"SOLID Principles and Dependency Injection"

Session 4

September 30th 2025

Jesper Gulmann Henriksen

Beautiful, cartoonish illustrations by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Menti #0:

Which SOLID Principles do you already know (and love)?

SOLID is…

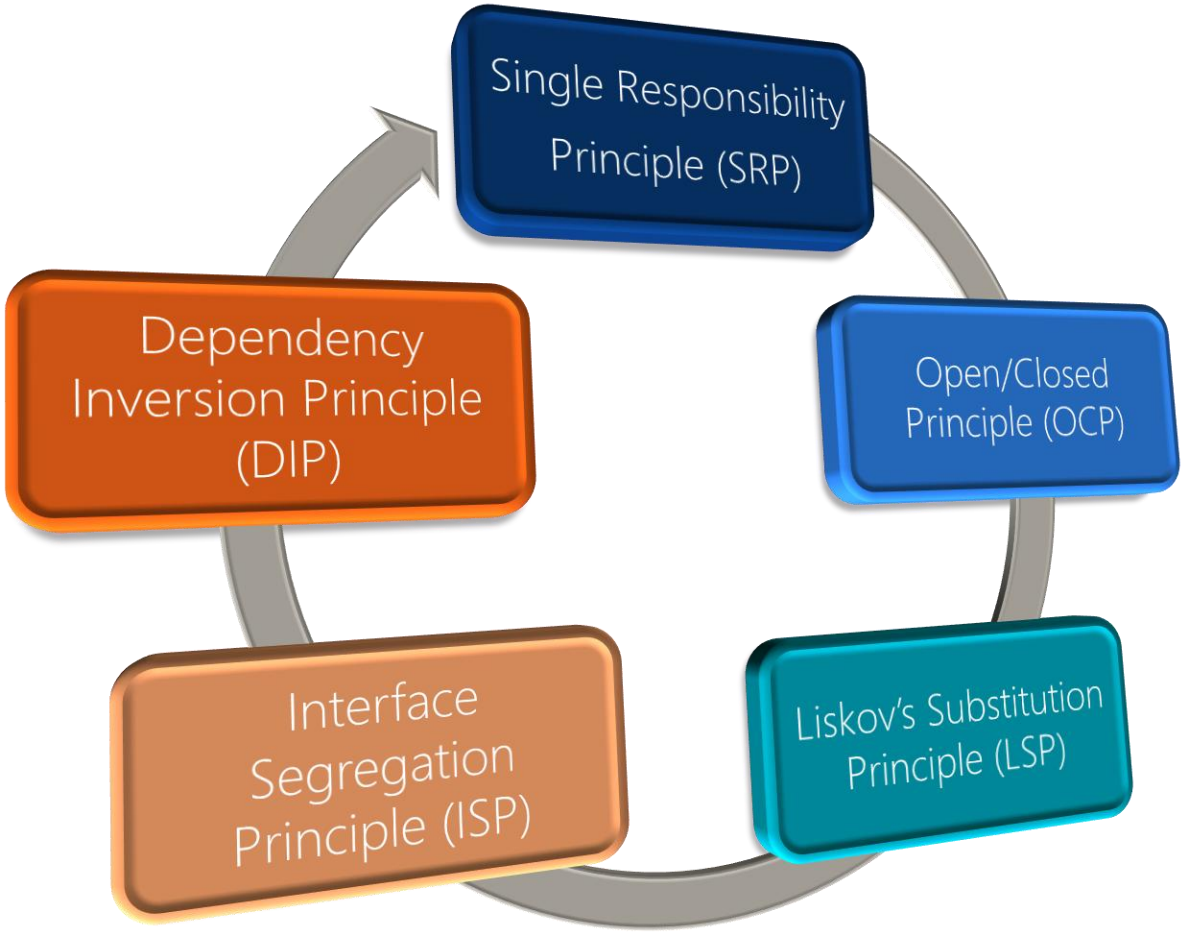
- … Five fundamental "commandments" for OOP
- … Coined by Robert C. Martin a.k.a. "Uncle Bob"
- … Programming language-agnostic
- … Not a framework or package!

- … Maintainability!

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live"

- John F. Woods (1991)

The Five Principles of SOLID



Agenda

- Introducing SOLID
- **Single Reponsibility Principle (SRP)**
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Single Responsibility Principle (SRP)

Each class should only have a single responsibility.

Each class should have only one reason to change

What Does That Mean Exactly?

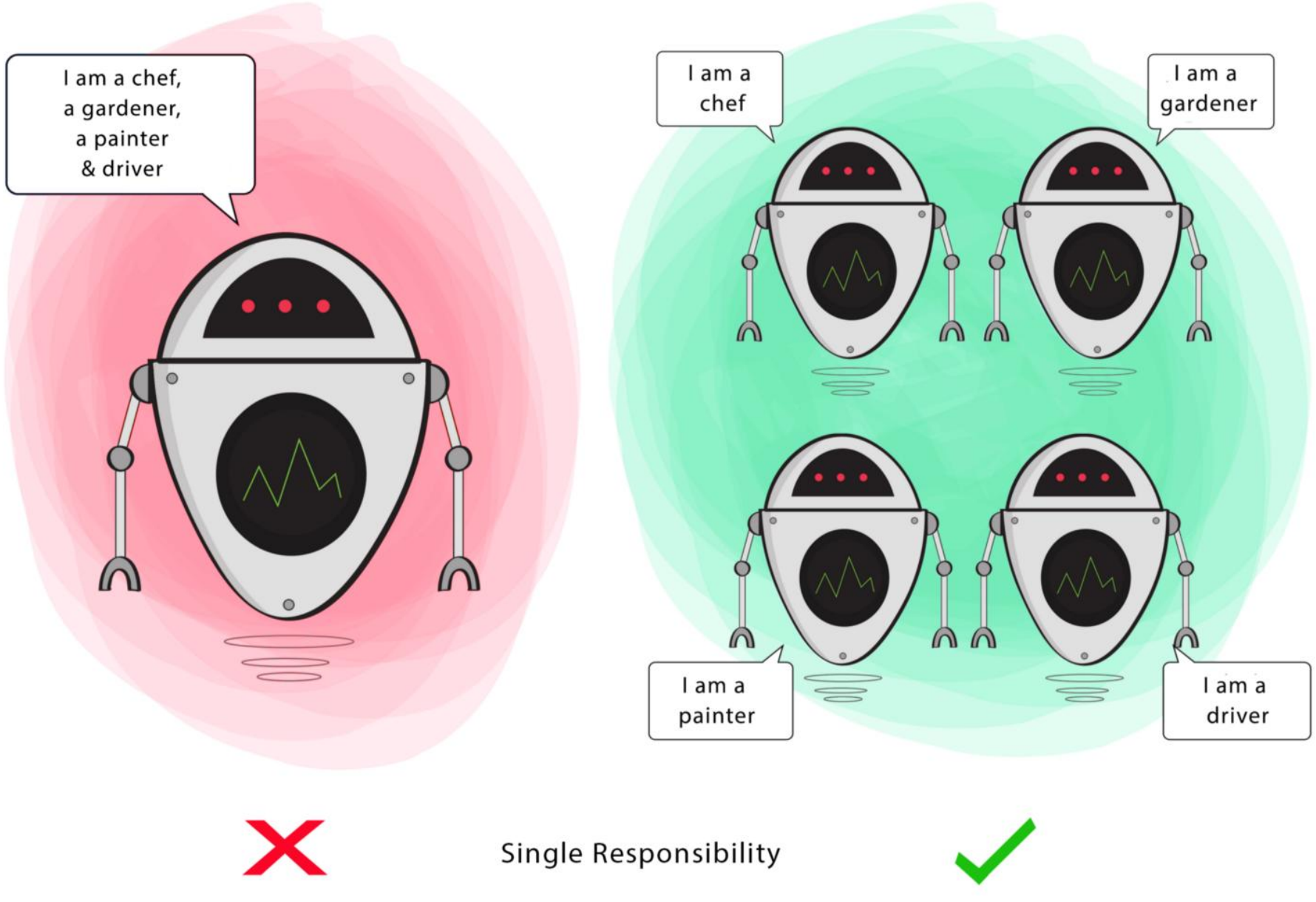
For each class there should be only one requirement which, when changed, incurs a change to that class

Discussion #1:

Do we now fully obey the Single Responsibility Principle?

SRP in Summary

- Idea
  - Avoid God classes and Swiss army knife classes
- Why?
  - Small classes are easy to understand, modify, and debug
  - Small classes are hard to get wrong ☺
  - Supports team collaboration
- Consequences
  - 4-5 times more classes – but small, simple classes!
  - Functionality will appear as classes



Single Responsibility

Illustration by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- **Open/Closed Principle (OCP)**
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Open/Closed Principle (OCP)

Software entities should be open for extension, but closed for modification

What Does That Mean Exactly?

When a class is done, it is done!

You add new functionality.
You derive from existing functionality.
You plug in new functionality into existing.

There should be no cascading modifications throughout classes!

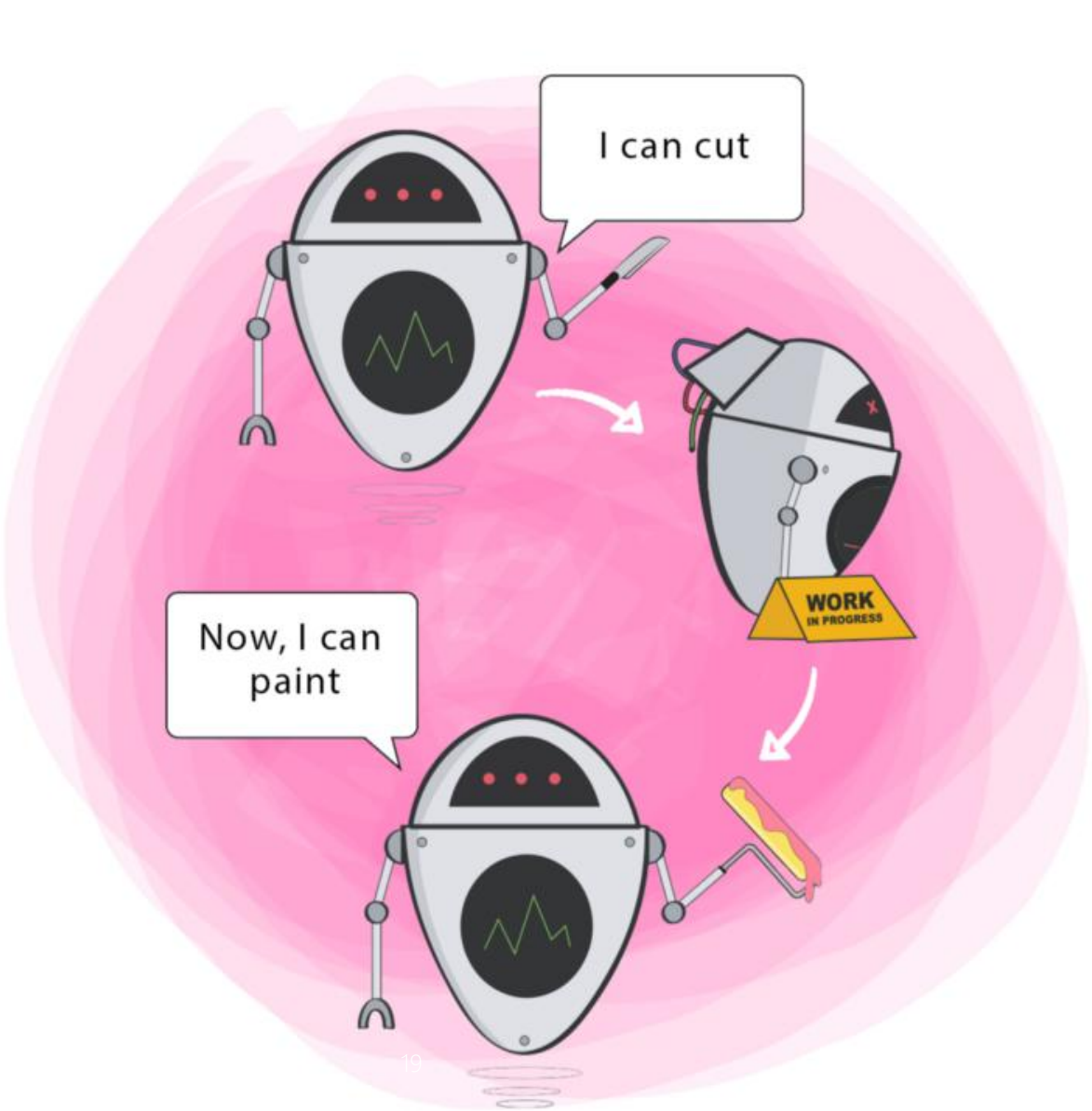
Abstract Base Classes or Interfaces?

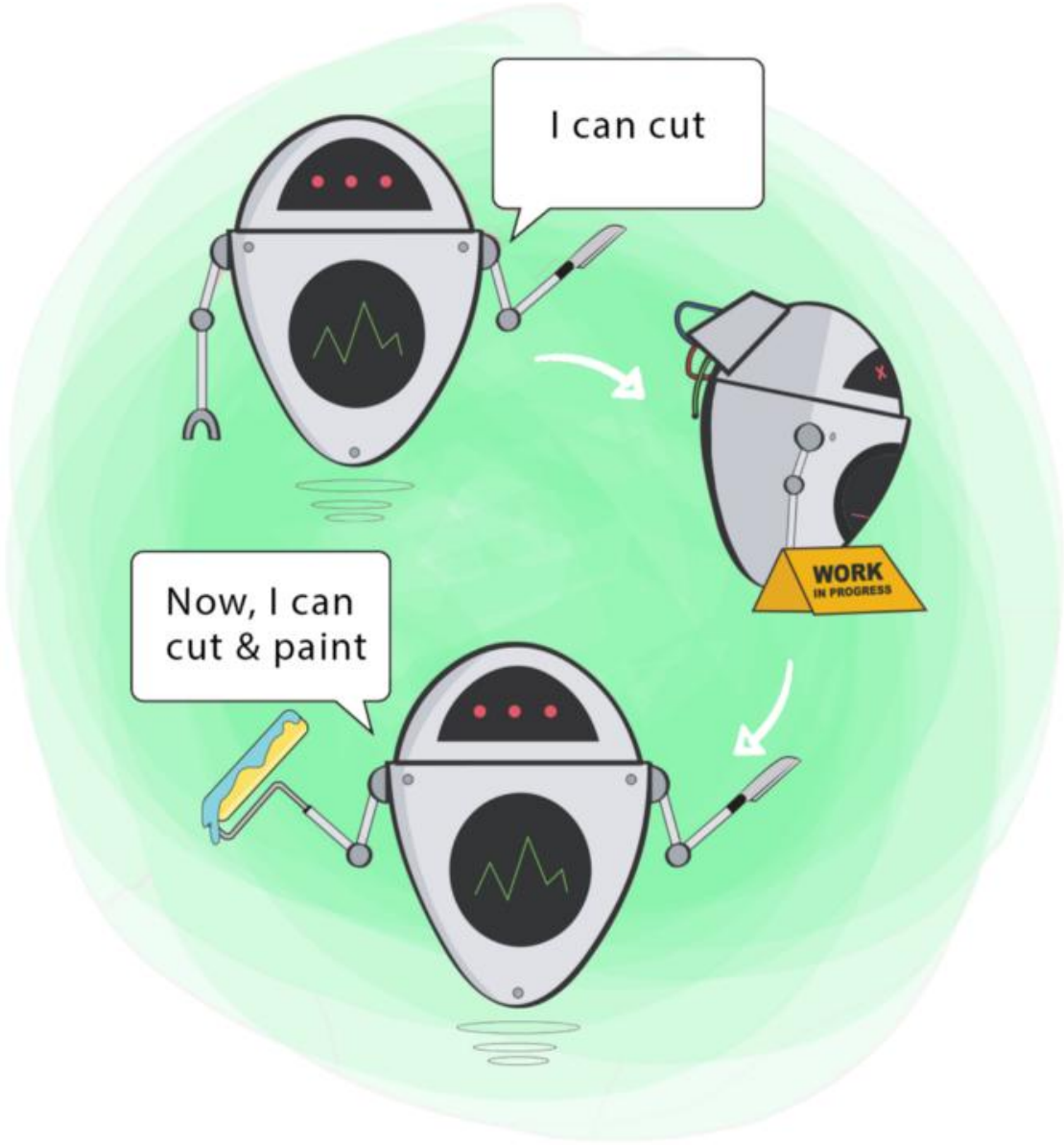
- Bertrand Meyer's original definition
  - Based on (abstract) classes and inheritance
  - Can lead to quirky and multiple levels of inheritance
  - Puts large responsibility on author of base class

- The modern interpretation (a.k.a. "Polymorphic")
  - Interfaces
  - Allows swapping out complete implementations to avoid quirkiness

OCP in Summary

- Idea
  - Add, derive or plug-in new functionality without changing existing classes
- Why?
  - Everything that worked before still works!
  - No accidental errors to existing code
  - Easier to locate newly introduced errors
  - Supports team collaboration
- Consequences
  - Changes are easy to locate and review
  - Existing tests still work when new requirements are added





Open-Closed

Illustration by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- **Liskov's Substitution Principle (LSP)**
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Liskov Substitution Principle (LSP)

If S is a subtype of T, then objects of type T may be replaced with objects of type S without breaking the program

What Does That Mean Exactly?

Any derived class should be substitutable for its base class.

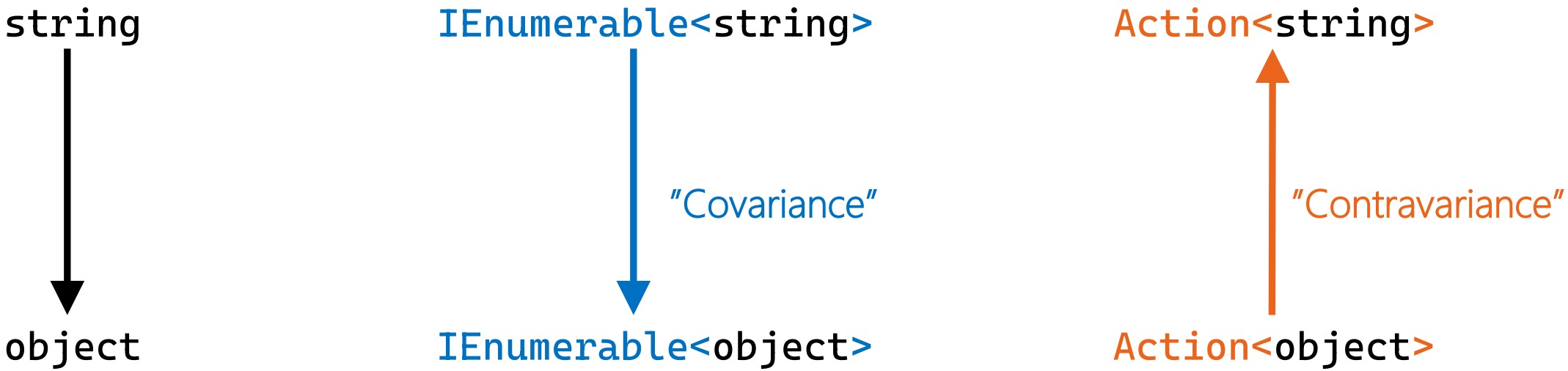
When you add new functionality, don't make any changes which cause existing code to break.

Essentially: "Behave well!"

LSP Variance Rules ~ Signature

- Contravariance of the method arguments in a subtype
- Covariance of the method return type in a subtype
- No new exceptions can be thrown by the subtype (unless they are part of the existing exception hierarchy)

Covariance and Contravariance



Discussion #2:

Are we – at this point – obeying the Liskov Substitution Principle?

Why?

Discussion #3:

Is this correct exception handling?

LSP Contract Rules ~ Behavior

- Preconditions cannot be strengthened in a subtype
- Postconditions cannot be weakened
- Invariants of the base type must be preserved in a subtype
- History constraint: Mutability vs. Immutability

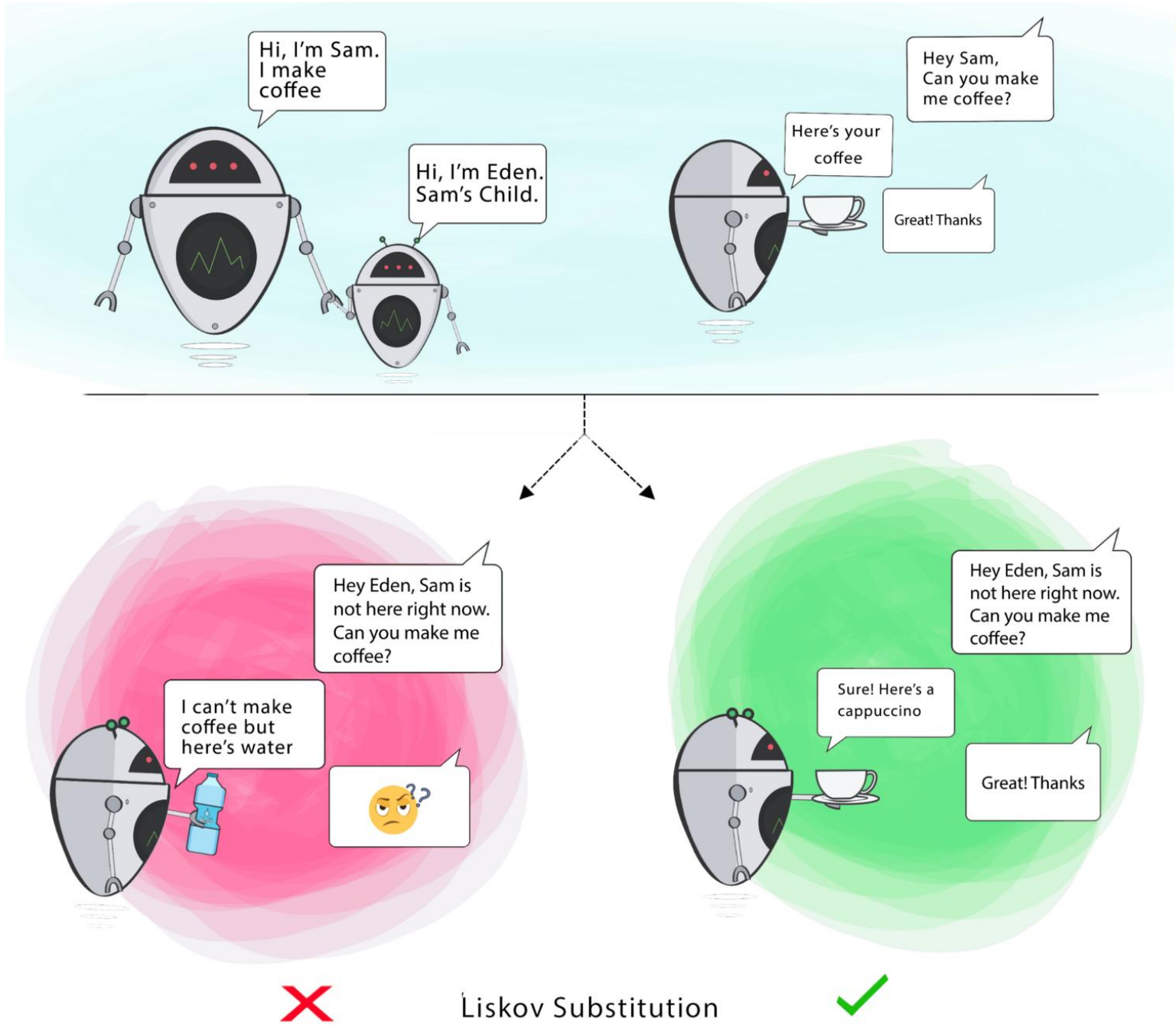
But…

- What about (pure) abstract base classes?
- What about interfaces?

- They have no existing implementation!

LSP in Summary

- Idea
  - Make sure your subtypes behave well within the existing program
- Why?
  - Due to OCP you will swap functionality all the time.
  - Swapping in new functionality should not break your program
  - Essentially, LSP is a vital enabler for the other SOLID rules
- Consequences
  - Must implement all methods and properties in subclasses in the ”spirit” of the existing program
  - Understand (and respect) the data invariants of the base classes
  - Nothing breaks…! ☺



Liskov Substitution

Illustration by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- **Interface Segregation Principle (ISP)**
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Interface Segregation Principle (ISP)

A client should not be forced to depend upon methods it doesn't use

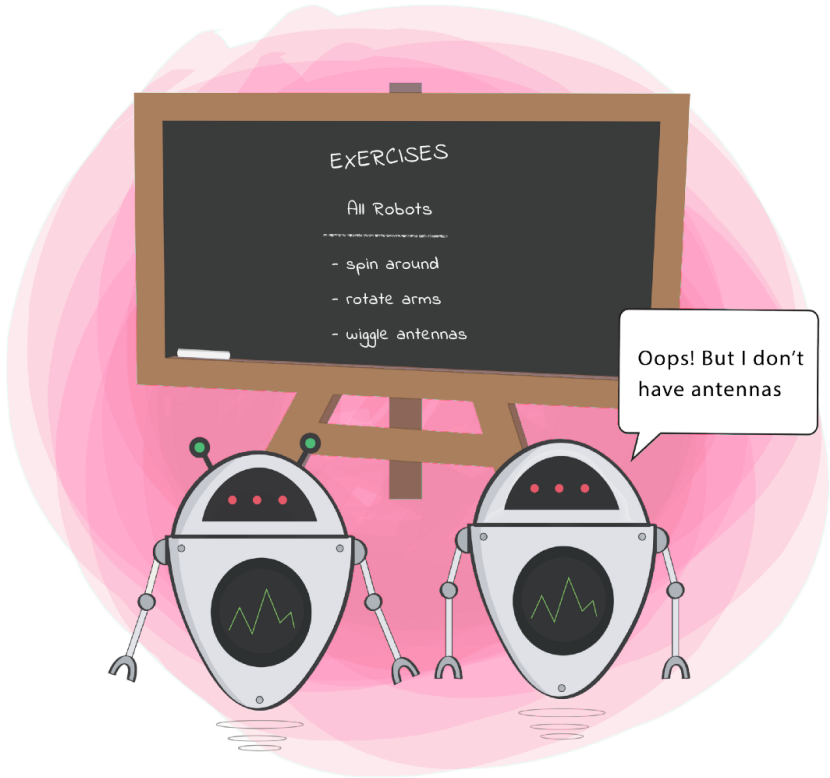
What Does That Mean Exactly?

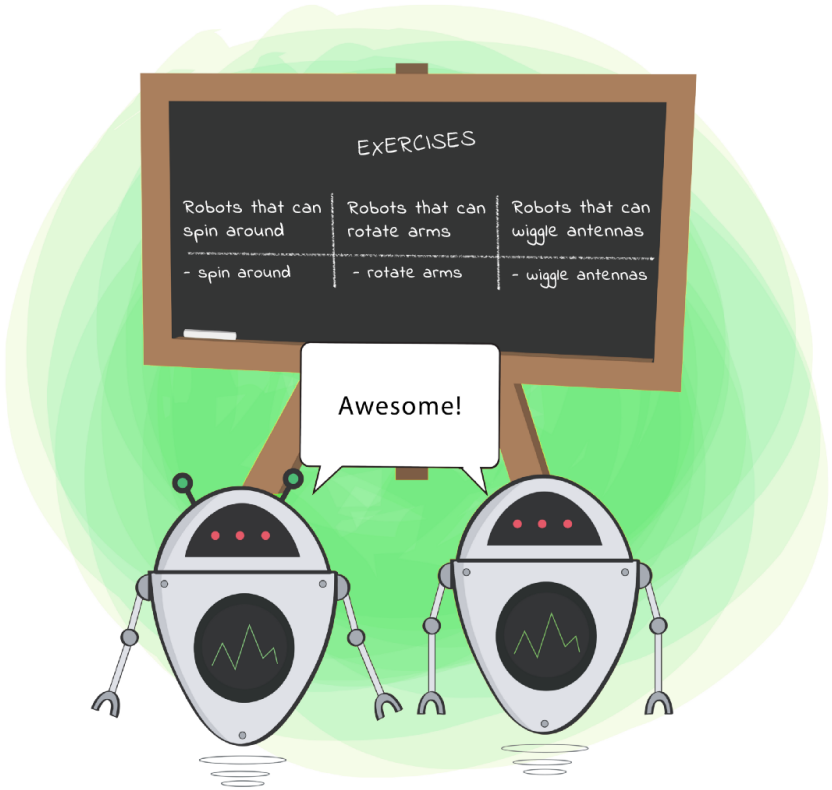
Break interfaces into smaller, more focused interfaces.

(Can still combine smaller interfaces using interface inheritance, though)

ISP in Summary

- Idea
  - Make your interfaces small and focused
    - This includes method parameters as well!
- Why?
  - Bloated interfaces probably violate SRP (or LSP)
  - Prevents references to unused dependencies
- Consequences
  - Interfaces become easier to implement
  - Classes and components have fewer dependencies





Interface Segregation

Illustration by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- **Dependency Inversion Principle (DIP)**
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Dependency Inversion Principle (DIP)

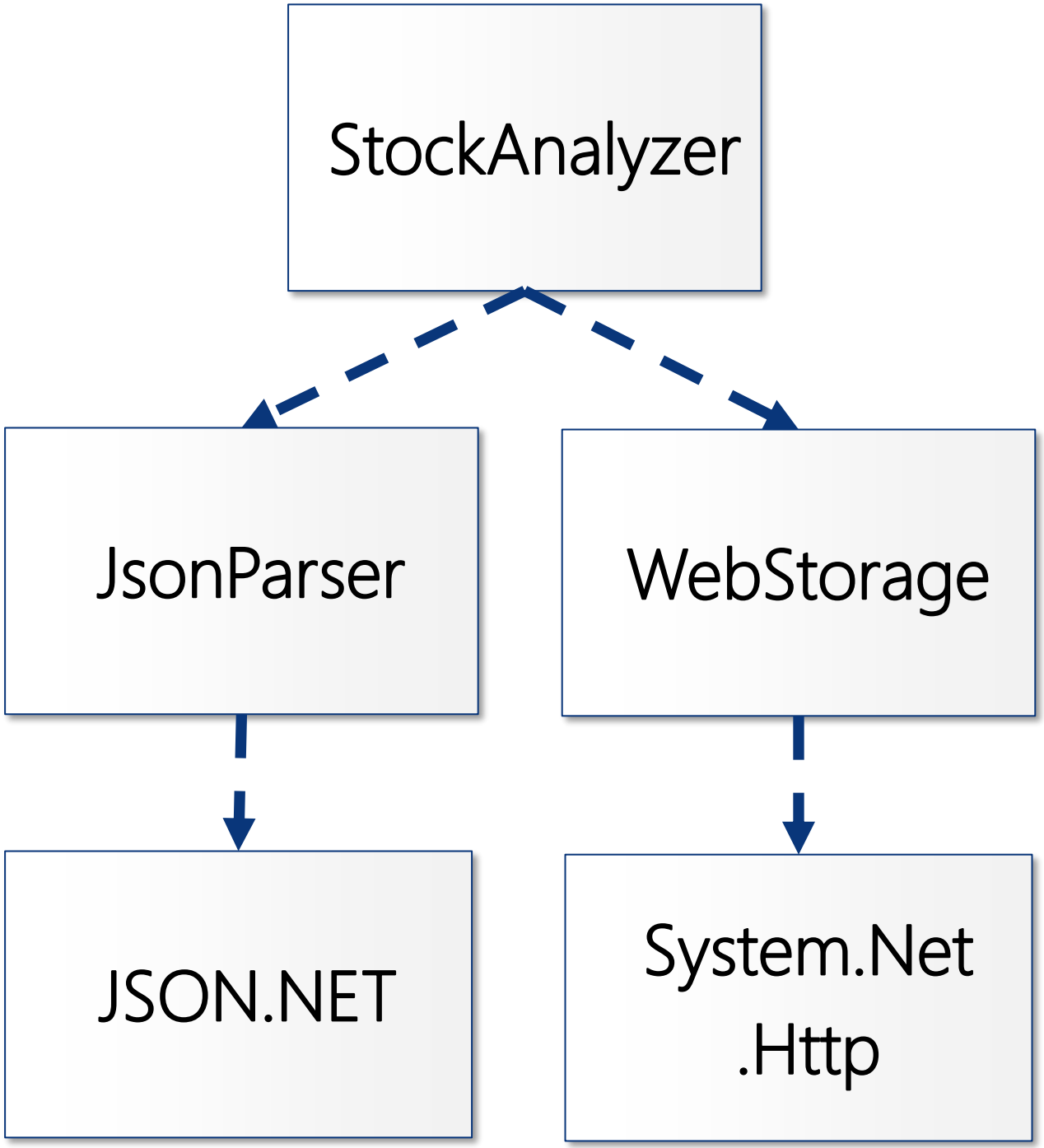
High-level modules should not depend on low-level modules. Both should depend on abstractions.

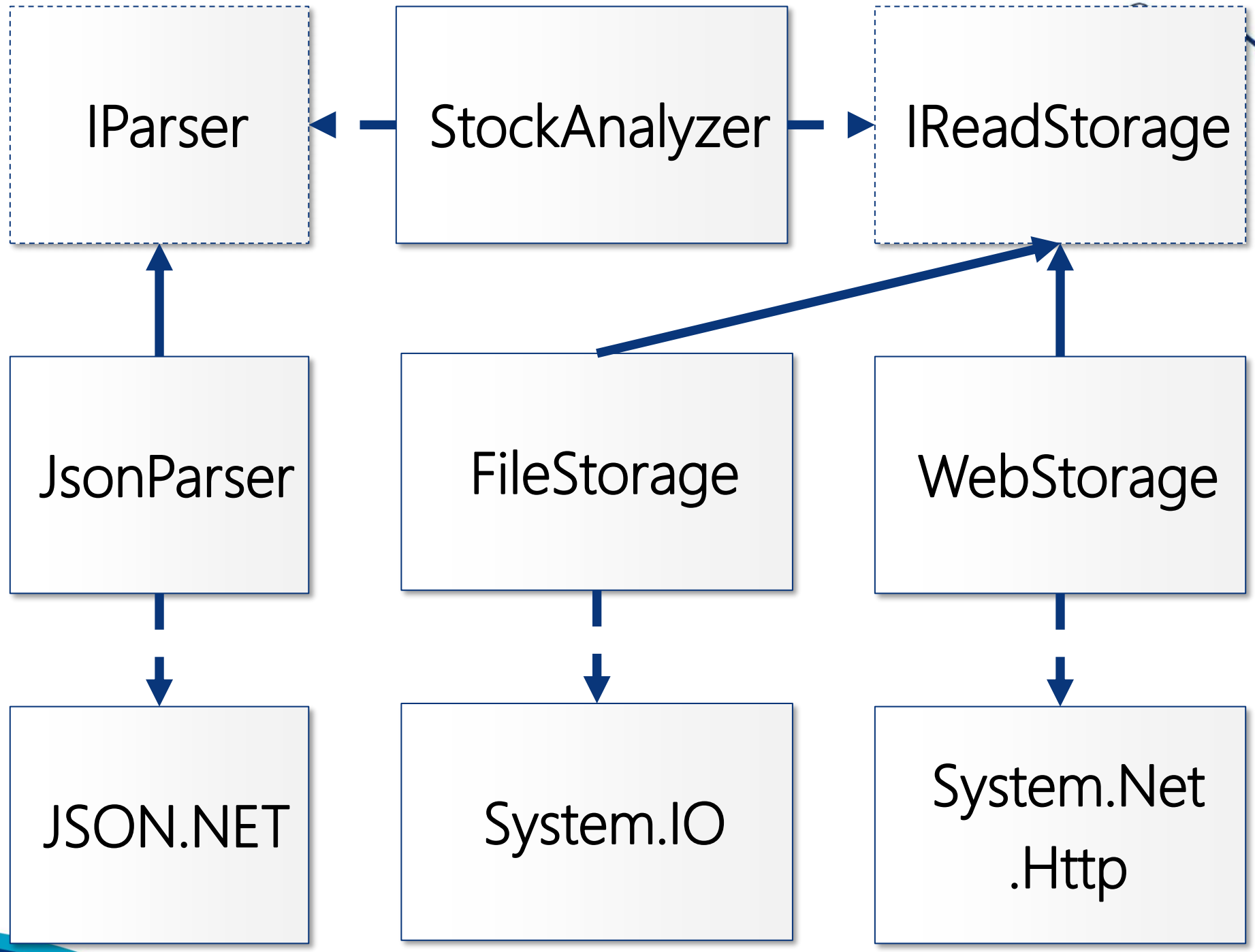
Abstractions should not depend upon details. Details should depend upon abstractions.

What Does That Mean Exactly?

Ensure that your classes do not depend upon specific implementations. Then you have the freedom to freely swap implementations and behavior.

A class' dependencies are supplied to the class – not created by the class itself!


StockAnalyzer
JsonParser
WebStorage
JSON.NET
System.Net
.Http


IParser
StockAnalyzer
IReadStorage
JsonParser
FileStorage
WebStorage
JSON.NET
System.IO
System.Net
.Http

Volatile Dependencies

- Out-of-process or unmanaged resources
- Nondeterministic resources
- Resources to be
  - Replaced
  - Intercepted
  - Decorated
  - Mocked

Examples of Volatile Dependencies

- Databases
- File system
- Web services
- Security contexts
- Message Queues
- **`System.Random`** (or similar)

Stable Dependencies

- A dependency is *stable* if it's not volatile...!

```
interface IUserRoleParser
{
    bool Parse(string role);
}
```

```
class DisplayState
{
    public string Name { get; }
    public string DisplayText { get; }

    public DisplayState(MovieDto movie) { ... }
}
```

Discussion #4:

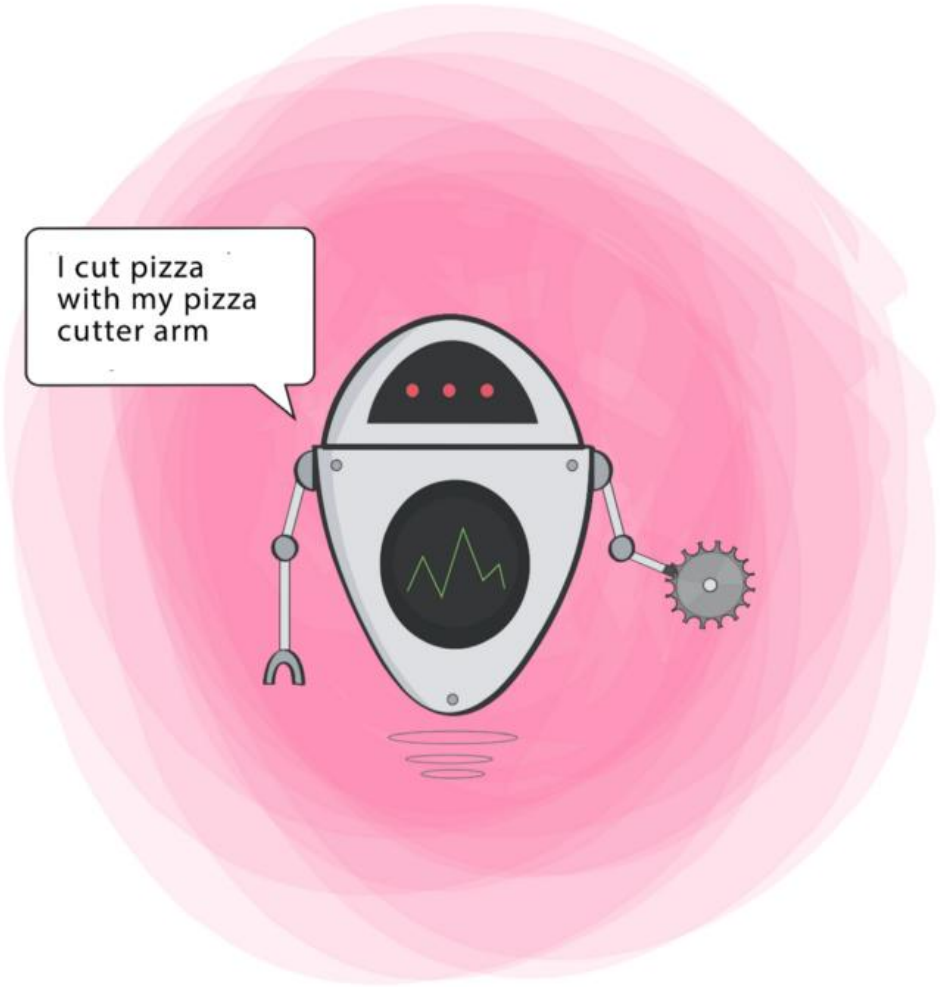
Which type of dependency is **`Computation`**?

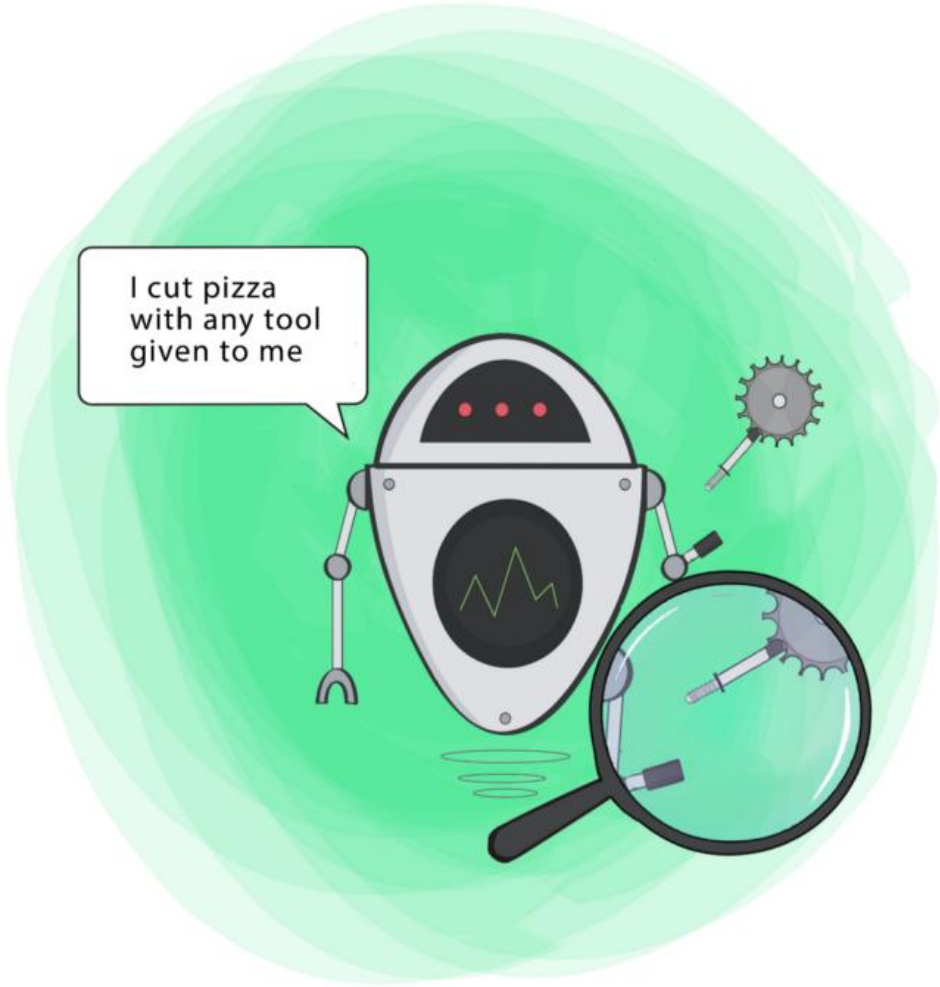
Dependency Injection applies exclusively to Volatile Dependencies.

Don't inject Stable Dependencies!

DIP in Summary

- Idea
  - Don't depend on concrete implementations! Only depend upon abstractions
  - Feed the dependencies needed into a class' constructor
- Why?
  - Maximize freedom to change implementations, because a class will never depend upon specific implementations – only their abstraction
  - Testability
  - Minimize dependencies and dependencies' dependencies
- Consequences
  - Classes will become loosely coupled
  - Your program becomes eligible for Dependency Injection





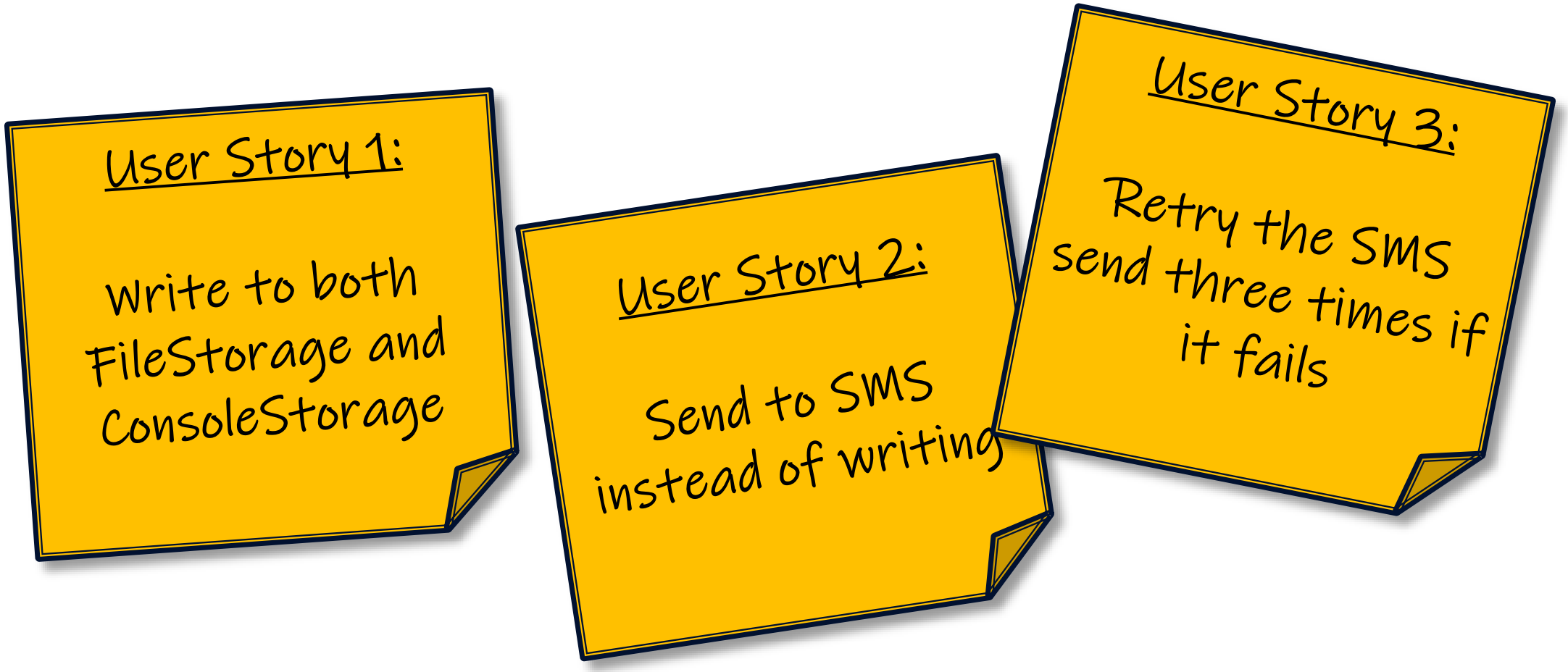
Dependency Inversion

Illustration by Ugonna Thelma from "THE S.O.L.I.D. PRINCIPLES IN PICTURES"

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- **Discussions: Living SOLID-ly**
- Dependency Injection Containers

Discussion: Refinement



User Story 1 Solution: Composite Pattern

```
class CompositeWriteStorage : IWriteStorage
{
    private readonly IEnumerable<IWriteStorage> _storages;

    public CompositeWriteStorage(IEnumerable<IWriteStorage> storages)
        ⇒ _storages = storages.ToList();

    public CompositeWriteStorage(params IWriteStorage[] storages) :
        this(storages.AsEnumerable()) {}

    public async Task StoreDataAsStringAsync(string outputDataAsString) ⇒
        await Task.WhenAll(_storages
            .Select(storage ⇒
                storage.StoreDataAsStringAsync(outputDataAsString)
            )
        );
}
```

User Story 2 Solution: Strategy Pattern

```
class TwilioSmsTransmissionStorage : IWriteStorage
{
    private readonly string _recipientPhone;
    public TwilioSmsTransmissionStorage(string recipientPhone) ⇒
        _recipientPhone = recipientPhone;

    public async Task StoreDataAsStringAsync(string outputDataAsString)
    {
        ...

        _ = await MessageResource.CreateAsync(
            to: new PhoneNumber(_recipientPhone),
            from: new PhoneNumber(TwilioConstants.FromPhone),
            body: outputDataAsString
        );
    }
}
```

User Story 3 Solution: Proxy Pattern

```
class RetryingWriteStorage : IWriteStorage
{
    private readonly IWriteStorage _proxee;

    public RetryingWriteStorage(IWriteStorage proxee) ⇒
        _proxee = proxee;

    public Task StoreDataAsStringAsync(string outputDataAsString)
    {
        IAsyncPolicy policy = Policy
            .Handle<Exception>()
            .WaitAndRetryAsync(3, _ ⇒ TimeSpan.FromSeconds(2));

         return policy.ExecuteAsync(() ⇒
            _proxee.StoreDataAsStringAsync(outputDataAsString));
    }
}
```

Discussion: Summarizing


Life is good

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers

Pattern: Pure DI (a.k.a. Poor Man's DI)

- *Pure DI is the practice of applying DI without a DI Container.*

- Outline
  - This is essentially what we have been doing throughout the workshop
  - Compose object graphs manually at Composition Root

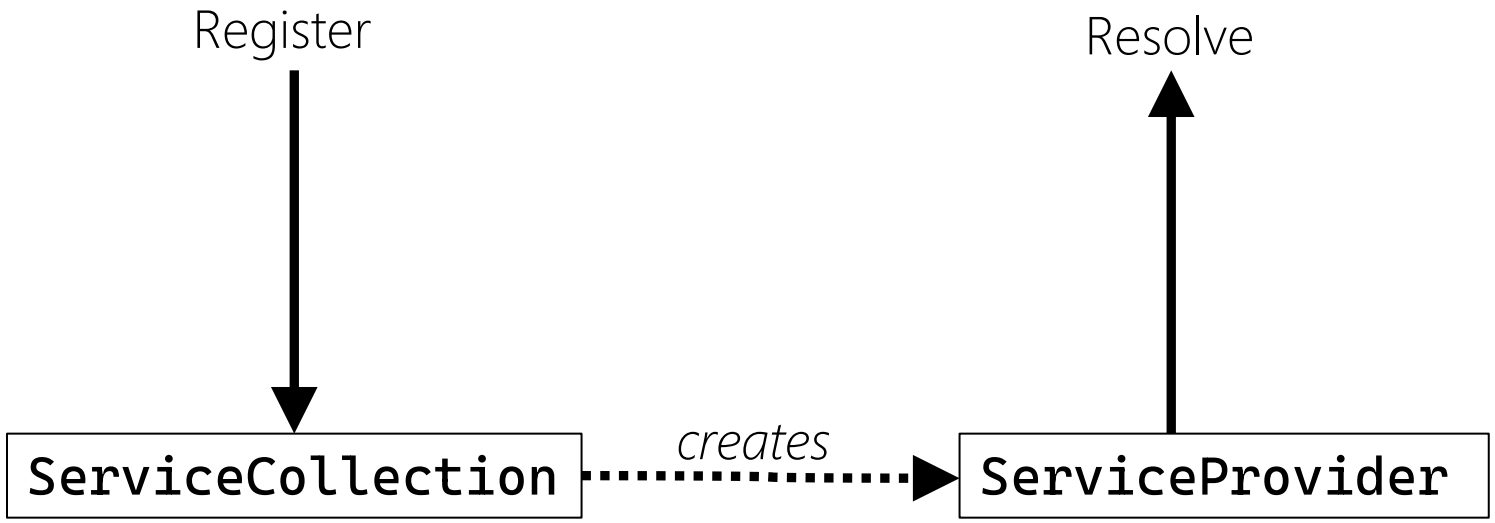
- See:
  "Dependency Injection Principles, Practices, and Patterns"
  Steven van Deursen and Mark Seemann (2019)

Dependency Injection Containers

Autofac
Simple Injector
Microsoft.Extensions.DependencyInjection
Unity
Ninject
Castle Windsor
Spring.NET
SimpleIoc
…

Don’t create your own! ☺

Microsoft.Extensions.DependencyInjection



Register and Resolve

```
IServiceCollection services = new ServiceCollection();
services
    .AddTransient<IParser, JsonParser>()
    .AddTransient<IFormatter, JsonFormatter>()
    .AddSingleton<IWriteStorage, ConsoleStorage>()
    .AddSingleton<IReadStorage>(sp ⇒
        new WebStorage(Url)
    )
    .AddSingleton<StockAnalyzer>()
    ;
```

```
IServiceProvider serviceProvider = services.BuildServiceProvider(true);
StockAnalyzer analyzer = serviceProvider.GetRequiredService<StockAnalyzer>();
```

Definition: Lifestyles

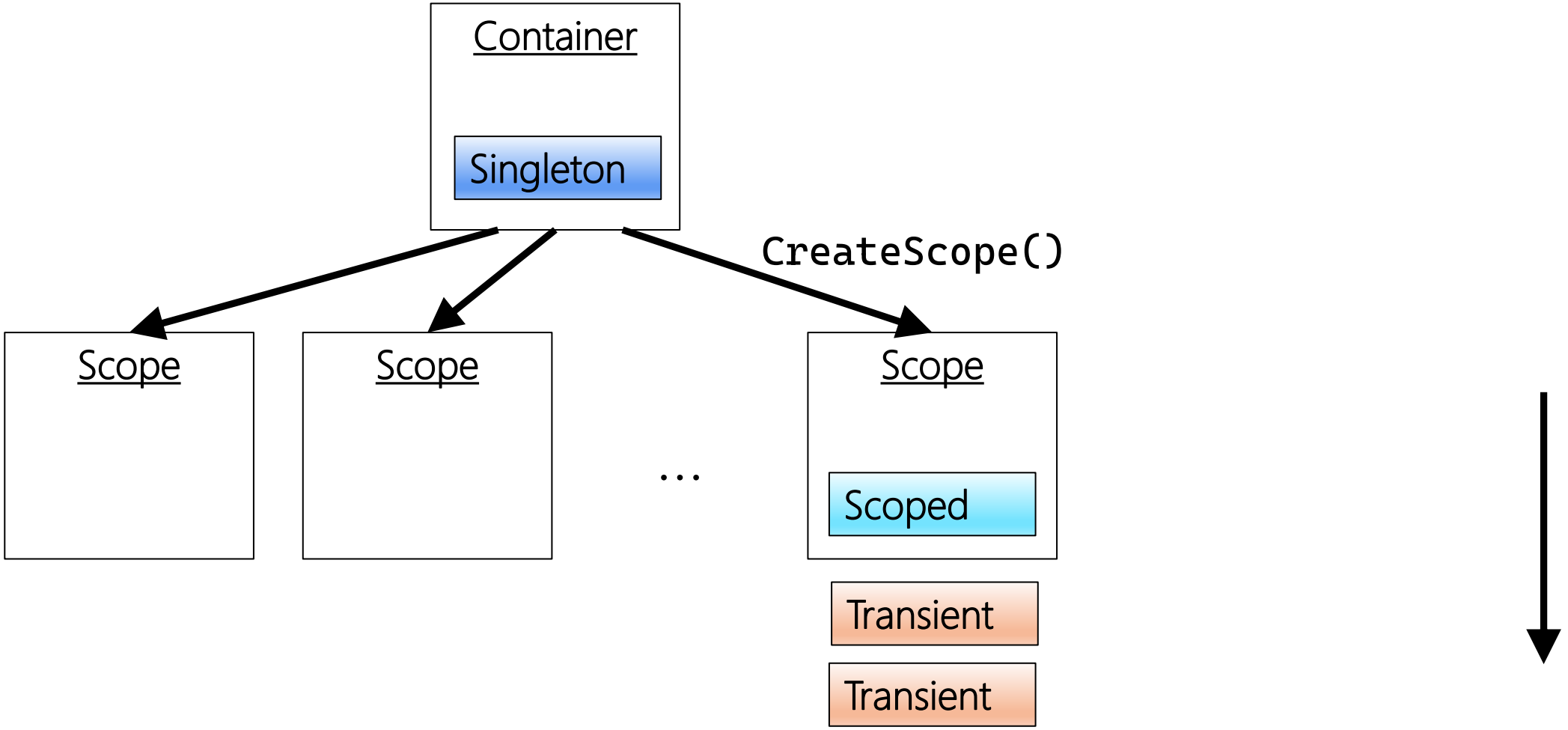
- *A Lifestyle is a formalized way of describing the intended lifetime of a dependency.*

- Transient ~ New instance at every resolve
- Singleton ~ Only one instance exists per container (*)
- Scoped ~ New instance at every scope

- Note: Lifetime is associated with registrations

Containers and Scopes

- Dependencies should be resolved from Scopes – not the Container itself



Creating Scopes

```
IServiceProvider serviceProvider = services.BuildServiceProvider(true);

using IServiceScope scope = serviceProvider.CreateScope();
StockAnalyzer analyzer = scope.serviceProvider.GetRequiredService<StockAnalyzer>();
```

Pattern: Register-Resolve-Release

- *Always do a sequence of three things with a container:*
  - *Register components with the container*
  - *Resolve root components*
  - *Release components from the container.*

- Outline
  - RRR captures the best practice of container use in the Composition Root only!

- See:
  https://blog.ploeh.dk/2010/09/29/TheRegisterResolveReleasepattern/
  Mark Seemann (2010)

Summary

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Discussions: Living SOLID-ly
- Dependency Injection Containers


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Ringgårdsvej 4A
8270 Højbjerg
Denmark