Module 14:
"The SOLID Principles"

TEKNOLOGISK
INSTITUT

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly
- Summary

SOLID is…

- … Five fundamental "commandments" for OOP
- … Programming language-agnostic
- … Not a framework or package!

- … Maintainability!

"Always write your code presuming it will be maintained by an ill-tempered axe murderer who knows where you live..."

The Five Principles of SOLID

- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

TEKNOLOGISK INSTITUT

Agenda

- Introducing SOLID
- **Single Reponsibility Principle (SRP)**
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly
- Summary

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

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

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- **Open/Closed Principle (OCP)**
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly
- Summary

TEKNOLOGISK
INSTITUT

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

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- **Liskov's Substitution Principle (LSP)**
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly
- Summary

TEKNOLOGISK
INSTITUT

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 stuff 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)

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 SRP and 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…! ☺

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- **Interface Segregation Principle (ISP)**
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly
- Summary

TEKNOLOGISK
INSTITUT

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

Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- **Dependency Inversion Principle (DIP)**
- Living Your Life SOLIDly
- Summary

Dependency Inversion Principle (DIP)

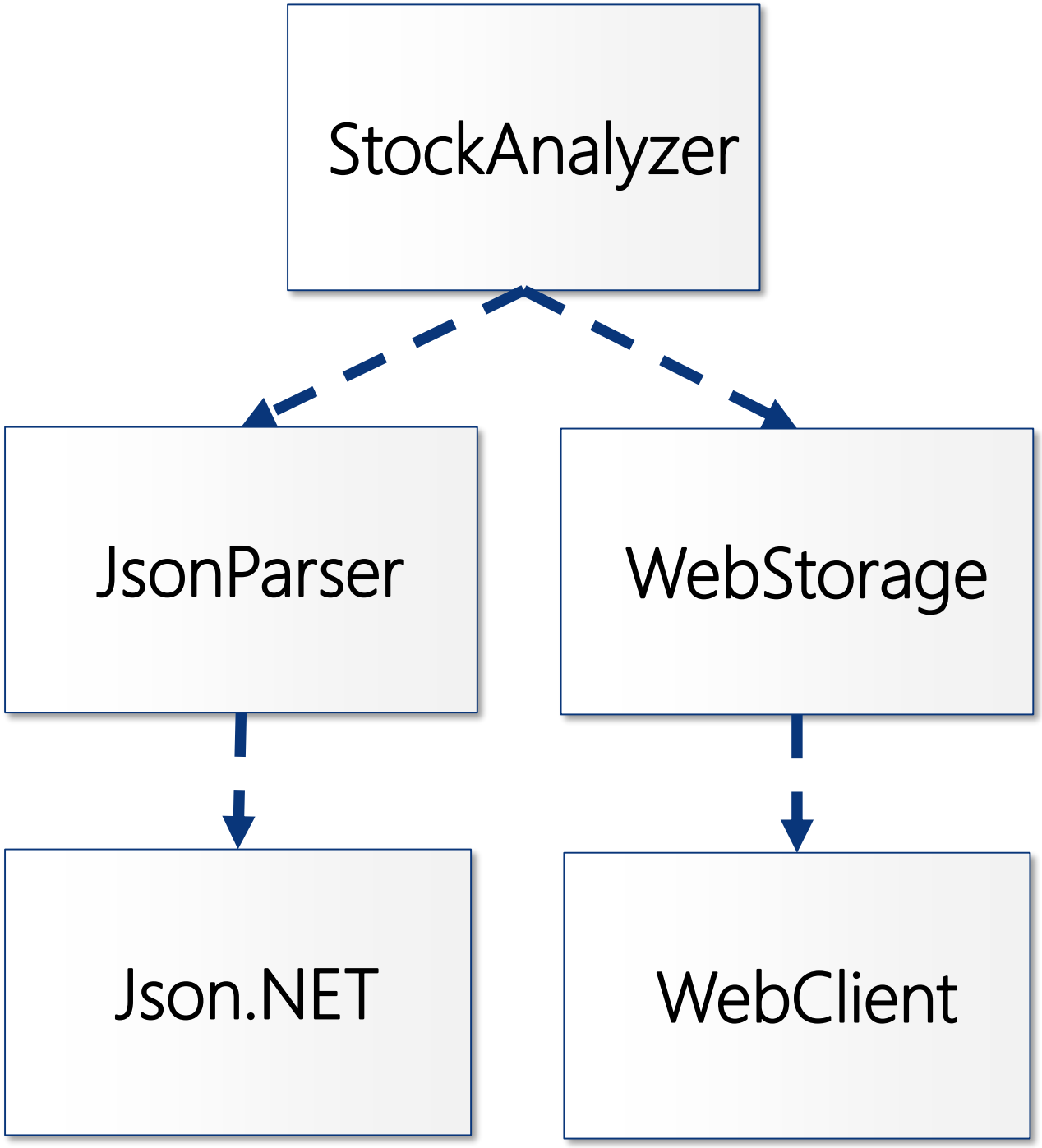
High-level modules should not depend on low-level modules. Both should depend on abstractions.

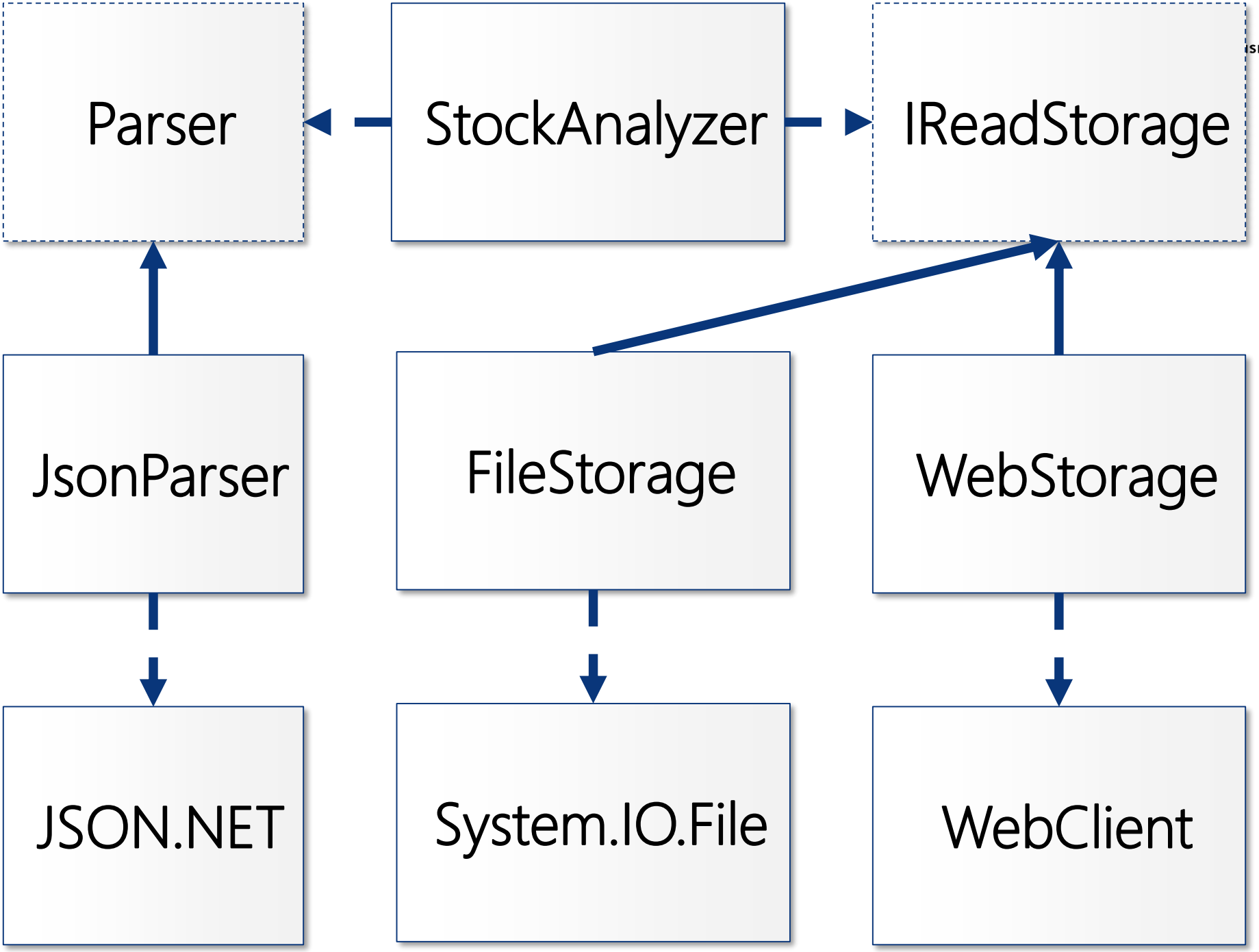
Abstractions should not depend upon details. Details should depend upon abstractions.

What Does That Mean Exactly?

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

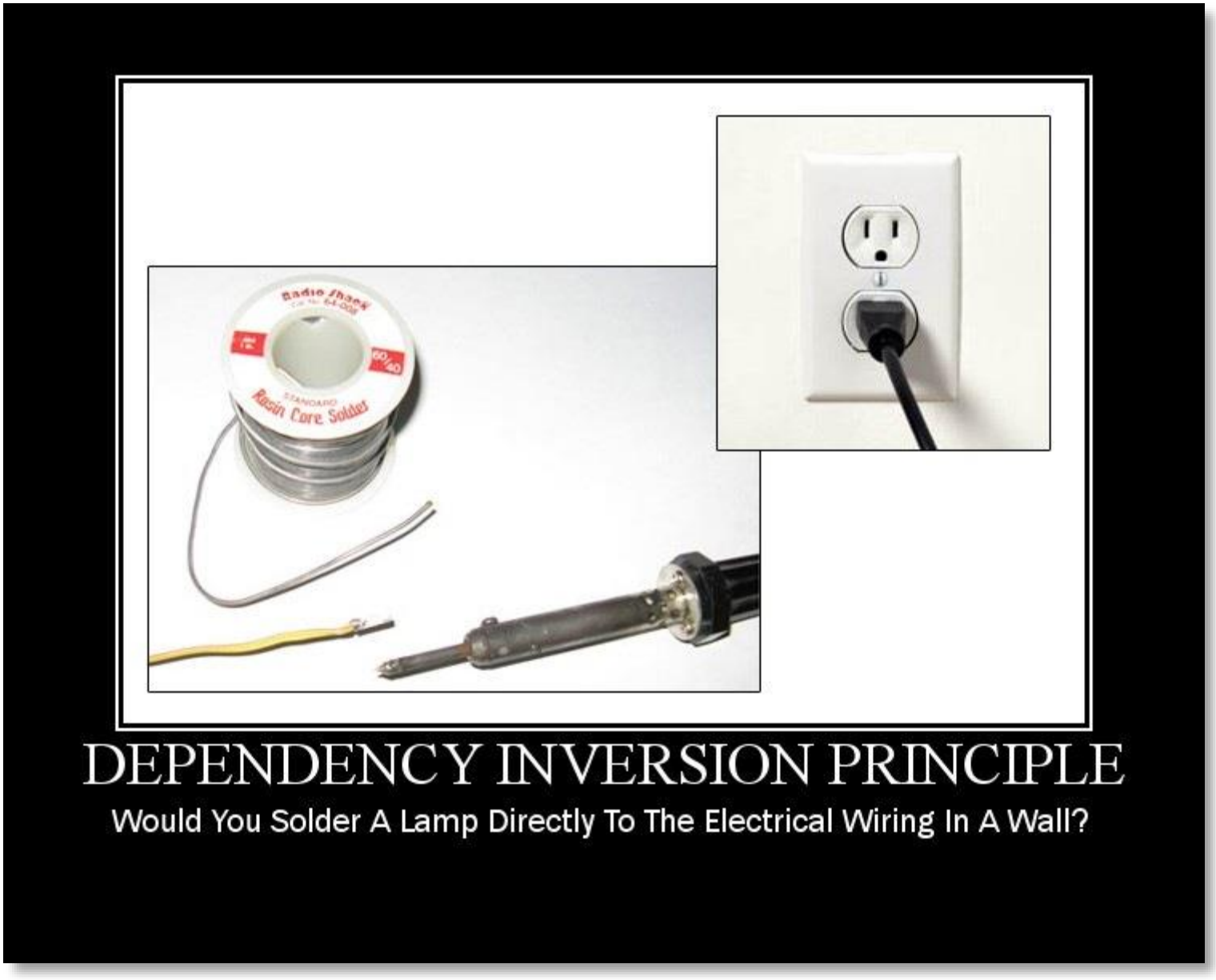
A class' dependencies are supplied to the class – not created by the class itself!


StockAnalyzer
JsonParser
WebStorage
Json.NET
WebClient


Parser
StockAnalyzer
IReadStorage
JsonParser
FileStorage
WebStorage
JSON.NET
System.IO.File
WebClient

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



Agenda

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- **Living Your Life SOLIDly**
- Summary

Discussion:

How would you enable the program to send an SMS instead?

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
        );
    }
}
```

Discussion:

How would you add multiple write storage objects?

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 Task StoreDataAsStringAsync(string outputDataAsString) =>
        Task.WhenAll(_storages
            .Select(storage =>
                storage.StoreDataAsStringAsync(outputDataAsString)
            )
        );
}
```

TEKNOLOGISK
INSTITUT

Discussion:

How would you add retry strategies?

Solution: Proxy/Decorator 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));
    }
}
```


THIS IS
THE WAY

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 MovieViewModel
{
    public string Name { get; }
    public string DisplayText { get; }

    public MovieViewModel(MovieShowing movie) { ... }
}
```

TEKNOLOGISK
INSTITUT

To Inject or Not To Inject?

Dependency Injection applies exclusively to Volatile Dependencies.

Don't inject Stable Dependencies!

Never Make Dependencies Optional!

- You should usually always require a dependency
  - Only exception is if it has a Local Default
  - If it is not present (yet?) then use a Null Object
- Local Default
  - A default implementation of a dependency which resides in the same module or layer of the application
  - (As opposed to Foreign Default)
- Note:
  - A Local Default cannot have any Foreign Defaults

SOLID – Concluding Remarks

- Principles as a progression of improvements to code
  - An important mindset for development
    - Improved maintainability
    - Better cooperation between developers
- Many, many more classes (but smaller!)
- SOLID vs. YAGNI?
  - Very hard in the beginning
- Might be reasons for not using SOLID (or only a subset)

TEKNOLOGISK INSTITUT

SOLID – Concluding Remarks

- Don't sacrifice everything for obtaining Nirvana!
- Is there a bucket of gold at the end of the rainbow…?

- No! But you will be very well rewarded for your walk! ☺

Summary

- Introducing SOLID
- Single Reponsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov's Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Living Your Life SOLIDly


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