.NET for C Programmers

Day 2

Overview

- Clean Code
- OOP
- SOLID
- DI in C#

.NET for C Programmers

Clean Code

Clean code is important

- Testable and impossible to hide bugs
- Single responsibility
- Extensible
- Easy to change
- Easy to understand

Things we will see often

Good design	Bad design
Loosely coupled	Rigid
Highly cohesive	Fragile
Easily composable	Immobile
Context independent	Viscous

- It’s all about dependencies
  - In .NET, a reference == dependency
  - A change in dependency == necessary change in code
  - Change in code == headache

So what do we do?

- Use OOP as a programming model
- Follow SOLID principles
- Use interfaces, interface based polymorphism and DI/IOC for reuse

.NET for C Programmers

Principals of OOP

OOP

- An approach to building applications that are:
  - Flexible
  - Natural
  - Well-crafted
  - Testable
- With a focus on business objects that interact cleanly with each other

Identifying Classes

Separating Responsibilities

Establishing Relationships

Leveraging Reuse

Concepts in OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- Treat objects with common attributes and behavior with a single representation (a class)
- Classes represent implementation of state and behavior
- For clients of classes, we tend to want to use interfaces instead as they allow us to get further away from dependencies

Abstraction

- Breaking the domain into categories of common data with common functionality
- Types with a common abstraction can be accessed consistently from outside code

Abstraction

- Simplifying reality
- Ignoring extraneous details
- Focusing on what is important for a purpose

Encapsulation

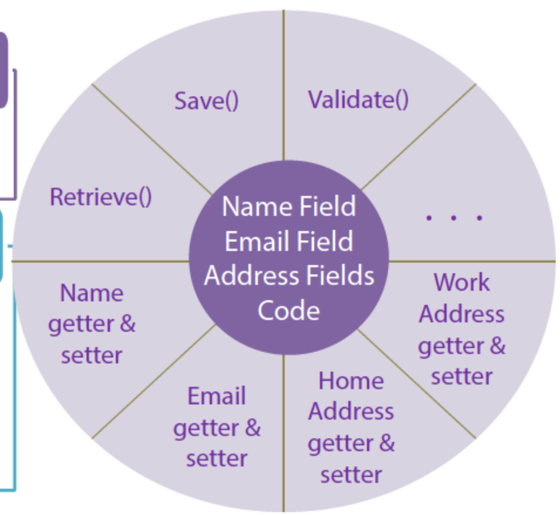
- Represents the state of the data
- A goal in OOP is to hide the state of the object from the outside
- Outsiders only access the object through methods and properties

Data hiding

- Protects the data

Implementation hiding

- Helps manage complexity
- Implementation can be changed without impacting the application

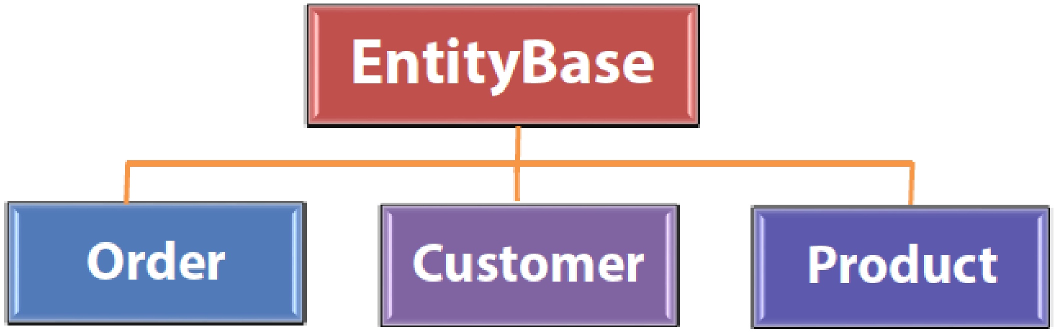


Encapsulation

- Hiding of the state of the object
- C# fields represent internal state, only accessible within the class, and usually not by derived classes. Usually, they are private.
- Properties represent state that is presented outside the class, or to derived classes, and normally treated as part of a contract that changing their value does not change behavior. They are almost always public.

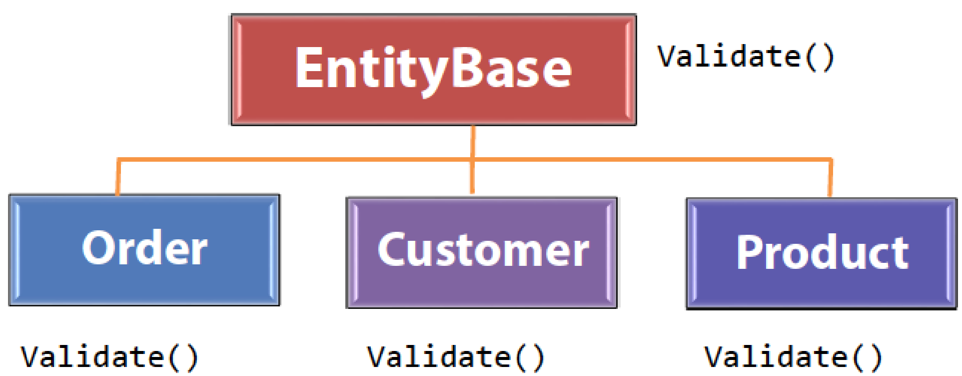
Inheritance

- A means of reusing code in OOP
- Derived classes gain functionality of their base classes
- And can change behavior based upon rules define in the base classes



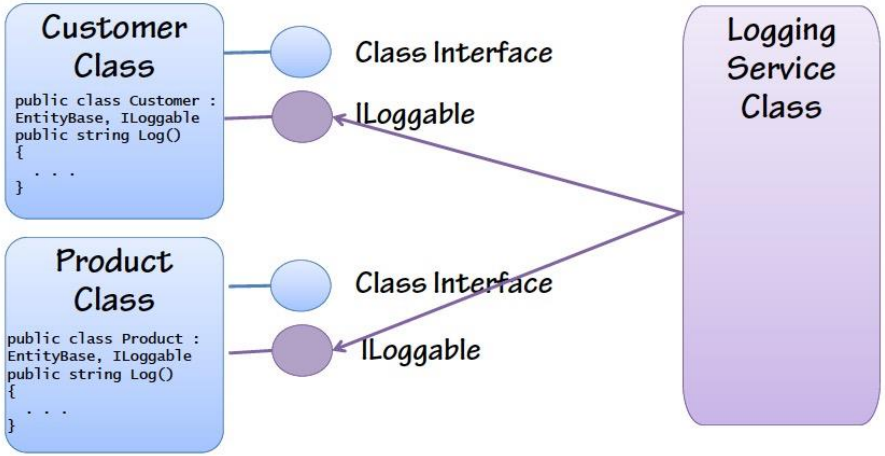
Inheritance based-polymorphism

- Derived classes can override certain methods (virtual / override)
- This allows them to selectively change specific pieces of behavior



Interface-based Polymorphism

- Facilitates the evolution of software
- Different implementations can be provided for a specific interface
- Allows change of implementation without breaking a client
- This will become much clearer after the SOLID module



.NET for C Programmers

Implementing OOP with C# / .NET

Review: Concepts in OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Encapsulation with C#

- Implement state as fields in the objects
- Fields should be private

Good Encapsulation

- State can not be depended upon by derived classes or by other classes
- _someState can not be changed outside of the class it is contained within
- Tests on GoodEncapsulation will not break due to subclasses or other classes

```
2 references | 0 authors | 0 changes
public class GoodEncapsulation
{
    // I can't be seen outside this clas
    // OR by derived classes
    private int _someState;

    0 references | 0 authors | 0 changes
    public GoodEncapsulation()
    {
    }
}

0 references | 0 authors | 0 changes
public class Derived : GoodEncapsulation
{
    // I have no access to _someState
    // Derived classes should not know of
    // state in the base classes
}
```

Bad Encapsulation

- This will be bad
- Any code in the base depending on the value of _someState can now do the wrong thing

```
2 references | 0 authors | 0 changes
public class BadEncapsulation
{
    // I can be modified by derived classes
    protected int _someState;

    // I can be modified by anything
    public int _someOtherState;

    0 references | 0 authors | 0 changes
    public BadEncapsulation(){}
}

1 reference | 0 authors | 0 changes
public class Derived : BadEncapsulation
{
    0 references | 0 authors | 0 changes
    public Derived(){}

    0 references | 0 authors | 0 changes
    public void IWillMessThingsUp()
    {
        _someState = 10;

        // code in the base can now be inconsisten
    }
}
```

What about properties?

- Don’t properties expose state?
  - …. Sometimes
- Properties should be considered
  - Part of the objects “interface”
  - Can be read only
  - As guards on fields to ensure validity
  - As a means of providing internal state change notifications (along with events)
- But definitely not a public exposing of internal state

Abstraction with C#

- Two or more classes with similar functionality can be abstracted by creating an inheritance hierarchy
- Common / duplicated code is moved into the abstracted base class
- That can be coded and tested independently
- Derived classes should not be able to break their correctness
    - After all, they are depending upon the correctness of their base class

Rules on abstraction in C#: methods

- Do not expose base class fields as protected or public
- A public method can be considered available to derived classes and other classes
  - They should either return data or cause internal state change
  - And be tested for correctness
- Non-virtual protected base class members are available to derived classes to provide those classes internal implementations a means of communicating privately to their implemenation

Good abstraction

```
2 references | 0 authors | 0 changes
public class GoodAbstraction
{
    // my state is not available outside myself
    private int _someState;

    // I am used only for my internal implemenation
    0 references | 0 authors | 0 changes
    private void someMethod() { }

    // I exist for derived classes to tell me to do
    // things only a derived class should do with me
    // And, I can not be overriden - I am law
    0 references | 0 authors | 0 changes
    protected void forDerivedClasses() { }

    // Anyone can use me and I'll be tested throughly
    0 references | 0 authors | 0 changes
    public void IamPartOfThePublicInterface() { }
}

0 references | 0 authors | 0 changes
public class Derived1 : GoodAbstraction
{
    // GoodAbstraction base will be consisten for me
}

0 references | 0 authors | 0 changes
public class Derived2 : GoodAbstraction
{
    // and any other derivations
}
```

Bad abstraction

- Base class only supports one subclass well
  - Shouldn’t be an abstraction then
- Derived classes can change the “correctness” of the abstraction
  - State is exposed to them, or
  - Protected methods I provide can be used to break me
    - Ie, the allow derived classes to unexpectedly change by state

Polymorphism

- C# implements polymorphism using inheritance and virtual functions
- There are several patterns for this
  - Base class implementation needs contextual information
  - Base class defines a default which is expected to change
  - Base class needs to inform derivation of changes of state

Base class implementation needs contextual information

- Template method pattern
- If state in the abstraction is needed to make the decision, it is passed to the method

```
1 reference | 0 authors | 0 changes
public abstract class Abstraction {
    // template method for derived classes to implement
    2 references | 0 authors | 0 changes
    protected abstract bool helpsMakeDecision(int someState);

    private int _someState;

    0 references | 0 authors | 0 changes
    private void someAlgorithm() {
        // what I do depends on derivations deciding
        if (helpsMakeDecision(_someState)){
            // do this
        } else {
            // or do this
        }
    }
}

0 references | 0 authors | 0 changes
public class Derivation : Abstraction {
    2 references | 0 authors | 0 changes
    protected override bool helpsMakeDecision(int someState) {
        // some decision that needs to be made
        return someState < 10;
    }
}
```

Base class defines a default which is expected to change

- Derived class can process the data differently
- Helps to handle unknown situations
- Data passed to the specialization should not be state

```
1 reference | 0 authors | 0 changes
public abstract class Abstraction {
    0 references | 0 authors | 0 changes
    private void algorithm(){
        doThis();
        doThat();
        process(1);
        doSomethingElse();
    }
    1 reference | 0 authors | 0 changes
    private void doThis() { }
    1 reference | 0 authors | 0 changes
    private void doThat() { }
    1 reference | 0 authors | 0 changes
    private void doSomethingElse() { }

    3 references | 0 authors | 0 changes
    public virtual void process(int someData){
        // default processing
    }
}

0 references | 0 authors | 0 changes
public class Derived : Abstraction {
    3 references | 0 authors | 0 changes
    public override void process(int someData) {
        // I do something else with this data than what
        // is the default of the abstraction
        base.process(someData);
    }
}
```

Base class needs to inform derivation of changes of state

- Efficient and sage means of notifying only derived classes of private state change
- Default is empty, and if not needed in specialization nothing changes
- IMHO: Better than events

```
1 reference | 0 authors | 0 changes
public class Abstraction {
    private int _someState;

    0 references | 0 authors | 0 changes
    private void algorithm() {
        // do something that changes private state
        _someState++;

        informedDerivationOfStateChange();
    }

    2 references | 0 authors | 0 changes
    protected virtual void informedDerivationOfStateChange() { }
}

0 references | 0 authors | 0 changes
public class Derived : Abstraction {
    2 references | 0 authors | 0 changes
    protected override void informedDerivationOfStateChange() {
        // do someting as an important thing happened in
        // my base class
    }
}
```

Interface Based Polymorphism

- The implementation of behavior is represented by implementations of an interface
- A component expects a service based on an interface
- How it behaves depends on what implementation it uses
- Very important: which implementation it uses is not its own decision
- That decision is inverted

IBP: Part I

- Interface is defined
- Multiple implementations

```
5 references | 0 authors | 0 changes
public interface ILogger {
    2 references | 0 authors | 0 changes
    void LogThisMessage();
}

1 reference | 0 authors | 0 changes
public class FileLogger : ILogger {
    2 references | 0 authors | 0 changes
    public void LogThisMessage() {
        // write to a file
    }
}

1 reference | 0 authors | 0 changes
public class DataBaseLogger : ILogger {
    2 references | 0 authors | 0 changes
    public void LogThisMessage() {
        // write to database
    }
}
```

IBP: Part II

- The component that needs logging defines a property or constructor
- This will be set by another piece of code
- The component doesn't care what kind of logger, nor should it; It just wants to log

```
2 references | 0 authors | 0 changes
public class AComponentThatNeedsLogging {
    1 reference | 0 authors | 0 changes
    public ILogger LoggerToUse { get; set; }
}
```

IBP: Part III

- A factory constructs the component based upon rules
- It decides which logger it uses
- And, then to the client of the component, it is polymorphic in behavior depending upon the configuration

```
0 references | 0 authors | 0 changes
public class ComponentFactory {
    private Dictionary<int, Func<ILogger>> _loggerFactories =
        new Dictionary<int, Func<ILogger>>() {
            {0, () => new FileLogger()},
            {1, () => new DataBaseLogger()},
        };

    0 references | 0 authors | 0 changes
    public AComponentThatNeedsLogging buildComponent() {
        var component = new AComponentThatNeedsLogging();

        // read type of logger from config and inject
        // proper logger
        var loggerToUse = 0;
        component.LoggerToUse = _loggerFactories[loggerToUse]();

        return component;
    }
}
```

Inheritance

- A means of reusing code and data in C# / OOP
- Generally a good idea within an application, when done correctly
- But tends to be very brittle
  - Best for extending data
  - Not so much for changing behavior
- I'm honestly not a big fan of it in business objects
  - Good for GUI frameworks and such (behavior changes)
  - Sometime unavoidable for business objects
- I'm a bigger fan of interface based polymorphism
- Often abused for sideffects, violating SRP
  - Ie: MyObjectThat dervices from a logging abstraction

The Reality of Inheritance?

- Business objects are typically flat, with little hierarchy
- Favor aggregation over inheritance for responsibilities (or use AoP – but that's a whole other course)
- Code operations like logging, data access, as pluggagle implementations via interfaces
- You can really only add, not take away

Some rules for inheritance in C#

- Only use when you need to change the CLR level functionality by using virtual functions
- There are specific needs for providing a common functionality that has several points of either
  - Not being able to specify a behavior and forcing derived classes to provide an implementation (abstract methods)
  - Explicitly planning for specific types of functionality to be changed (override)
- Do not use inheritance to implement concerns / responsibilities

Good base classes plan for inheritance…

- Do not expose their internal state to the base classes (fields are private)
- Methods that are used for providing a different implementation of internal state should be protected virtual
- Methods that are for clients of the object only change internal state or defer to overrides
- Methods for use by other methods in this class only are private
- Methods for use in operations visible to derived classes, but not changeable, are protected
- Properties return a view of the internal state to the outside world

Good derived class characteristics

- Provide either or both of:
  - Overriding of specific virtual methods of the base class
  - Do not depend on internal state of a base class
    - If a base class even lets you do that, your are in trouble
  - Provide their own internal state (fields), external state (properties) and operations through various methods in accordance to the previous slide

.NET for C Programmers

Modeling Examples

Exercises / Demos

- Modeling an ordering system
  - Customer, order, product
  - Move through implementation of
    - Core classes
    - Abstraction
    - Relationships
    - Identity
    - Storage
    - Segregation of responsibility
  - Using OOP concepts

.NET for C Programmers

Preliminaries for SOLID: Clean Code and Smells

Overview

- Some examples of bad code and fixing it
- Conceptually we will be coding a message store that messages can be saved and loaded from
- We will look at a bunch of bad decisions and examine how to refactor them

Most Code Sucks

- What does this code do?
- What does **Save** return?
- We don’t know, so we have to look at source

```
0 references | mheydt, 1 day ago | 1 change
public class FileStore
{
    2 references | mheydt, 1 day ago | 1 change
    public string WorkingDirectory { get; set; }

    0 references | mheydt, 1 day ago | 1 change
    public string Save(int id, string message)...

    public event EventHandler<MessageEventArgs> MessageRead;

    0 references | mheydt, 1 day ago | 1 change
    public void Read(int id)...
}
```

Examining the code of Save

- Save returns a path of where something has been saved

```
0 references
public string Save(int id, string message)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    File.WriteAllText(path, message);
    return path;
}
```

What does Read do?

- Reads data in a file specified by id and then returns the contents via an event

```
0 references
public void Read(int id)
{
    var path = Path.Combine(this.WorkingDirectory, id + ".txt");
    var msg = File.ReadAllText(path);
    MessageRead(this, new MessageEventArgs(msg));
}
```

Well, that code sucks

- We needed to read the code to figure out what it does
  - That is really bad
- This hampers
  - Long-term productivity
  - Maintainability
- The point: Write code that is understandable

Revisiting encapsulation

- The literature says it is about information hiding
- I like to say it is more about implementing hiding
  - It is ok to expose information; fundamentally we need to at some point
  - But only expose what the client of the object requires, not any of the internal details

Invariants

- States that an object can never be in an invalid state
- How do we ensure this?
  - Pre and post condition checks
  - Using assertions
  - And run through automated tests

Command Query Separation

- Commands have side effects: they change state
- Queries only return data, no change of state

- CQS says that an action on an object should be one or the other, but not both

- http://en.wikipedia.org/wiki/Command%E2%80%93query_separation

Commands

- Mutate state
- Can invoke queries

```
void Save(Order order);

void Send(T message);

void Associate(
    IFoo foo, Bar bar);
```

- What’s common with all of these methods?
- Void: hence usually a command

Queries

- Do not mutate observable state
- Queries should be idempotent
  - Do it more than once, the state is the same
  - Re-asking the question does not change the answer

```
Order[] GetOrders(
    int userId);
```

- This returns something, so it is a query

Where is the Query?

- Is it Read()?
- Read() is both correct and incorrect
- This code sucks because the event is a side effect
- Read should not raise the event, we can remove it

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
        return path;
    }

    public event EventHandler<MessageEventArgs> MessageRead;

    public void Read(int id)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        var msg = File.ReadAllText(path);
        this.MessageRead(this, new MessageEventArgs { Message = msg });
    }
```

Where is the Command?

- Save()?
- but it returns a value – that is a smell
- Save does two things: saves and returns where the data was saved
- We fix this by adding a method that returns the filename for an id and make Save return void

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public string Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
        return path;
    }

    public event EventHandler<MessageEventArgs> MessageRead;

    public void Read(int id)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        var msg = File.ReadAllText(path);
        this.MessageRead(this, new MessageEventArgs { Message = msg });
    }
```

A good CQS class

- We can tell where the commands and queries are

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public void Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
    }

    public string Read(int id)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        var msg = File.ReadAllText(path);
        return msg;
    }

    public string GetFileName(int id)
    {
        return Path.Combine(this.WorkingDirectory, id + ".txt");
    }
```

CQS: Summary

- Makes it easier to REASON about code

Robustness

- Postel's law, aka the Robustness principle
  - Be conservative in what you send
  - Be liberal in what you accept
- Corollary…
  - The stronger the guarantee a method provides, the easier it is for a client to use it

What's wrong with this code?

- Think relative to robustness…

- WorkingDirectory can be null
- ID's can be negative

```
public class FileStore
{
    public string WorkingDirectory { get; set; }

    public void Save(int id, string message)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        File.WriteAllText(path, message);
    }

    public string Read(int id)
    {
        var path = Path.Combine(this.WorkingDirectory, id + ".txt");
        var msg = File.ReadAllText(path);
        return msg;
    }

    public string GetFileName(int id)
    {
        return Path.Combine(this.WorkingDirectory, id + ".txt");
    }
```

How do we fix WorkingDirectory?

- Add a constructor and not have a default constructor
- User now knows they have to set this value

```
public class FileStore
{
    public FileStore(string workingDirectory)
    {
        this.WorkingDirectory = workingDirectory;
    }

    public string WorkingDirectory { get; set; }

    public void Save(int id, string message)

    public string Read(int id)

    public string GetFileName(int id)
}
```

Also…

- Change the property to have a private setter
- Now, only the implementation can change the value

```
public string WorkingDirectory { get; private set; }
```

But what if the user passes null?

- We add a guard
- This “fails fast”

```
public FileStore(string workingDirectory)
{
    if (workingDirectory == null)
        throw new ArgumentNullException("workingDirectory");

    this.WorkingDirectory = workingDirectory;
}
```

Nullable types

- Ie: References in .NET
- Are evil!

- Later languages such as F# do not allow nulls

Ramification in OOP

- Because of null references (and other things…)

- You can not rely on the compiler to find all error
- We must build guards

What if the directory does not exist?

- The user can't plan for this
- Nor should they
- So we need another guard
  - But not create the directory – that is a side effect

```
public FileStore(string workingDirectory)
{
    if (workingDirectory == null)
        throw new ArgumentNullException("workingDirectory");
    if (!Directory.Exists(workingDirectory))
        throw new ArgumentException("Boo", "workingDirectory");

    this.WorkingDirectory = workingDirectory;
}
```

There is a smell in those guards

- They should have significantly better exception message
- They are a type of documentation

What’s wrong with this method?

- What if there is not a message/file with that id?
- How do we notify the caller it was bad?
- Note: we should not return null
  - That lowers our trust of this method
  - Which forces defensive coding

```
public string Read(int id)
{
    var path = this.GetFileName(id);
    var msg = File.ReadAllText(path);
    return msg;
}
```

How do we fix this?

- Three ways
  - Tester/Doer
  - TryRead
  - Maybe

Tester/Doer

- Add a "CanDo" method
- This exists in .NET
- framework

```
public bool Exists(int id)
{
    var path = this.GetFileName(id);
    return File.Exists(path);
}
```

- It has issues too:
  - Not thread safe
  - Exists could return true on thread, but another deletes the file between Exists and Read

Try/Read

- Also used in .NET framework classes
- IMHO: these suck

```
public bool TryRead(int id, out string message)
{
    message = null;
    var path = this.GetFileName(id);
    if (!File.Exists(path))
        return false;
    message = File.ReadAllText(path);
    return true;
}
```

Maybe

- Used as a return type
- Very explicit to the client that the method "Maybe" will return the value

```
public class Maybe<T> : IEnumerable<T>
{
    private readonly IEnumerable<T> values;

    public Maybe()
    {
        this.values = new T[0];
    }

    public Maybe(T value)
    {
        this.values = new[] { value };
    }

    public IEnumerator<T> GetEnumerator()
    {
        return this.values.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
}
```

We then change Read to…

- Maybe will hold either a result or be able to signify nicely that there is not a return value
- But, … it always returns a non-null reference!

```
public Maybe<string> Read(int id)
{
    var path = this.GetFileName(id);
    if (!File.Exists(path))
        return new Maybe<string>();
    var message = File.ReadAllText(path);
    return new Maybe<string>(message);
}
```

The client side code for maybe

```
var message = fileStore.Read(49).DefaultIfEmpty("").Single();
```

- What is this saying?
  - There is LINQ involved (which we cover tomorrow in detail)
  - But give us a default value of “” if the Maybe is empty,
  - Otherwise give us the single value represented in the Maybe object

- We have effectively provided to the client a robust and type-safe means of executing the method with trust.

Summary

- What code does should be understandable by the public API alone
- Try to make invalid states impossible
- Never return null
- CQS greatly exists in achieving these three

.NET for C Programmers

SOLID

What is SOLID?

- A set of principles to:
  - Make you more productive
  - By making code more maintainable
  - Through decomposition and decoupling

- It is not:
  - A framework
  - A library
  - A pattern

SOLID Design Smells

- SOLID is a reaction to a set of design smells
- It tries to address the following smells:

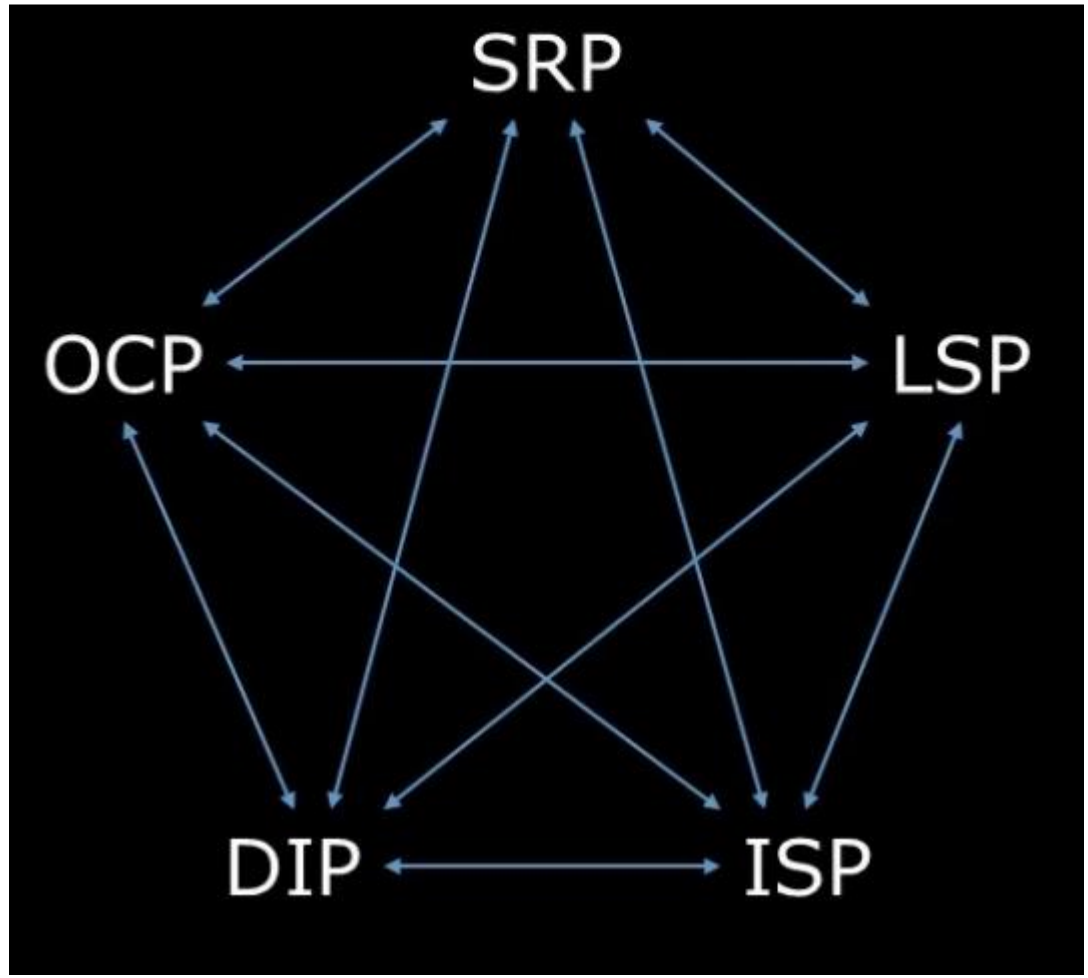
Small	
Rigidity	The design is difficult to change
Fragility	The design is easy to break
Immobility	The design is difficult to use
Viscosity	The design is difficult to do the right thing
Needless Complexity	The design is over-designed

Principles of SOLID

- **S**ingle Responsibility principle (SRP)
- **O**pen/closed principle (OCP)
- **L**iskov substitution principle (LSP)
- **I**nterface segregation principles (ISP)
- **D**ependency inversion principle (DIP)

Relationship of principles

- Independent but have strong relationships



.NET for C Programmers

SOLID Principles

A change to the code

- Better for the examples
- This is also actually common code
- But also very problematic
- We will see why

```
public void Save(int id, string message)
{
    Log.Information("Saving message {id}.", id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    Log.Information("Saved message {id}.", id);
}

public Maybe<string> Read(int id)
{
    Log.Debug("Reading message {id}.", id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        Log.Debug("No message {id} found.", id);
        return new Maybe<string>();
    }
    var message =
        this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName)
    Log.Debug("Returning message {id}.", id);
    return new Maybe<string>(message);
}
```

.NET for C Programmers

Segregation of Responsibility

Segregation of Responsibility Principle (SRP)

- A class should have one, and only one, reason to change
  - That one is that the one thing the class is supposed to do needs to change
- A class should do one thing and do it well

How many reasons are there to change in this code?

- Many
  - Storage
  - Logger
  - Cache
  - Orchestration

```
public void Save(int id, string message)
{
    Log.Information("Saving message {id}.", id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    Log.Information("Saved message {id}.", id);
}

public Maybe<string> Read(int id)
{
    Log.Debug("Reading message {id}.", id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        Log.Debug("No message {id} found.", id);
        return new Maybe<string>();
    }
    var message =
        this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName)
    Log.Debug("Returning message {id}.", id);
    return new Maybe<string>(message);
}
```

What we need to do

- Separate each reason for change into their own classes
- Each class has a single responsibility
    - MessageStore
    - StoreCache
    - StoreLogger
    - FileStore

StoreLogger

```
2 references
public class StoreLogger
{
    1 reference
    public void Saving(int id)
    {
        Log.Information("Saving message {id}.", id);
    }

    1 reference
    public void Saved(int id)
    {
        Log.Information("Saved message {id}.", id);
    }

    1 reference
    public void Reading(int id)
    {
        Log.Debug("Reading message {id}.", id);
    }

    1 reference
    public void DidNotFind(int id)
    {
        Log.Debug("No message {id} found.", id);
    }

    1 reference
    public void Returning(int id)
    {
        Log.Debug("Returning message {id}.", id);
    }
}
```

Now replace in FileStore

- First step
  - Replace all calls to Log in to objects

```
public void Save(int id, string message)
{
    new StoreLogger().Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    new StoreLogger().Saved(id);
}

public Maybe<string> Read(int id)
{
    new StoreLogger().Reading(id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        new StoreLogger().DidNotFind(id);
        return new Maybe<string>();
    }
    var message =
        this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName)
    new StoreLogger().Returning(id);
    return new Maybe<string>(message);
}
```

Well, we can refactor that

- Make it a field of the class

```
public class FileStore
{
    private readonly ConcurrentDictionary<int, string> cache;
    private readonly StoreLogger log;

    public FileStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new ConcurrentDictionary<int, string>();
        this.log = new StoreLogger();
    }
```

And update the usages

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    File.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message, (i, s) => message);
    this.log.Saved(id);
}

public Maybe<string> Read(int id)
{
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var message =
        this.cache.GetOrAdd(id, _ => File.ReadAllText(file.FullName))
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```

StoreCache

- Do the same for StoreCache

```
public class StoreCache
{
    private readonly ConcurrentDictionary<int, string> cache;

    public StoreCache()
    {
        this.cache = new ConcurrentDictionary<int, string>();
    }

    public void AddOrUpdate(int id, string message)
    {
        this.cache.AddOrUpdate(id, message, (i, s) => message);
    }

    public string GetOrAdd(int id, Func<int, string> messageFactory)
    {
        return this.cache.GetOrAdd(id, messageFactory);
    }
}
```

Now, what if we want to…

- Store data somewhere other than files?
- We will need to rewrite this
- What we really want is a “MessageStore” abstraction, where can be deferred to an specialization
- A client does not care how a MessageStore persists the message
- We then let the MessageStore decide how to persist / load messages

So, to do this…

- We create a MessageStore class
- And create a FileStore that MessageStore will utilize
- And have the MessageStore pick the storage implementation

The MessageStore

```
13 references
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    12 references | 0/12 passing
    public MessageStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }
```

MessageStore Save

- New implemantion

8 references | 0/7 passing

```
public void Save(int id, string message)
{
    this.log.Saving(id);
    var file = this.GetFileInfo(id);
    this.fileStore.WriteAllText(file.FullName, message);
    this.cache.AddOrUpdate(id, message);
    this.log.Saved(id);
}
```

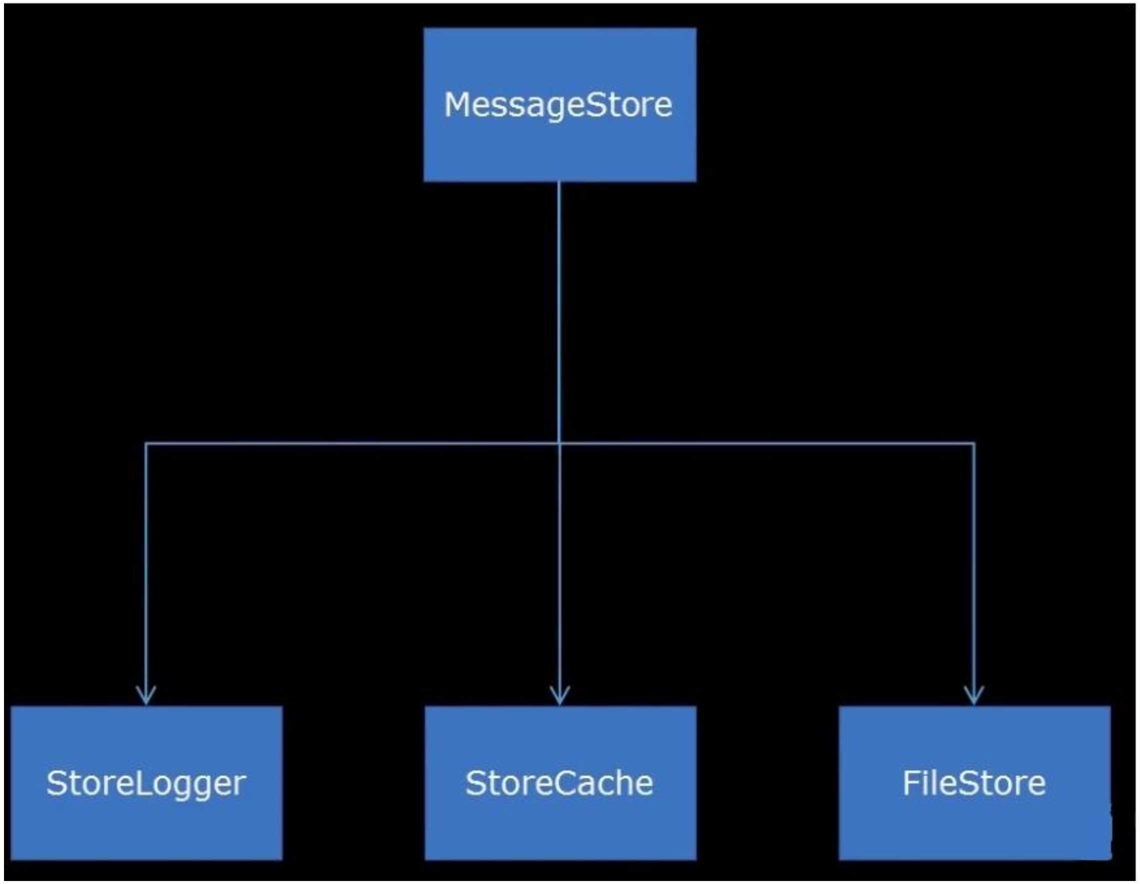
MessageStore Read

10 references | 0/8 passing

```
public Maybe<string> Read(int id)
{
    this.log.Reading(id);
    var file = this.GetFileInfo(id);
    if (!file.Exists)
    {
        this.log.DidNotFind(id);
        return new Maybe<string>();
    }
    var message = this.cache.GetOrAdd(
        id, _ => this.fileStore.ReadAllText(file.FullName));
    this.log.Returning(id);
    return new Maybe<string>(message);
}
```

We now have 4 classes

- Each with one concern



Summary

- We have come a long way with our model
- We identified four reasons to change, and ended up with four classes in our better solution
  - That is a good rule of thumb…
  - Each reason for change tends towards a different class
- We ended up with one class that has the required functionality: MessageStore
- And it can then select implementations

.NET for C Programmers

SOLID: Open/Close

Open/closed principle

- Classes should be:
  - Open for extension
  - Closed for modification
    - If you need to modify, build a new implementation conforming to a specific interface

Quote:

- Developers have a tendency to attempt to solve specific problems with general solutions
- This leads to coupling and complexity
- Instead of being general, code should be specific

- Greg Young

Therefore, following SRP:

- Each concrete class is very specific
- We saw this with FileStore, StoreLogger, StoreCache

But what if we need generality?

- How do we prevent duplicate code?
- Hence, need a more general API?

Example: Finer Grained Roles

- Abstracted to more generalized interfaces
- Much like Collections in .NET

```
public class FileStore : IStoreWriter, IStoreReader, IFileLocator
{
    public void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    public string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    public FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

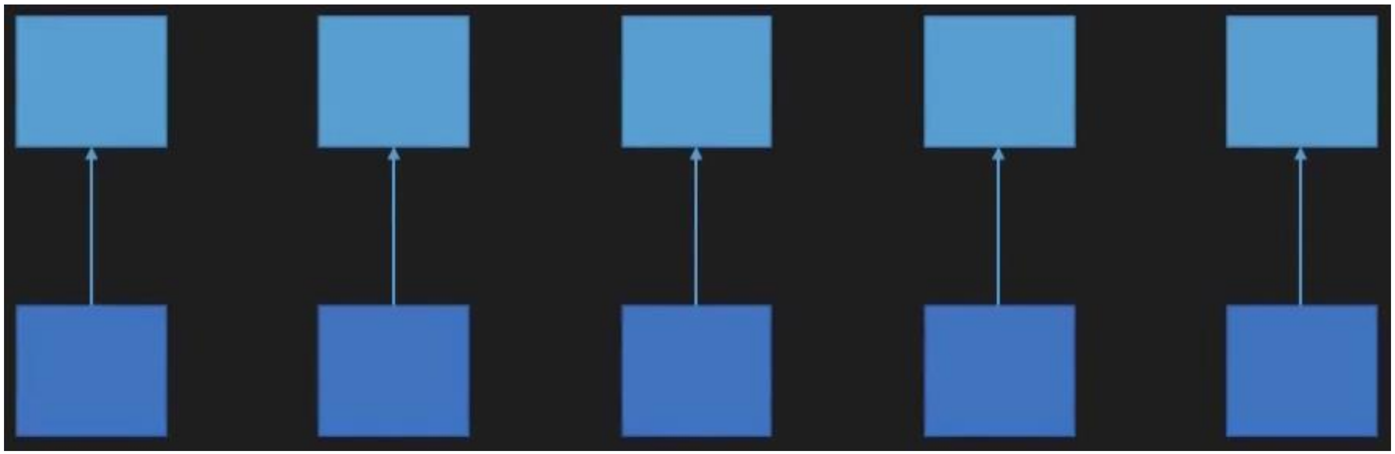
Point is…

- These are discovered, not necessarily designed
- And leads us to…

The Reused Abstractions Principle

- Not a part of SOLID, but useful
- If you have abstractions / interfaces, and are not being reused, then you have poor abstractions
- Abstraction is the elimination of the irrelevant and the amplification of the essential
    - Uncle Bob

Example of no Reuse

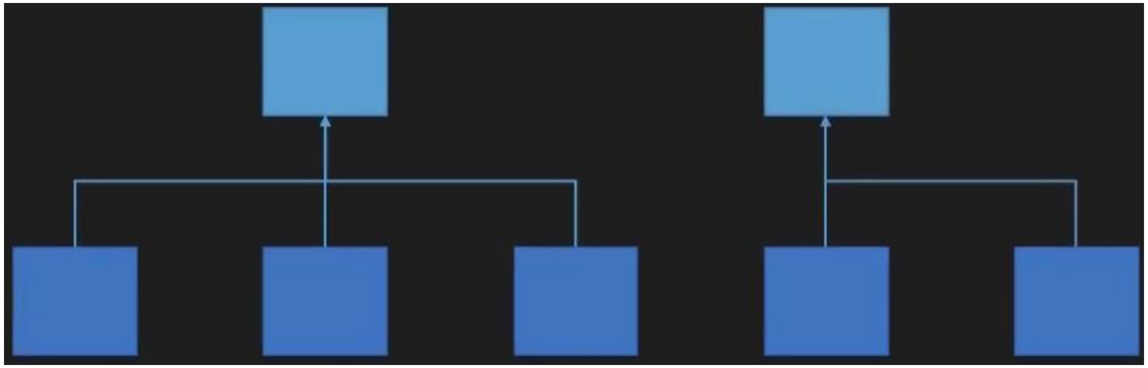


- One implementation of each interface
- This is violating Reused Abstractions Principle
- You have over designed, and the interfaces are too specifically designed

Start with the concrete and evolve

- From this

- To some common behavior



Rule of Three

- Don’t generalize until you find three cases of implementing the same interface
- Put another way, a sample size of two is not enough of a reason to generalize as you may create the wrong abstraction

Open/closed principle, revisited

- Classes should be:
  - Open for extensibility
  - Closed for modification
    - If you need to modify, build a new implementation conforming to a specific interface
    - Bug fixing is OK. Likely wont change behavior

- Open close was originally designed for inheritance
  - Now we favor composition over behavior

How do we make a class open for extensibility?

- Declare virtual functions
- We can subclass to change implementation

```
0 references
public class FileStore
{
    0 references
    public virtual void WriteAllText(string path, string message)
    {
        File.WriteAllText(path, message);
    }

    0 references
    public virtual string ReadAllText(string path)
    {
        return File.ReadAllText(path);
    }

    0 references
    public virtual FileInfo GetFileInfo(int id, string workingDirectory)
    {
        return new FileInfo(
            Path.Combine(workingDirectory, id + ".txt"));
    }
}
```

This has a problem…

- MessageStore is hard coded to a FileStore object
- Ie: It is closed for extension

```
13 references
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    12 references | 0/12 passing
    public MessageStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }
```

Making MessageStore open for extension

- Add factory properties
- Allow someone else to provide other implementations
- Make methods use the properties

```
protected virtual FileStore Store
{
    get { return this.fileStore; }
}

protected virtual StoreCache Cache
{
    get { return this.cache; }
}

protected virtual StoreLogger Log
{
    get { return this.log; }
}
```

```
public void Save(int id, string message)
{
    this.Log.Saving(id);
    var file = this.GetFileInfo(id);
    this.Store.WriteAllText(file.FullName, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}
```

Or virtualize the properties

```
3 references
protected virtual FileStore Store
{
    get { return this.fileStore; }
}

2 references
protected virtual StoreCache Cache
{
    get { return this.cache; }
}

5 references
protected virtual StoreLogger Log
{
    get { return this.log; }
}
```

```
public class MessageStore
{
    private readonly StoreCache cache;
    private readonly StoreLogger log;
    private readonly FileStore fileStore;

    12 references | 0/12 passing
    public MessageStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore();
    }
```

```
8 references | 0/7 passing
public void Save(int id, string message)
{
    this.Log.Saving(id);
    var file = this.GetFileInfo(id);
    this.Store.WriteAllText(file.FullName, message);
    this.Cache.AddOrUpdate(id, message);
    this.Log.Saved(id);
}
```

- We can subclass and provide another impl
- Or assign from outside

These however use Inheritance

- Not the preferred way, but how OOP originally taught us to do things
- We want to tend towards composition instead of inheritance
- We will come to a fix of this soon
- Onto LSP first

.NET for C Programmers

SOLID: Liskov Substitution Principle

Liskov substitution principle (LSP)

- Objects in a program should:
  - Be replaceable with instances of their subtypes
  - Without altering the correctness of the program

- Another way
  - Tests should not fail no matter what implementation is used

Creating an additional store

- We want to add an additional storage medium: SQL
- We will abstract out the interface from FileStore

```
4 references
public interface IStore
{
    3 references
    void WriteAllText(int id, string message);

    3 references
    Maybe<string> ReadAllText(int id);

    5 references
    FileInfo GetFileInfo(int id);
}
```

Derive FileStore from IStore

```
2 references
public class FileStore : IStore
{
    private readonly DirectoryInfo workingDirectory;

    1 reference
    public FileStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.workingDirectory = workingDirectory;
    }

    3 references
    public virtual void WriteAllText(int id, string message)
    {
        var path = this.GetFileInfo(id).FullName;
        File.WriteAllText(path, message);
    }
```

```
3 references
public virtual Maybe<string> ReadAllText(int id)
{
    var file = this.GetFileInfo(id);
    if (!file.Exists)
        return new Maybe<string>();
    var path = file.FullName;
    return new Maybe<string>(File.ReadAllText(path));
}

5 references
public virtual FileInfo GetFileInfo(int id)
{
    return new FileInfo(
        Path.Combine(this.workingDirectory.FullName, id + ".txt"));
}
```

Do the same for StoreCache

```
3 references
public interface IStoreCache
{
    2 references
    void AddOrUpdate(int id, string message);

    2 references
    Maybe<string> GetOrAdd(int id, Func<int, Maybe<string>> messageFactory);
}
```

```
2 references
public class StoreCache : IStoreCache
{
    private readonly ConcurrentDictionary<int, Maybe<string>> cache;

    1 reference
    public StoreCache()
    {
        this.cache = new ConcurrentDictionary<int, Maybe<string>>();
    }
```

Modify MessageStore to use IStoreCache and IStore

```
13 references
public class MessageStore
{
    private readonly IStoreCache cache;
    private readonly StoreLogger log;
    private readonly IStore fileStore;

    12 references | 0/12 passing
    public MessageStore(DirectoryInfo workingDirectory)
    {
        this.WorkingDirectory = workingDirectory;
        this.cache = new StoreCache();
        this.log = new StoreLogger();
        this.fileStore = new FileStore(workingDirectory);
    }
```

Now lets create SQLStore

- We can now swap this for FileStore
- But note that we break LSP in tests because of GetFileInfo ☹

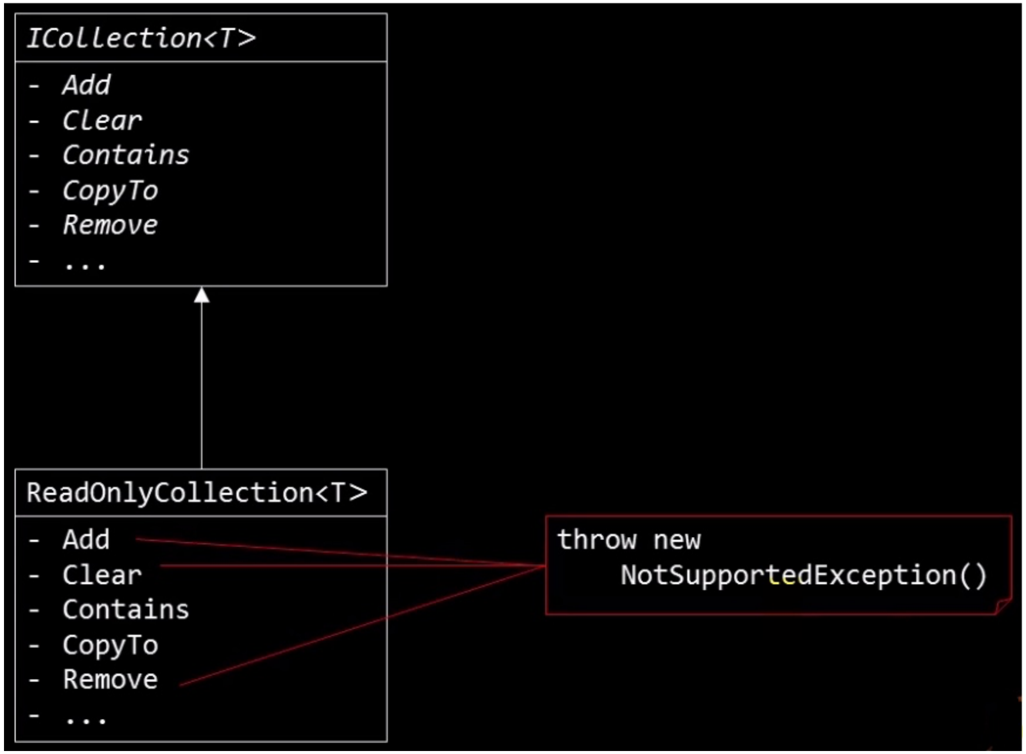
```
0 references
public class SqlStore : IStore
{
    3 references
    public void WriteAllText(int id, string message)
    {
        // Write to database here
    }

    3 references
    public Maybe<string> ReadAllText(int id)
    {
        // Read from database here
        return new Maybe<string>();
    }

    5 references
    public FileInfo GetFileInfo(int id)
    {
        throw new NotImplementedException();
    }
}
```

.NET example of Breaking the LSP

- .NET Collections when removing functionality
- Code depending on ICollection<T> could break if given a read only collection



Fixing our broken LSP

- We will take care of this in the next lesson on ISP

.NET for C Programmers

SOLID: Interface Segregation Principle

Interface segregation

- Many interfaces are large sets of functions
- We don't want clients to depend on methods they don't need
- Case in point: IStore
  - SqlStore does not need the file based methods
  - And it will break LSP
- Fundamentally adding is much more possible than removing
  - So if something is too small, its easier for fix that then to be too big

Interface Segregation Principle

- **Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not hierarchies.***
- ***Martin & Martin. *Agile Principles, Patterns, and Practices in C#*. Pearson Education, 2006.**
- **We should have granular interfaces that only include the members that a particular function needs.**

```
public class List<T> : IList<T>,
    ICollection<T>,IList, ICollection,
    IReadOnlyList<T>,IReadOnlyCollection<T>,
    IEnumerable<T>,IEnumerable
```

Let the clients define the interfaces

- SqlStore does not need GetFileInfo
- So we remove it from IStore

```
4 references
public interface IStore
{
    9 references
    void Save(int id, string message);

    3 references
    Maybe<string> ReadAllText(int id);
}
```

- And SqlStore no long needs to implement the method or throw the exception

```
0 references
public class SqlStore : IStore
{
    9 references
    public void Save(int id, string message)
    {
        // Write to database here
    }

    3 references
    public Maybe<string> ReadAllText(int id)
    {
        // Read from database here
        return new Maybe<string>();
    }
}
```

But now we have broken…

- Clients using the previous IStore that use GetFileInfo (like FileStore)
- We fix by creating a new interface IFileLocator

```
3 references
public interface IFileLocator
{
    4 references
    FileInfo GetFileInfo(int id);
}
```

And make FileStore…

- Also use IFileLocator

```
2 references
public class FileStore : IStore, IFileLocator, IStoreWriter
{
    private readonly DirectoryInfo workingDirectory;

    1 reference
    public FileStore(DirectoryInfo workingDirectory)
    {
        if (workingDirectory == null)
            throw new ArgumentNullException("workingDirectory");
        if (!workingDirectory.Exists)
            throw new ArgumentException("Boo", "workingDirectory");

        this.workingDirectory = workingDirectory;
    }
```

- Clients that need file info now use both interfaces
- And we don’t break LSP

.NET for C Programmers

SOLID: Dependency Inversion Principle

Dependency Inversion Principle

- Any object with a dependency should not be in control of picking the specific implementation
- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend on abstractions

- And favor composition over inheritance

Our previous code has a problem

- MessageStore is hard coded to an implementation through inheritance

We inject the dependencies

- Via a constructor
- And we can remove the virtual properties

```
6 references
public class MessageStore
{
    private readonly IFileLocator fileLocator;
    private readonly IStoreWriter writer;
    private readonly IStoreReader reader;

    3 references
    public MessageStore(
        IStoreWriter writer,
        IStoreReader reader,
        IFileLocator fileLocator)
    {
        if (writer == null)
            throw new ArgumentNullException("writer");
        if (reader == null)
            throw new ArgumentNullException("reader");
        if (fileLocator == null)
            throw new ArgumentNullException("fileLocator");

        this.fileLocator = fileLocator;
        this.writer = writer;
        this.reader = reader;
    }
```

Wow!

- That was simple!
- And we now have composable components

But…

- Who provides these objects to MessageStore?

- It is the responsibility of the client to pick the implementation that is desired.
- That's ok, it is made explicit by the constructor. It does not have to be guessed.
- And you will also provided default implementations for those dependencies
- But a client can make their own implementations
- And if SOLID has been followed, the program will still be "correct"

And…

- We will explore DI more in the last section of today’s course: DI and IoC with Ninject

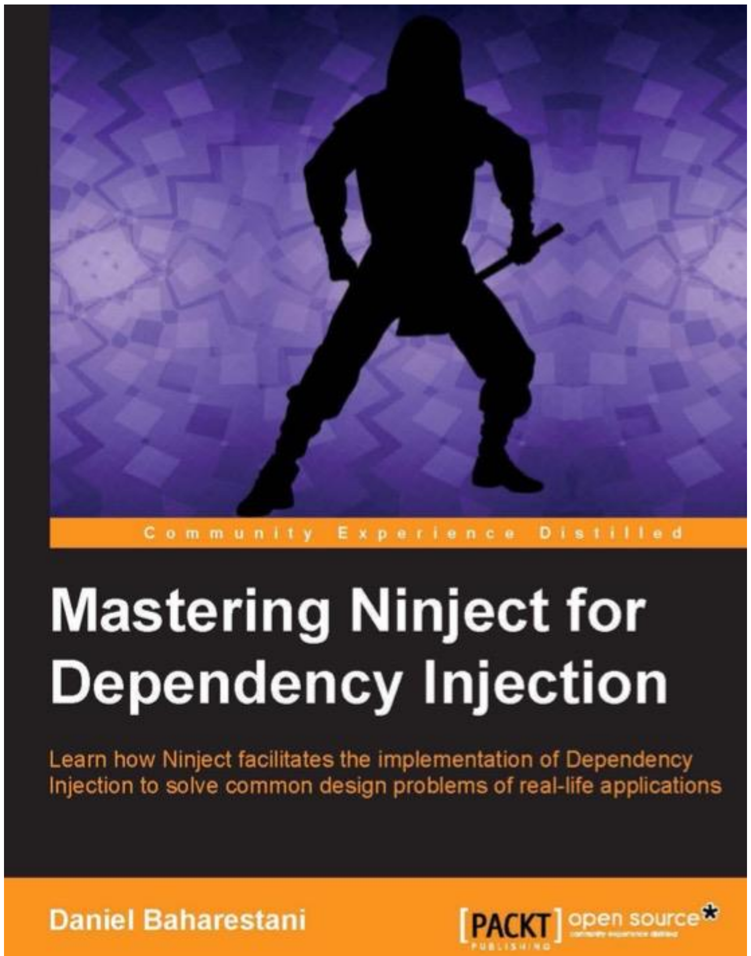
.NET for C Programmers

SOLID/DI/IoC using Ninject

IoC and DI with Ninject

1 – Dependency Inversion

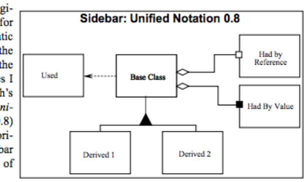
References



http://www.objectmentor.com/resources/articles/dip.pdf

The Dependency Inversion Principle

This is the third of my *Engineering Notebook* columns for *The C++ Report*. The articles that will appear in this column will focus on the use of C++ and OOD, and will address issues of software engineering. I will strive for articles that are pragmatic and directly useful to the software engineer in the trenches. In these articles I will make use of Booch's and Rumbaugh's new *unified* notation (Version 0.8) for documenting object oriented designs. The sidebar provides a brief lexicon of this notation.



Introduction

My last article (Mar, 96) talked about the Liskov Substitution Principle (LSP). This principle, when applied to C++, provides guidance for the use of public inheritance. It states that every function which operates upon a reference or pointer to a base class, should be able to operate upon derivatives of that base class without knowing it. This means that the virtual member functions of derived classes must expect no more than the corresponding member functions of the base class; and should promise no less. It also means that virtual member functions that are present in base classes must also be present in the derived classes; and they must do useful work. When this principle is violated, the functions that operate upon pointers or references to base classes will need to check the type of the actual object to make sure that they can operate upon it properly. This need to check the type violates the Open-Closed Principle (OCP) that we discussed last January.

In this column, we discuss the structural implications of the OCP and the LSP. The structure that results from rigorous use of these principles can be generalized into a principle all by itself. I call it "The Dependency Inversion Principle" (DIP).

1

Vocabulary

- DIP: Dependency Inversion Principle
- IoC: Inversion of Control
- DI: Dependency Injection
- IoC Container
- Interfaces
- Lifetime management

Dependency Inversion

- Instead of lower level modules defining an interface that higher level modules depend on…
- Higher level modules define an interface that lower level module implement

Don’t try and provide every type of connection

- Instead, make devices conform to an interface
  - Like USB

DEPENDENCY INVERSION PRINCIPLE

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

Basic Tenants

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend upon abstractions.

Net Effect

- Changes in lower layers do not impact code in higher layers

Summary: DI

- Systems are not layered – they are component based
  - Components provide a capability
- Capabilities are abstracted by interfaces
- Where implementations are found is not known by the user
  - “new” is evil
  - Builds in dependencies
  - And creates directional dependencies
- Any implementation can use capabilities of any other implementation
- Need to decide who creates and manages specific instances and lifetimes

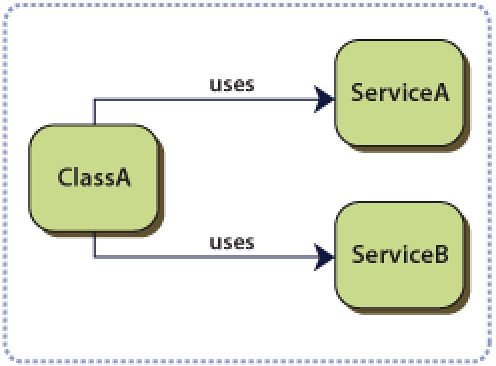
IoC and DI with Ninject

2 – Inversion of Control

What is Inversion of Control?

- A pattern of Dependency Management

- Different types
  - Control over interfaces between two components
  - Control over the flow of an application
  - Control over dependency creation and binding



Forces

- You want to decouple your classes from their dependencies so that the dependencies can be replaced or updated with minimal or no changes to your classes' source code.
- You want to write classes that depend on classes whose concrete implementations are not known at compile time.
- You want to test your classes in isolation, without using the dependencies.
- You want to decouple your classes from being responsible for locating and managing the lifetime of dependencies.

IoC vs DIP

- IoC is an implementation of DIP
- Many implementations of IoC / DIP
  - Unity, Castle, Ninject, StructureMap, …
- Flattening of hierarchy through interfaces
- Removal of dependency on physical implementation
- Rules engine for deciding
  - which implementations are used
  - And how object lifetimes are managed

IoC implementations (1)

- Factory pattern
  - In class-based programming, the **factory method pattern** is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exactclass of object that will be created. This is done by creating objects via calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.
  - http://en.wikipedia.org/wiki/Factory_%28object-oriented_programming%29

IoC implementations (2)

- Service Locator
  - The **service locator pattern** is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the "service locator", which on request returns the information necessary to perform a certain task.[1]
  - http://en.wikipedia.org/wiki/Service_locator_pattern

IoC Implementations (3)

- Dependency Injection
  - In software engineering, **dependency injection** is a software design pattern that implements inversion of control for software libraries, where the caller delegates to an external framework the control flow of discovering and importing aservice or software module. Dependency injection allows a program design to follow the dependency inversion principle where modules are loosely coupled. With dependency injection, the client part of a program which uses a module or service doesn't need to know all its details, and typically the module can be replaced by another one of similar characteristics without altering the client.
  - An injection is the passing of a dependency (a service) to a dependent object (a client). The service is made part of the client's state. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.
  - There are three common forms of dependency injection: setter-, interface- and constructor-based injection, where the responsibility of injecting the dependency lies upon the client, the service or the constructor method respectively.
  - http://en.wikipedia.org/wiki/Dependency_injection

Issues

- My code still needs to “get” objects
- I would prefer that I be given the objects so I don’t have to create them

- This is solved by Injection – module 3

Summary

- IoC lets us invert taking away the creation of objects
- By using an interface we don’t need to worry about the implementation
- Generally leverages factories
- But even knowing about factories is too much knowledge

IoC and DI with Ninject

3 – Dependency Inversion

What is Dependency Injection

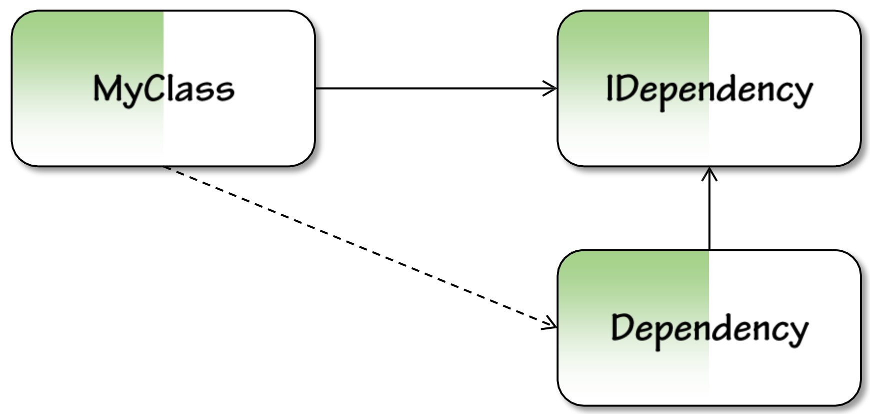
- A type of IoC where we move the creation and binding of a dependency to outside of a class that depends upon it

- Different types
  - Constructor
  - Property / setter
  - Interface

Where do they come from?

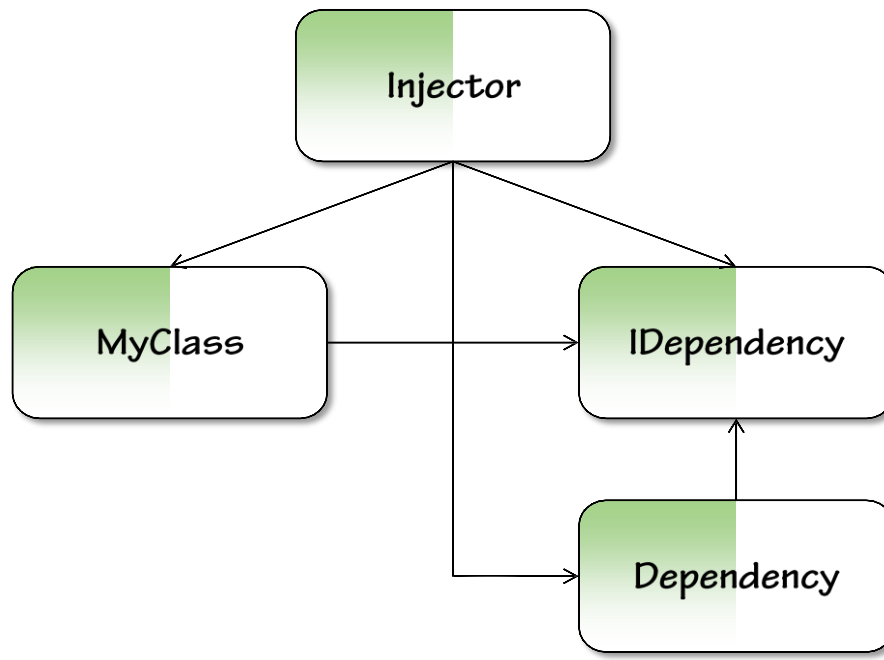
- I'm still dependent on the fact that I have to create objects

- So we create an "Injector"



The injection process

- The injector knows how to resolve dependencies
- It injects objects into another object



Constructor Injection

- Objects are passed into the constructor
- Now don’t need to explicitly create in the constructor

- But:
  - Can’t use class before things are injected
  - Bad for GUI’s and MVVM, or with async data

Basic example

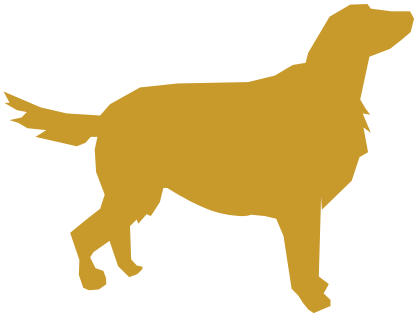
```
ICreditCard creditCard = new MasterCard();
Shopper shopper = new Shopper(creditCard);

public class Shopper
{
   private readonly ICreditCard creditCard;

   public Shopper(ICreditCard creditCard)
   {
      this.creditCard = creditCard;
   }
}
```

Setter Injection

- Objects set by injector via property setters
- Objects can be used before injection
- But need to know dependencies may not be resolved



Setter injection example

```
ICreditCard creditCard = new MasterCard();
Shopper shopper = new Shopper();
shopper.CreditCard = creditCard;

public class Shopper
{
   public ICreditCard CreditCard { get; set; }
}
```

Interface Injection

- Dependent class implements an Injection interface
- Injector uses this interface to set the dependency

Interface Injection Example

```
ICreditCard creditCard = new MasterCard(); Shopper
shopper = new Shopper();
((IDependOnCreditCard)shopper).Inject(creditCard);

public class Shopper : IDependOnCreditCard
{
   private ICreditCard creditCard;
   public void Inject(ICreditCard creditCard)
   {
      this.creditCard = creditCard;
   }
}

public interface IDependOnCreditCard
{
   void Inject(ICreditCard creditCard);
}
```

Caution

- Leaks internal implementation
- Prevents deferred creation
- Large object graphs
- Sometimes hard to know where objects came from

- Great for mocks though
- Easier unit testing

Summary

- Dependency injection lets us not need to know where objects come from
- We specify what objects we need by interface specification
- An external force then gives us objects
- These are containers

IoC and DI with Ninject

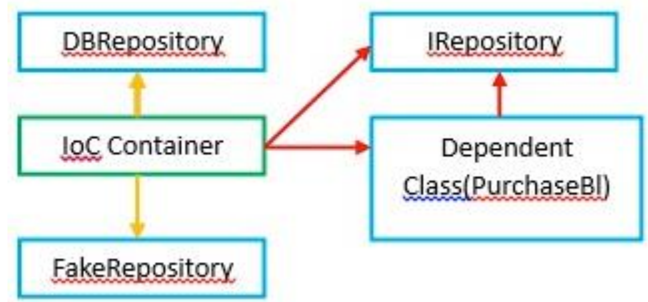
4 – Containers

Overview

- What is an IoC container?
- Manual constructor injection

What is a container?

- Manages the rules of dependency
- Creates objects
- Injects dependencies
- Manages object lifetime



Summary

- Ninject is a great IoC container
- Very flexible in options for creating objects
- Interception framework is great for AoP

IoC and DI with Ninject

5 – Using Ninject

Overview

- What is Ninject?
- Setting up the Container
- Using the container
- Managing Lifecycle
- Other features

What is Ninject?

- Newer
- Simple
- Extensible

Ninject Constructs

- StandardKernel
- .Bind.To
- .Get
- Named instances
- Lifecycle management
- Factory methods
- Events
- Modules

Our focus today…

- Not to cover all the details on Ninject
- Only enough to clean up our last module in SOLID

Kernels

- This is the container for the application – the “Injector”
- One default implementation: StandardKernel

```
var kernel = new StandardKernel();
```

Registrations

- Performed with .Bind<>()
- Fluent syntax
- .To<>() specifies target
- Can use names

```
kernel.Bind<IDataService>().To<MockDataService>();
```

Resolution

- .Get<>()

```
var service = kernel.Get<IDataService>();
```

Use the object

```
Console.WriteLine(service.ID);
var employees = service.GetEmployees();
employees.ToList().ForEach(
    e => Console.WriteLine("{0} {1}", e.FirstName, e.LastName));
```

Constructor injection

- Preferred model
- How to disambiguate constructors

```
public class MockConstructorInjectionViewModel
{
    0 references
    public MockConstructorInjectionViewModel(IDataService service)
    {
        Console.WriteLine("MockConstructorInjectionViewModel: {0} {1}",
            service.GetType().Name, service.ID);
    }
}
```

Property Injection

- Initialized after constructor executes
- Use the [Inject] attribute

```
public class MockPropertyInjectionViewModel
{
    private IDataService _service;
    [Inject]
    0 references
    public IDataService DataService
    {
        get { return _service; }
        set
        {
            _service = value;
            Console.WriteLine("MockPropertyInjectionViewModel.DataService[set] {0} {1}",
                _service.GetType().Name,
                _service.ID);
        }
    }
    0 references
    public MockPropertyInjectionViewModel()
    {
        Console.WriteLine("MockPropertyInjectionViewModel constructor");
    }
}
```

Summary

- Ninject is a great IoC container
- Very flexible in options for creating objects
- Interception framework is great for AoP

Demo

- Quick code demo of NInject

Lab

- Use Ninject to create a MessageStore and inject parameters.