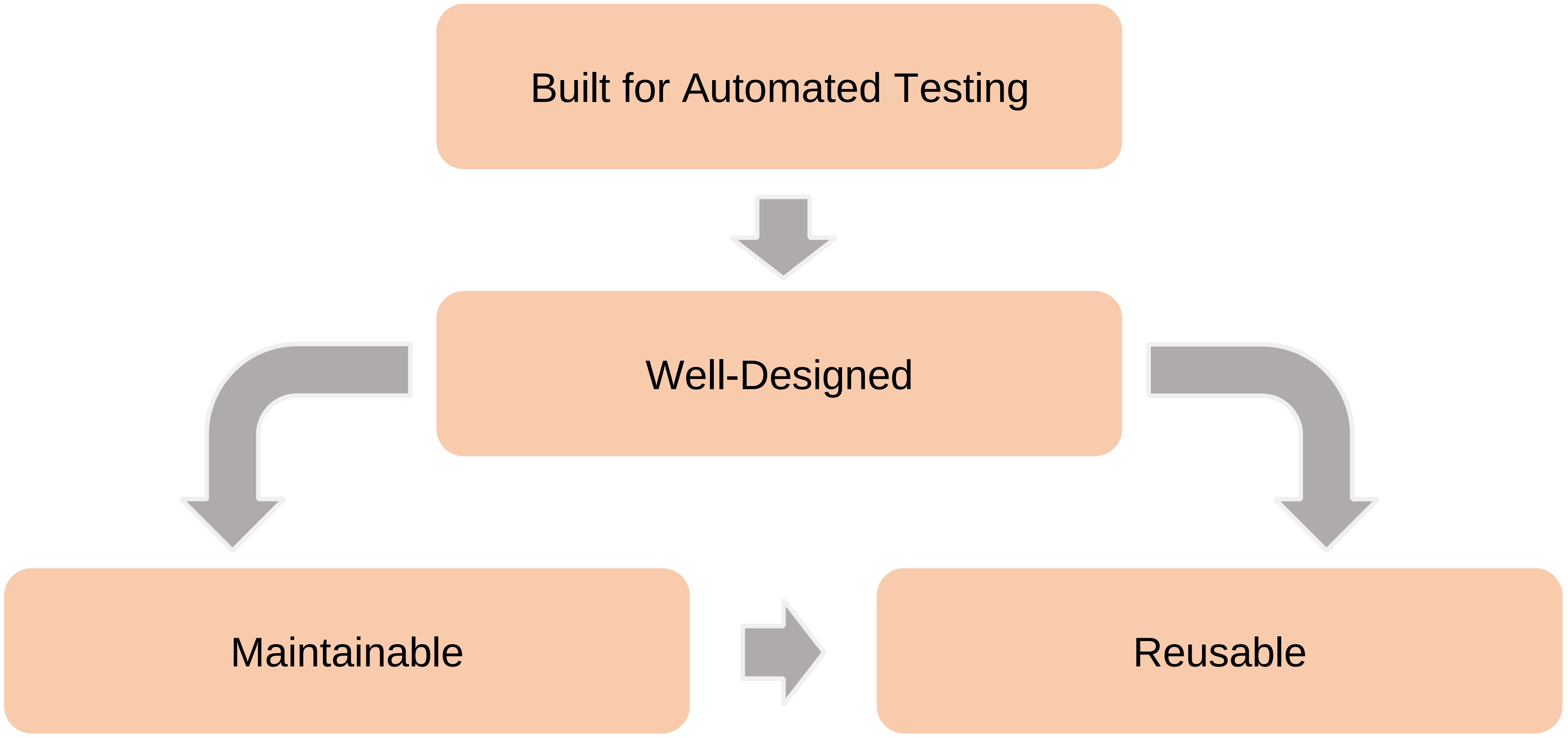
- Coupling between components or modules in a system causes problems, especially for maintaining the system over time
- System components that are tightly coupled, are more difficult to change (or enhance) – changes to one part of the system may break one or more other areas
- With coupling, you now must manage the connected components as a unit instead of having the option to manage the components in different ways (e.g. production scalability)
- Makes the job of unit testing the components more difficult because tests must now account for a broader set of logic and dependencies (e.g. tight coupling to a database makes it difficult to mock)

Designing for Testability

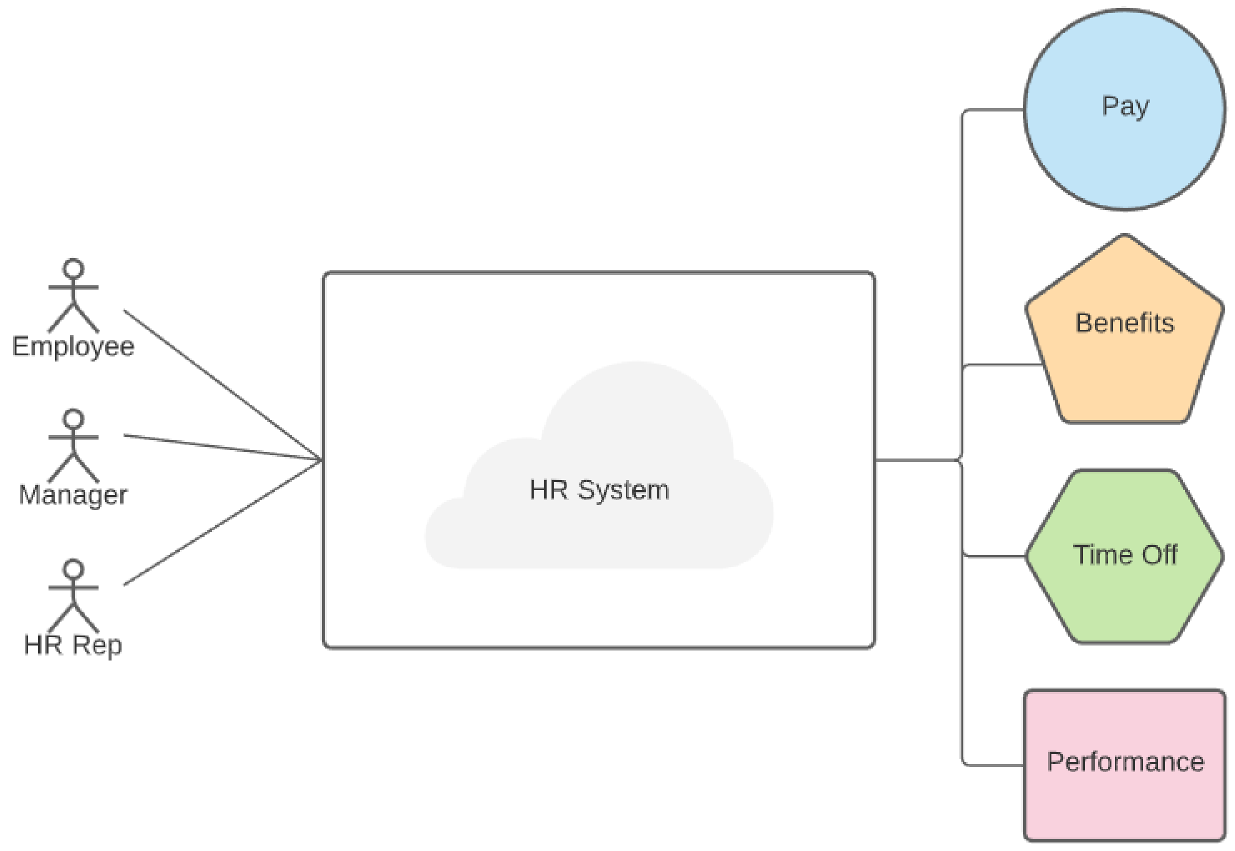
- Practicing the previous principles helps lead to a system that is testable
- Testability is important because it is a key enabler for verifying the quality of the system – at multiple stages along the Software Development Lifecycle (SDLC)
- When building a system, quality issues become more expensive to correct the later they are discovered in the development lifecycle – good architecture practices help you test early and often
- Ideally, testing at each stage will be automated as much as possible in support of quickly running the tests as and when needed

Designing for Testability

Testable code is…



Exercise – Designing for Testability



- Exercise 1 (10 min) – What kind of technology or code-level tests would you expect to see with this system?
- Exercise 2 (10 min) – What kind of business or end user-level tests would you expect to see with this system?

SOLID Principles

SOLID principles help us build testable code

- ➢ Single Responsibility Principle (SRP)
- ➢ Open-Closed Principle (OCP)
- ➢ Liskov Substitution Principle (LSP)
- ➢ Interface Segregation Principle (ISP)
- ➢ Dependency Inversion Principle (DIP)

Glossary

Before delving deeper into each principle, let's define some terms

- Client – A class, component or system that leverages another, separately-defined library or component for design simplification and reuse
- Encapsulation – Object-oriented design tenet that dictates that a class should hide & protect its state, providing a controlled interface for interacting with or changing that state (if required)
- Inheritance – Object-oriented design tenet that enables the building of "is a" hierarchies of related functionality through logical grouping and class reuse (including the ability to extend that functionality)

Glossary

Before delving deeper into each principle, let’s define some terms

- Polymorphism – Means “many forms”; object-oriented design tenet that uses context to determine functionality dynamically at runtime, accounting for the type of “thing” against which the functionality is being exercised

- Refactoring – Improving the internal structure of existing code without changing its external behavior or operation

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

A system module or component should have only one reason to change

Single Responsibility Principle (SRP)

SINGLE RESPONSIBILITY PRINCIPLE

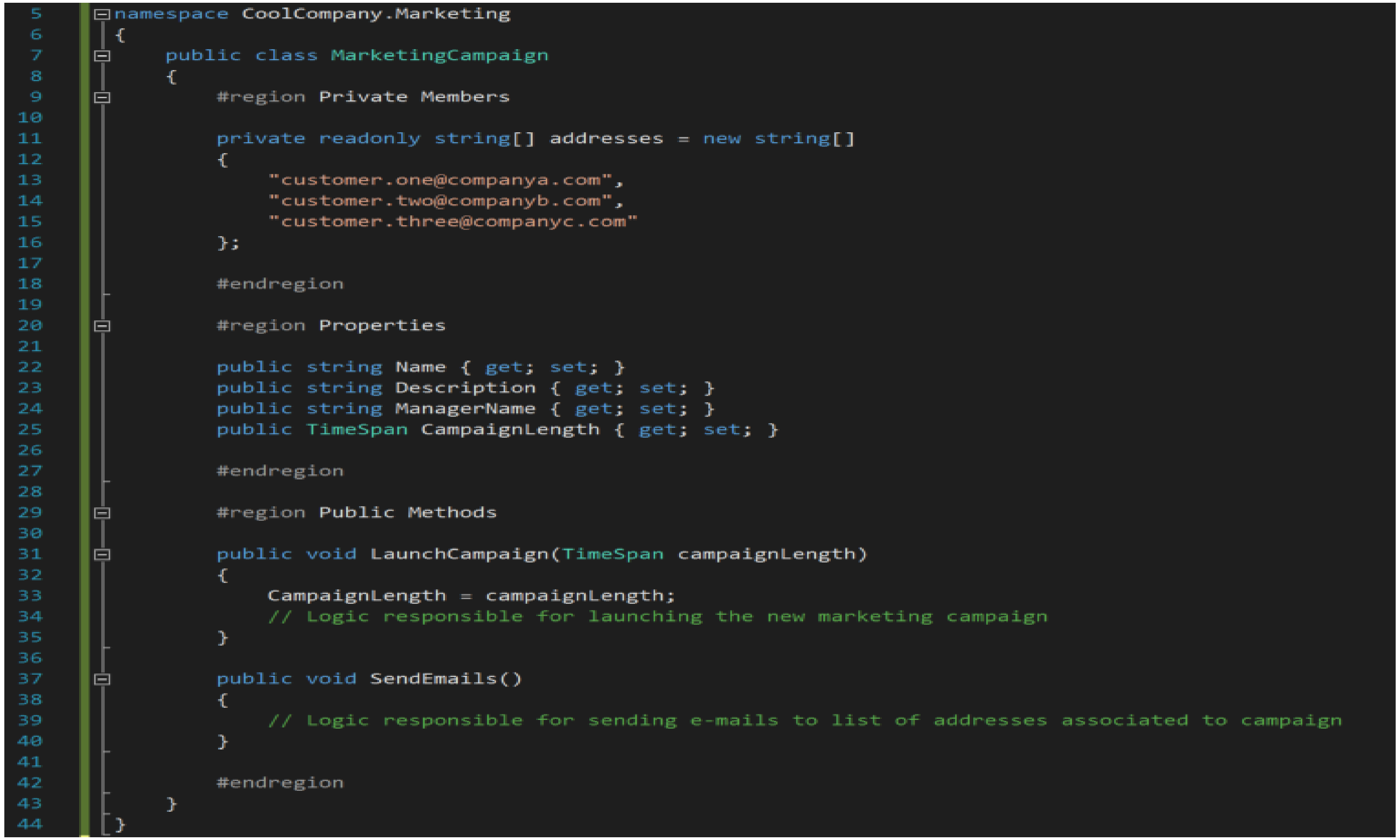
Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

Single Responsibility Principle (SRP)

Why important?

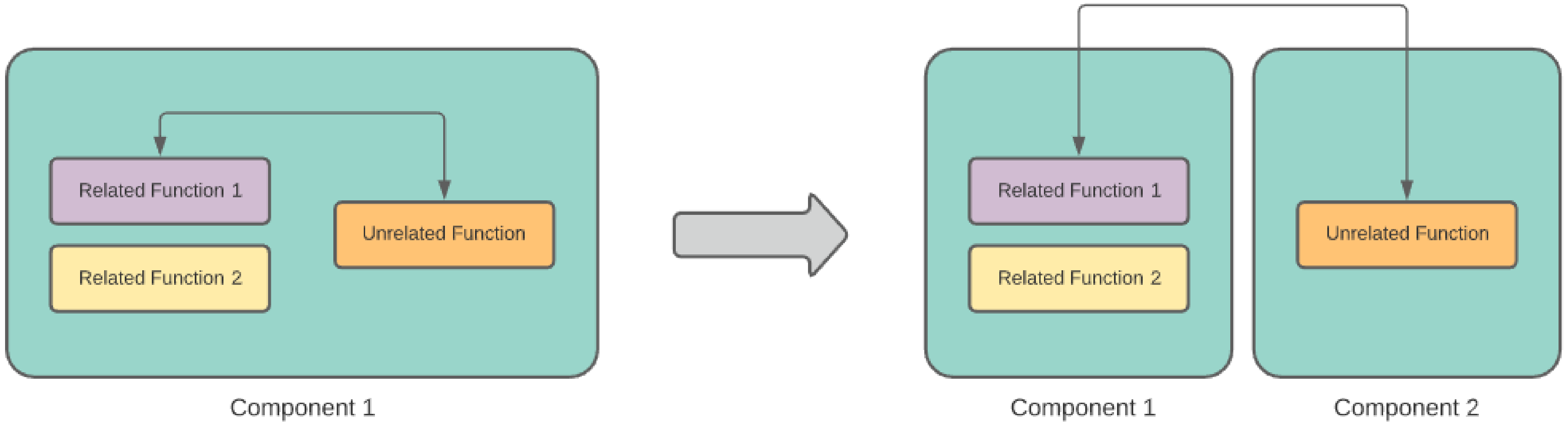
- Related functions organized together breed understanding (logical groupings) – think ***cohesion***
- Multiple, unrelated functionalities slammed together breed coupling
- Reduced complexity (cleaner, more organized code)
- Promotes smaller code modules
- Reduced regression – tension of change

Single Responsibility Principle (SRP)



```
namespace CoolCompany.Marketing
{
    public class MarketingCampaign
    {
        #region Private Members

        private readonly string[] addresses = new string[]
        {
            "customer.one@companya.com",
            "customer.two@companyb.com",
            "customer.three@companyc.com"
        };

        #endregion

        #region Properties

        public string Name { get; set; }
        public string Description { get; set; }
        public string ManagerName { get; set; }
        public TimeSpan CampaignLength { get; set; }

        #endregion

        #region Public Methods

        public void LaunchCampaign(TimeSpan campaignLength)
        {
            CampaignLength = campaignLength;
            // Logic responsible for launching the new marketing campaign
        }

        public void SendEmails()
        {
            // Logic responsible for sending e-mails to list of addresses associated to campaign
        }

        #endregion
    }
}
```

Single Responsibility Principle (SRP)



Single Responsibility Principle (SRP)

How to practice?

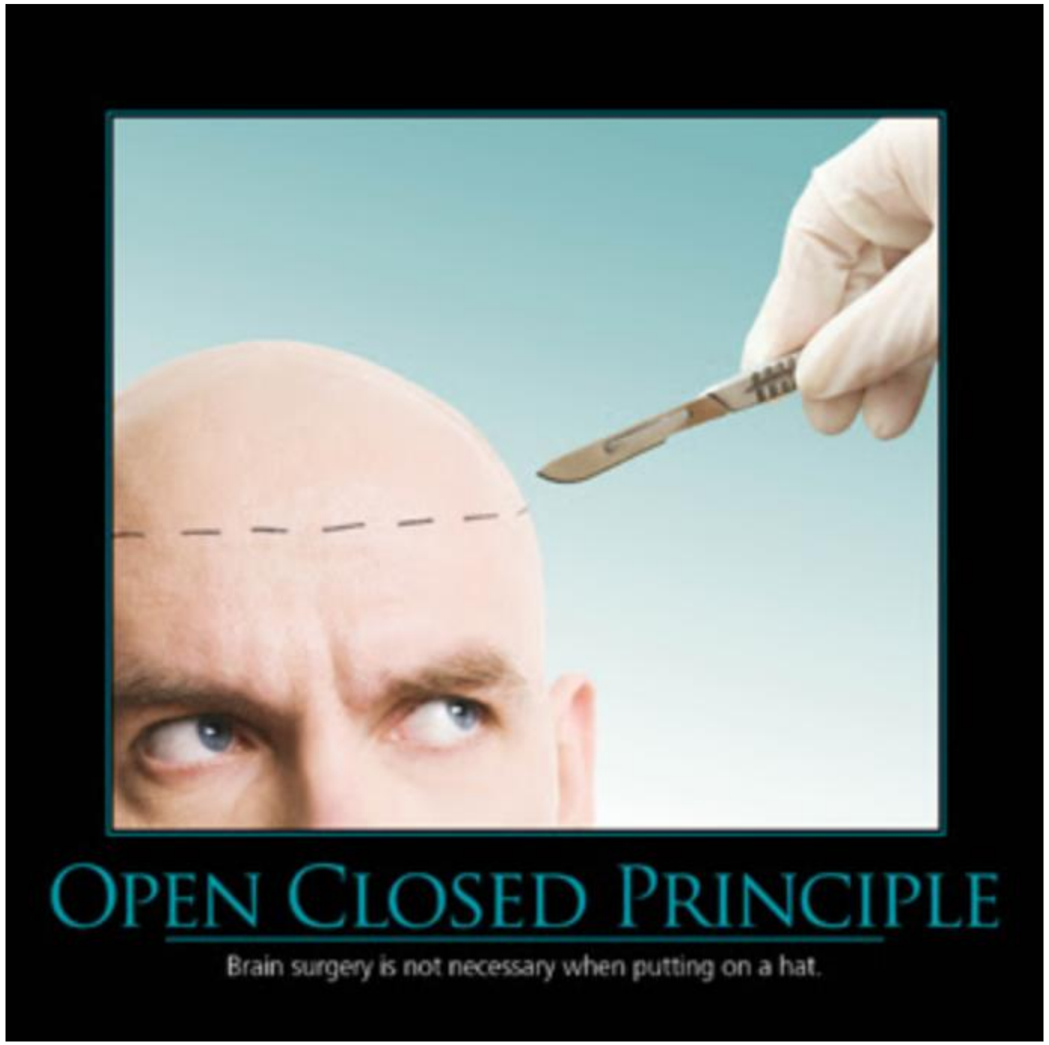
- When building new modules (or refactoring existing), think in terms of logical groupings
- Look for “axes of change” as points of separation
- Build new modules to take on new entity or service definitions – provides abstraction
- Structure clean integrations between separated modules
- Use existing tests (or build new ones) to verify a successful separation

Open-Closed Principle (OCP)

Open-Closed Principle (OCP)

Software entities should be open for extension but closed for modification

Open-Closed Principle (OCP)



Open-Closed Principle (OCP)

Why important?

- Our systems need to be able to evolve
- We need to be able to minimize the impact of that evolution

Open-Closed Principle (OCP)

How to practice?

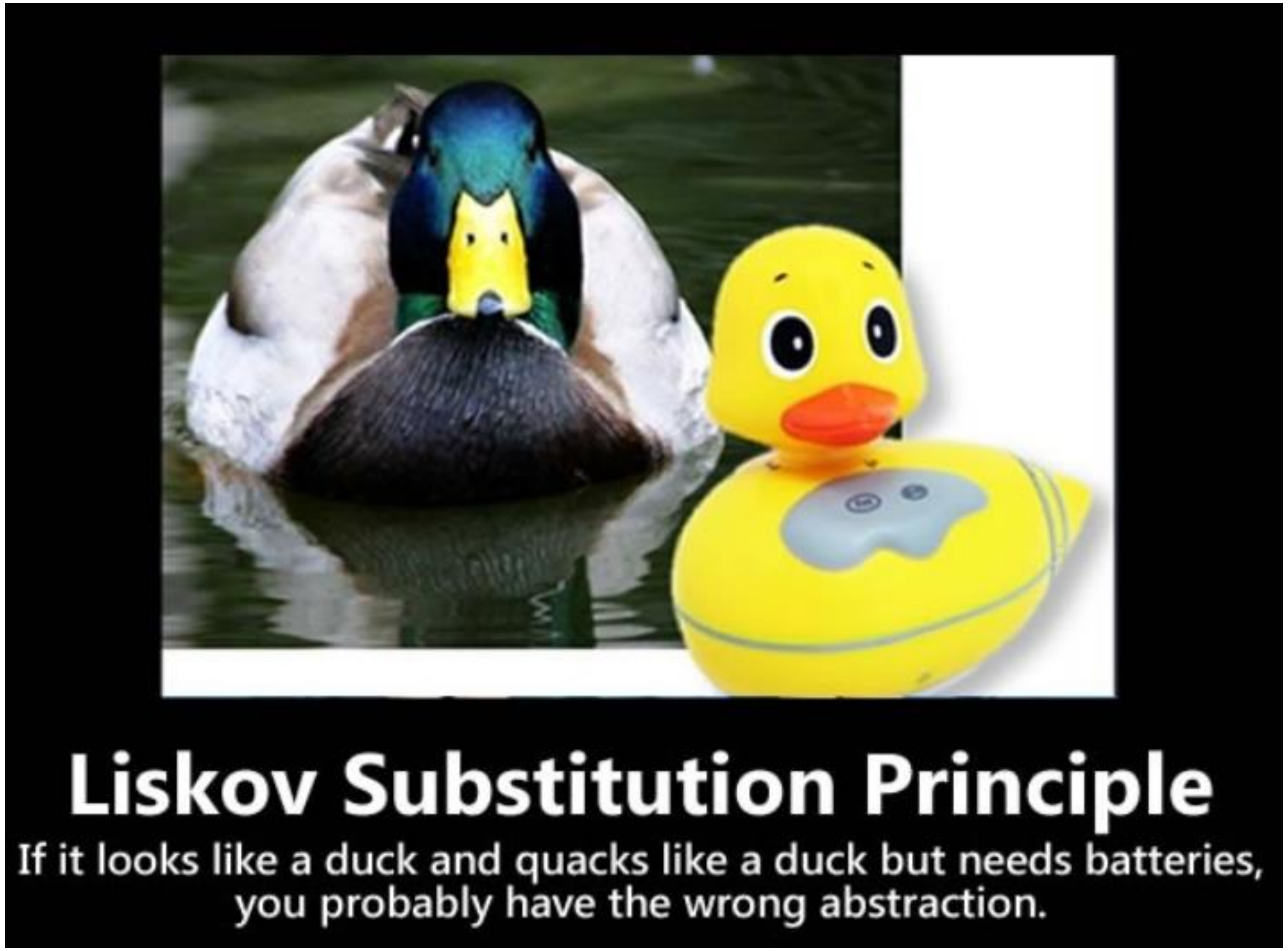
- When building new modules (or refactoring existing), leverage abstractions
- Use the abstractions as levers of extension
- Use existing tests (or build new ones) to verify the abstractions

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types

Liskov Substitution Principle (LSP)


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

Liskov Substitution Principle (LSP)

Why important?

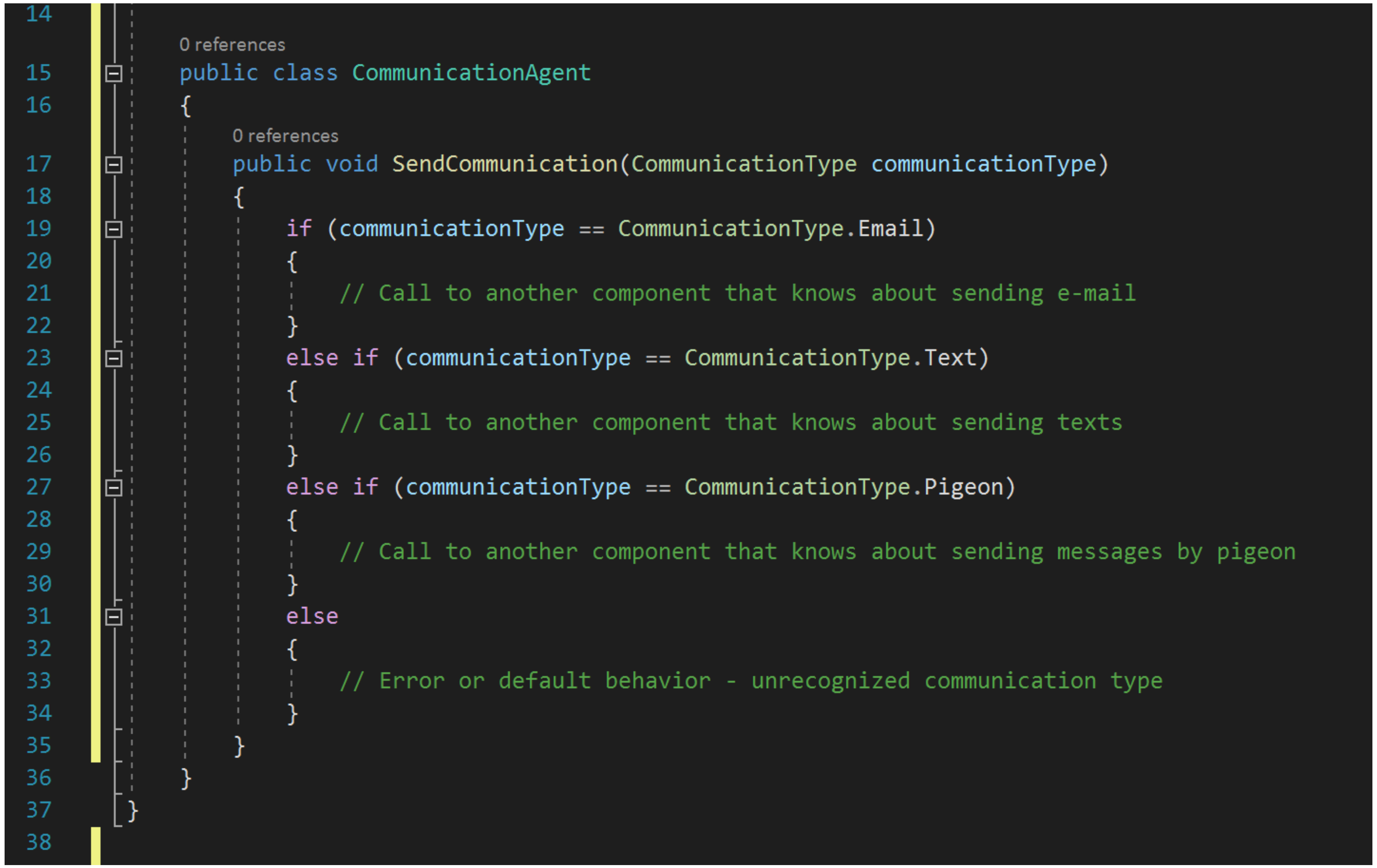
- Want to be able to use the abstractions created by OCP to extend existing functionality (vs. modify it)
- Especially useful when multiple variants of a type need to be processed as a single group
- Promotes looser coupling between our modules
- Without it, we may have to include if/else or switch blocks to route our logic
- Or keep adding parameters/properties for new but related types

Liskov Substitution Principle (LSP)

How to practice?

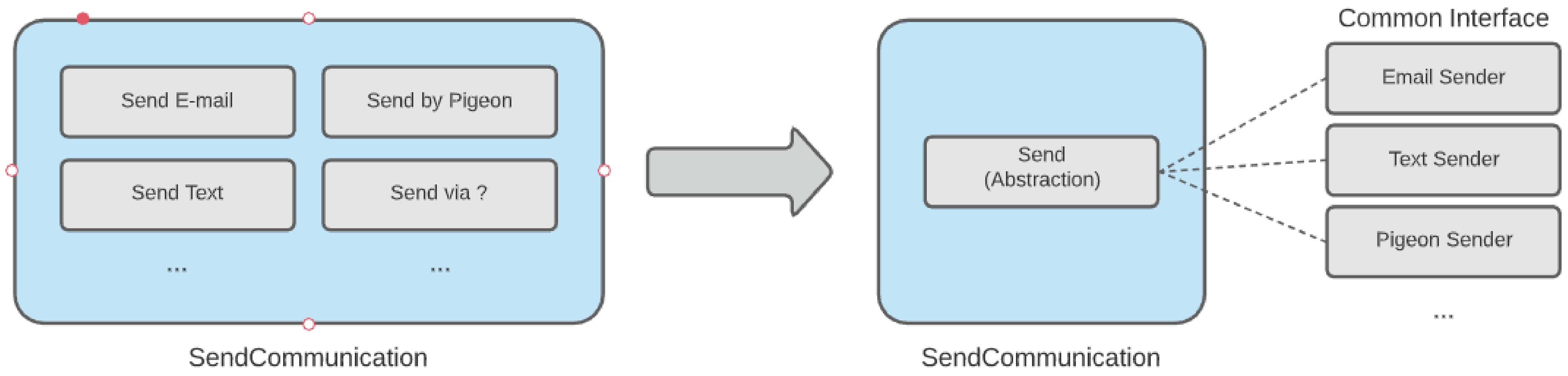
- Module members should reference the abstractions in consuming code
- Look for ways to encapsulate type-specific (or type-aware) logic in the type instead of in the code using the type
- Use existing tests (or build new ones) to verify our ability to effectively substitute

OCP & LSP



```
0 references
public class CommunicationAgent
{
    0 references
    public void SendCommunication(CommunicationType communicationType)
    {
        if (communicationType == CommunicationType.Email)
        {
            // Call to another component that knows about sending e-mail
        }
        else if (communicationType == CommunicationType.Text)
        {
            // Call to another component that knows about sending texts
        }
        else if (communicationType == CommunicationType.Pigeon)
        {
            // Call to another component that knows about sending messages by pigeon
        }
        else
        {
            // Error or default behavior - unrecognized communication type
        }
    }
}
```

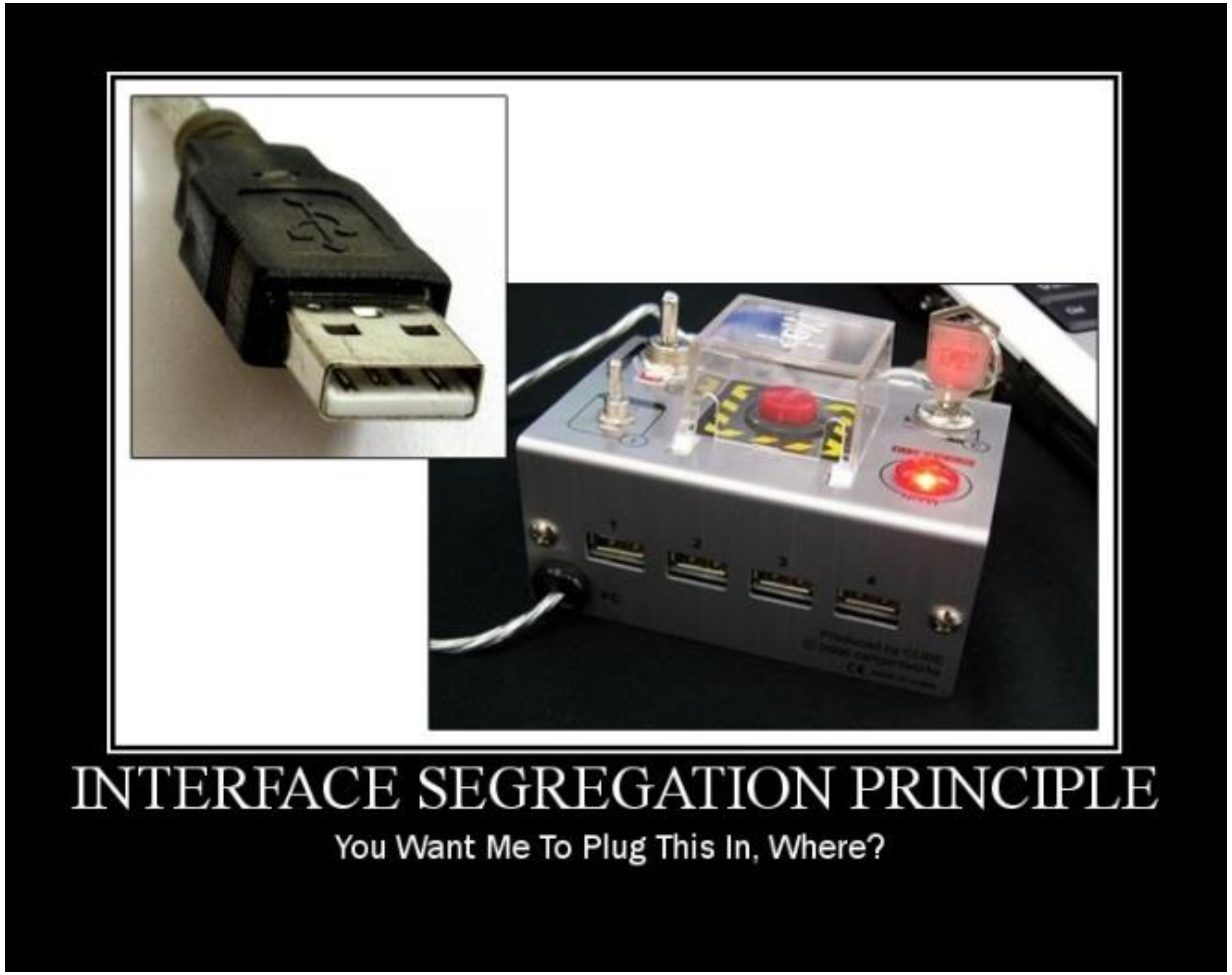
OCP & LSP


Send E-mail
Send by Pigeon
Send Text
Send via ?
...
...
SendCommunication
Send (Abstraction)
SendCommunication
Common Interface
Email Sender
Text Sender
Pigeon Sender
...

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use

Interface Segregation Principle (ISP)

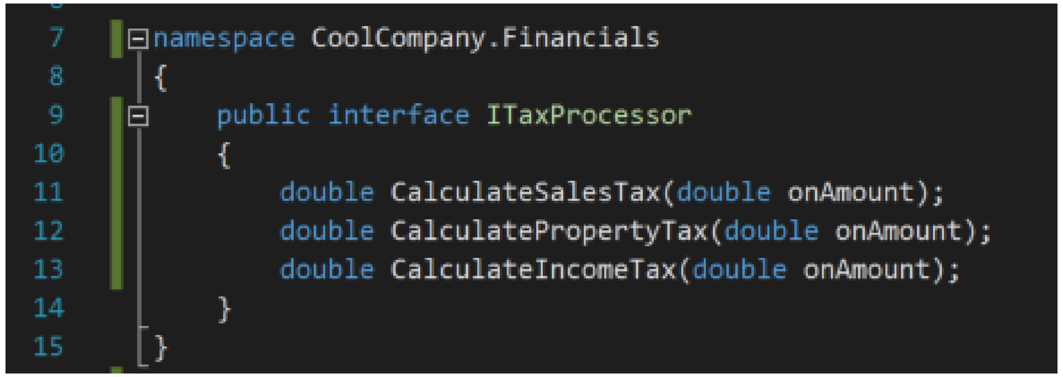

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle (ISP)

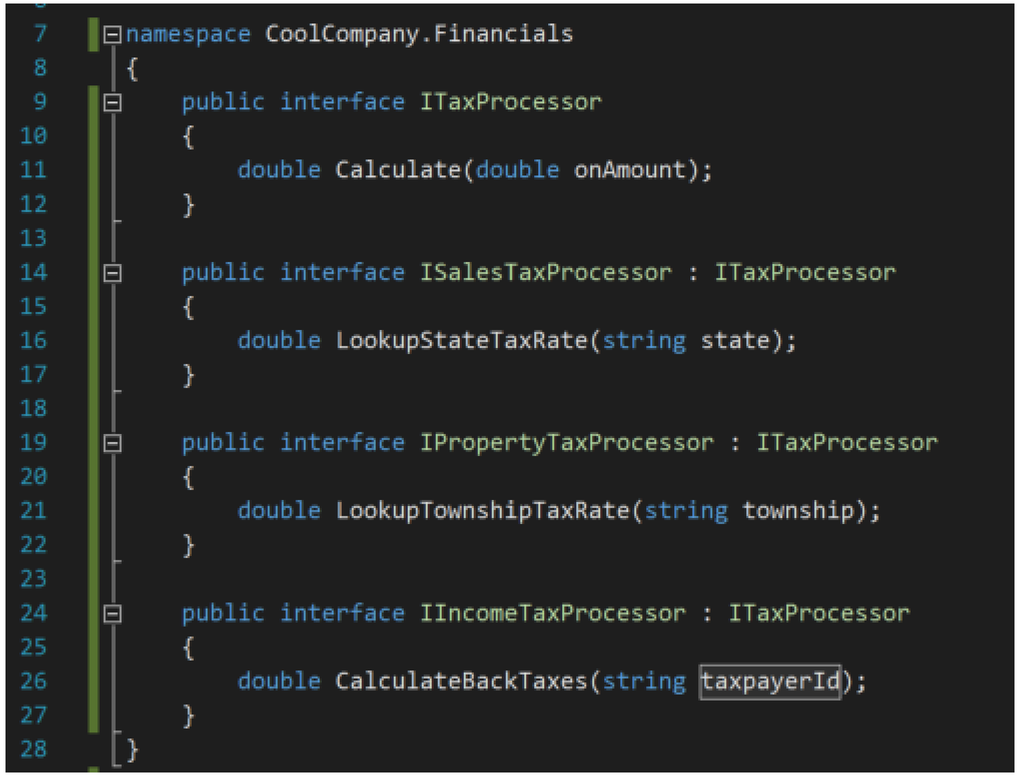
Why important?

- By following SRP, OCP and LSP, we can build a set of cohesive abstractions enabling reuse in multiple clients
- However, sometimes the abstractions we need are not cohesive (even though they may seem like it at first)
- We need a mechanism for logical separation that still supports combining functions together in a loosely-coupled way
- Otherwise, we'll see "bloating" in our abstractions that can cause unintended/unrelated impact during normal change

Interface Segregation Principle (ISP)



```
namespace CoolCompany.Financials
{
    public interface ITaxProcessor
    {
        double CalculateSalesTax(double onAmount);
        double CalculatePropertyTax(double onAmount);
        double CalculateIncomeTax(double onAmount);
    }
}
```



```
namespace CoolCompany.Financials
{
    public interface ITaxProcessor
    {
        double Calculate(double onAmount);
    }

    public interface ISalesTaxProcessor : ITaxProcessor
    {
        double LookupStateTaxRate(string state);
    }

    public interface IPropertyTaxProcessor : ITaxProcessor
    {
        double LookupTownshipTaxRate(string township);
    }

    public interface IIncomeTaxProcessor : ITaxProcessor
    {
        double CalculateBackTaxes(string taxpayerId);
    }
}
```

Interface Segregation Principle (ISP)

How to practice?

- In your abstractions, don't force functions together that don't belong together (or that you might want to use separately)
- Leverage delegation in the implementation of those abstractions to support variance
- Use OCP to bring together additional sets of features in a cohesive way (that still adheres to SOLID)
- Use existing tests (or build new ones) to verify aggregate features

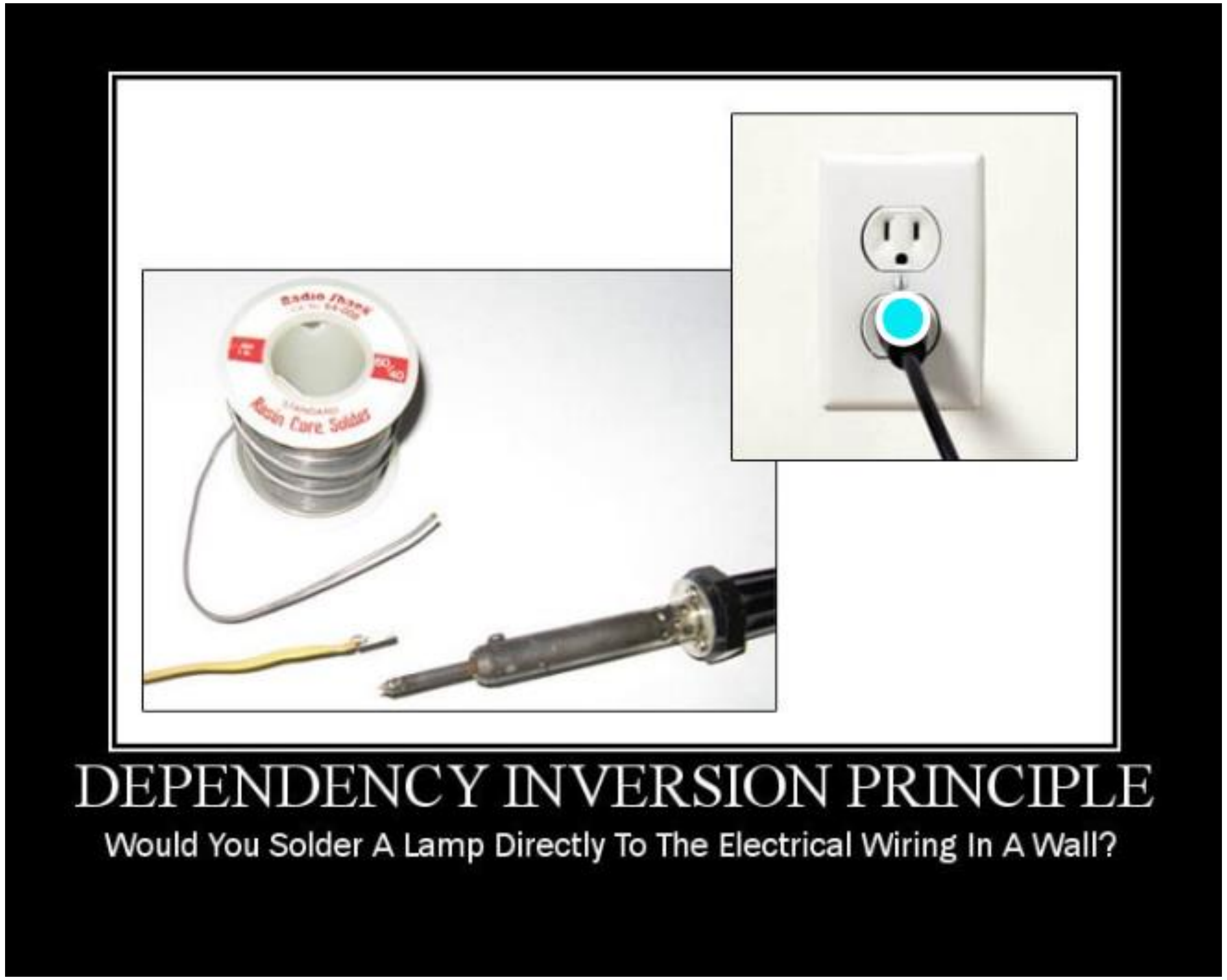
Dependency Inversion Principle (DIP)

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules – both should depend on abstractions

Abstractions should not depend upon details – details should depend upon abstractions

Dependency Inversion Principle (DIP)


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle (DIP)

Why important?

- As with all the SOLID principles, we want to build reusable but loosely-coupled code modules
- We don't want to limit reuse to lower-level utility classes only
- If higher-level modules are tightly-coupled to and dependent on low-level modules, change impact can cascade up
- The change impact can be transitive (flowing through multiple layers in-between)
- There are patterns and utility libraries built to enable

Dependency Inversion Principle (DIP)

How to practice?

- As with the other principles, use (and depend on) abstractions
- Use mechanisms like dependency injection and IoC (Inversion of Control) to build looser coupling between logic providers and consumers
- Build layering into your architectures and limit references to the same or immediately adjacent layer only
- Keep ownership of abstractions with the clients that use them or, even better, in a separate namespace/library
- Use existing tests (or build new ones) to verify functionality in each layer and use mocking techniques to isolate testing

So, What Does This Have to Do With the Cloud?

- Complex Cloud-enabled workflows are composed of multiple systems
- Those multiple systems are composed of multiple components
- Those multiple components are composed of multiple modules or classes
- Those multiple modules are composed of multiple routines or blocks of code
- To help ensure quality, testability, stability and maintainability of the complex Cloud-enabled workflows, we must employ good design and build practice in each building block (like a fractal)
- Furthermore, this requires forethought and intentionality – it doesn't happen by accident

Break (10 min.)

THANK YOU

Develop Intelligence