Most Useful

Design Patterns

Steve Smith **(@ardalis)**

steve@ardalis.com

Ardalis.com

Podcast: WeeklyDevTips.com

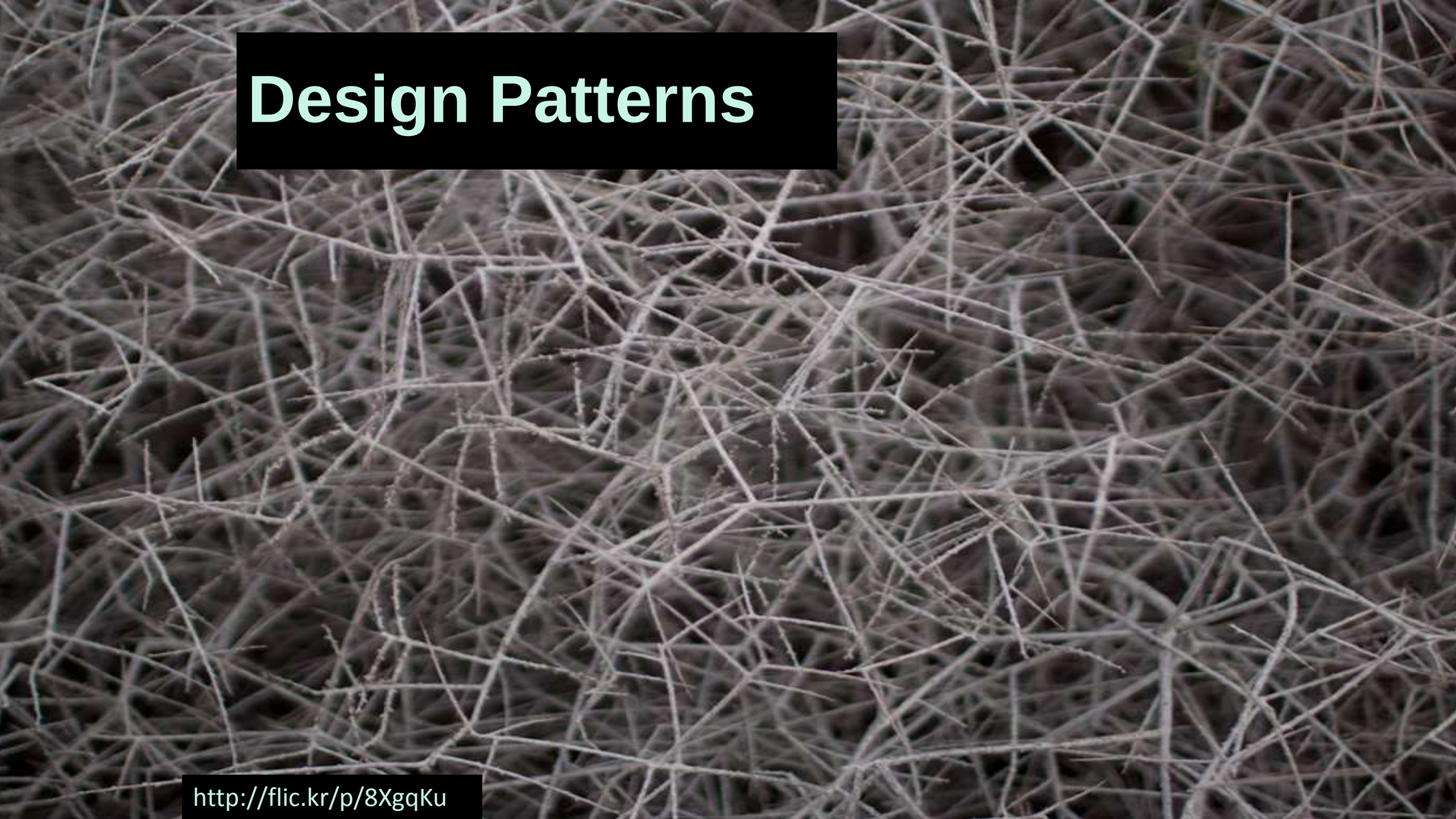
Are you too busy to improve?

Steve Smith

Courses by Steve

Domain-Driven Design Fundamentals	by Smith, Lerman	Intermediate	4h 16m	25 Jun 2014
Refactoring Fundamentals	by Steve Smith	Intermediate	8h 1m	13 Dec 2013
Creating N-Tier Applications in C#, Part 2	by Steve Smith	Intermediate	1h 40m	31 Dec 2012
Creating N-Tier Applications in C#, Part 1	by Steve Smith	Intermediate	2h 1m	17 Jul 2012
Kanban Fundamentals	by Steve Smith	Beginner	1h 31m	13 Feb 2012
Web Application Performance and Scalability Testing	by Steve Smith	Intermediate	3h 19m	27 Jul 2011
Design Patterns Library	by Steve Smith, et al.	Intermediate	15h 1m	10 Sep 2010
SOLID Principles of Object Oriented Design	by Steve Smith	Intermediate	4h 8m	10 Sep 2010

Design Patterns



http://flic.kr/p/8XgqKu

Stages of Learning

Stage Zero - Ignorance

Stage One - Awakening

Stage Two - Overzealous

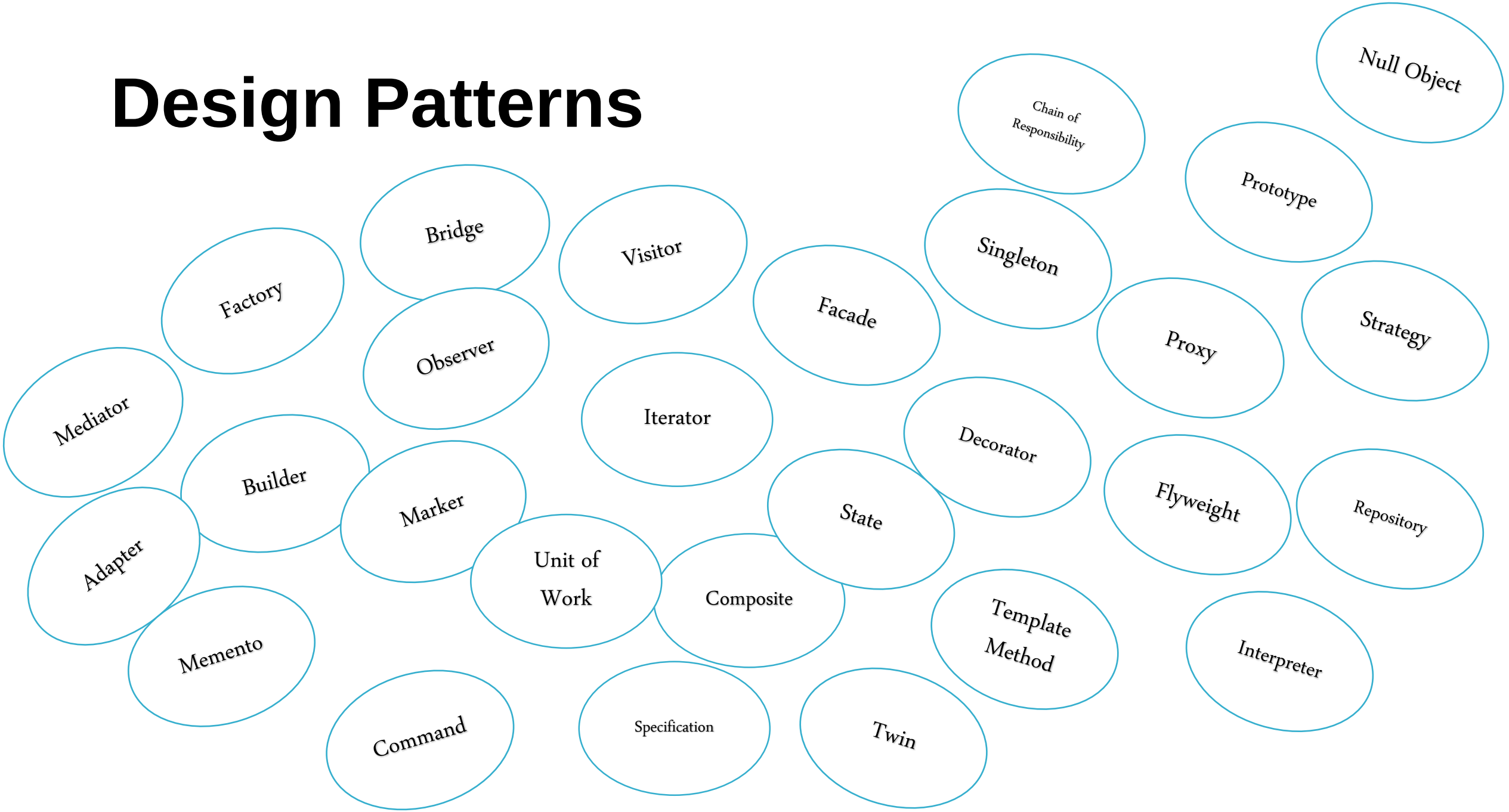
Stage Three – Mastery

http://flic.kr/p/6StgW5

Language

http://flic.kr/p/6UpMt9

Design Patterns



Most Useful (for web app design)

- (Singleton)
- Strategy
- Repository
- Proxy
- Command
- Factory
- Null Object

(Singleton)

HIGHLANDER

There can be only one.

Singleton: Intent

- **Ensure a class has only one instance**
- **Make *the class itself* responsible for keeping track of its sole instance**
- **“There can be only one”**

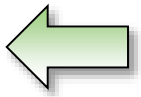
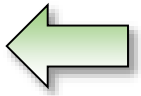
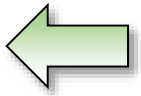
Singleton Structure (not thread-safe)

```
1  public class Singleton
2  {
3      private static Singleton _instance;
4
5      private Singleton()
6      {
7      }
8
9      public static Singleton Instance
10     {
11         get
12         {
13             if (_instance == null)
14             {
15                 _instance = new Singleton();
16             }
17             return _instance;
18         }
19     }
20 }
```

How Singleton Is Used

- **Call methods on Instance directly**
- **Assign variables to Instance**
- **Pass as Parameter**

```
public class SampleUsage
{
    public void SomeMethod()
    {
        // Call Singleton's DoStuff() method
        Singleton.Instance.DoStuff();

        // Assign to another variable
        var myObject = Singleton.Instance;
        myObject.DoStuff();

        // Pass as parameter
        SomeOtherMethod(Singleton.Instance);
    }

    private void SomeOtherMethod(Singleton singleton)
    {
        singleton.DoStuff();
    }
}
```

Singleton Structure (thread-safe and fast)

```
1  public class LazySingleton
2  {
3      private LazySingleton()
4      {
5      }
6
7      public static LazySingleton Instance
8      {
9          get { return Nested.instance; }
10     }
11
12     private class Nested
13     {
14         static Nested()
15         {
16         }
17
18         internal static readonly LazySingleton instance = new LazySingleton();
19     }
20 }
```

Source: http://csharpindepth.com/Articles/General/Singleton.aspx

Singleton Consequences

- **The default implementation is not thread-safe**
  - ☐ Avoid in multi-threaded environments
  - ☐ Avoid in web server scenarios (e.g. ASP.NET)

- **Introduce tight coupling among collaborating classes**

- **Singletons are notoriously difficult to test**
  - ☐ Commonly regarded as an ***anti-pattern***

Some commercial tools can be used to mock and test Singletons

But why would you intentionally write code you can only test with certain premium tools?

Singleton Consequences

- **Violate the *Single Responsibility Principle***
  - Class is responsible for managing its instances as well as whatever it does

- **Using an *IOC Container* it is straightforward to avoid the coupling and testability issues**

Managing Object Lifetime
Using an IOC Container (StructureMap)

```
// Configure EF
c.For<DbContext>().HybridHttpOrThreadLocalScoped().Use<PluralSightBookContext>();

// Configure NH
c.For<ISessionFactory>().Singleton().Use(() => DatabaseConfiguration.CreateSessionFactory());

// Configure Implementations
c.For<ISendEmail>().Use<DebugEmailSender>();
```

Object Lifetime in ASP.NET Core

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    // you don't need to implement singleton behavior in class!
    services.AddSingleton<ISomeService>();
}
```

Singleton *behavior* is fine and often desirable.

Implementing this behavior *using the Singleton Pattern on the class itself* is usually not ideal.

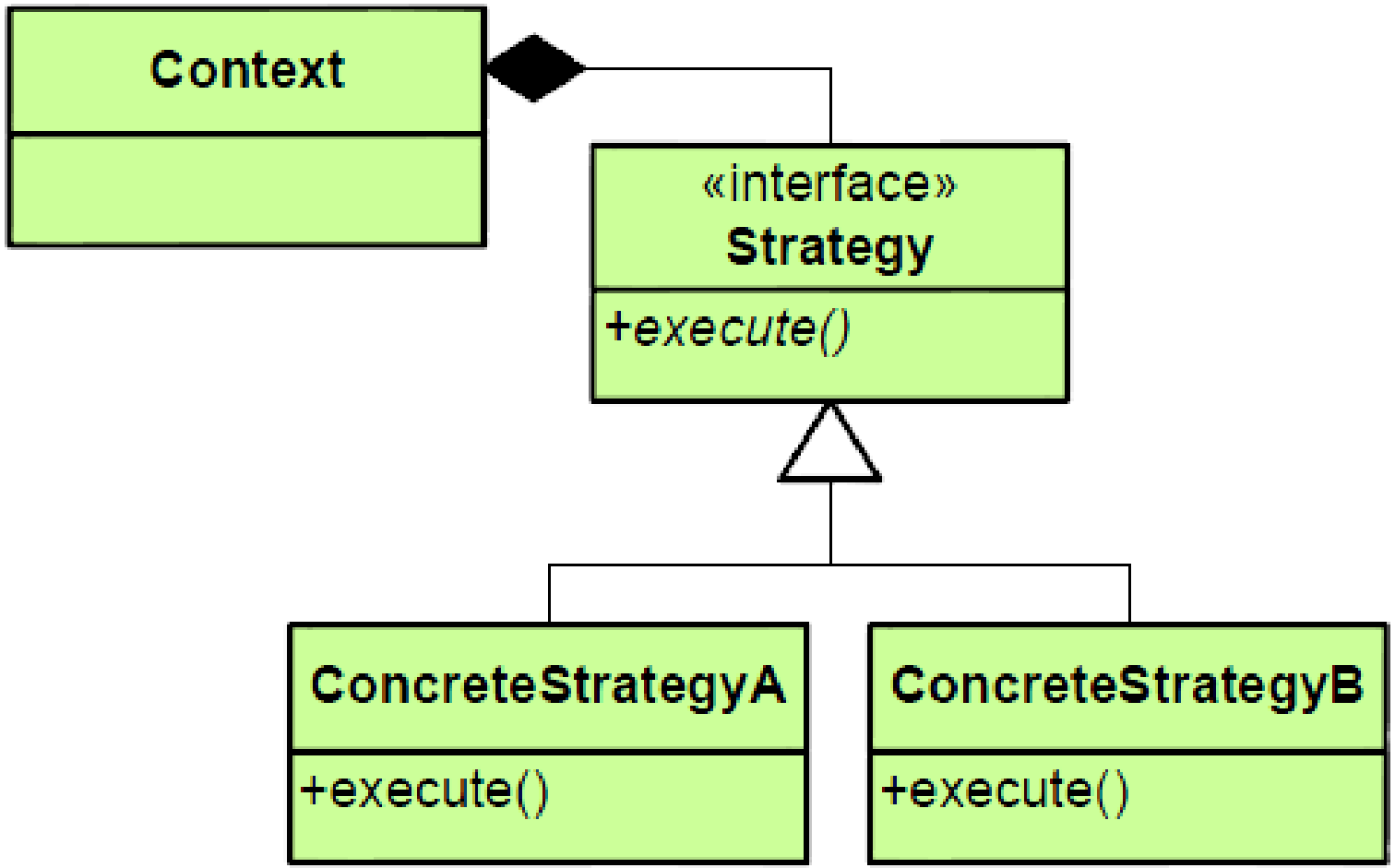
Strategy

http://flic.kr/p/6spkwo

Strategy: Intent

- **Encapsulate a family of related algorithms**
- **Let the algorithm vary and evolve independently from the class(es) that use it**
- **Allow a class to maintain a single purpose**
  - Single Responsibility Principle (SRP)
  - Also enables Open/Closed Principle (OCP)
  - Learn more about SRP and OCP in my Pluralsight course:
    - http://bit.ly/SOLID-OOP

Strategy : Structure



Strategy : Common Usage

- **Dependency Inversion and Dependency Injection**
- **Decouple class dependencies and responsibilities**

Refactoring Steps

- **Create interfaces for each responsibility**
- **Inject the implementation of the interface via the constructor**
- **Move the implementation to a new class that implements the interface**

Hidden Dependencies

- **Classes should declare their dependencies via their constructor**
  - Follow the *Explicit Dependencies Principle*
- **Avoid *hidden* dependencies**
  - Anything the class needs but which is not passed into constructor
- **Avoid making non-stateless static calls**
- **Avoid directly instantiating classes (except those with no dependencies, like strings)**
  - Instead, move these calls to an interface, and call a local instance of the interface

“New is Glue”

“new” creates tight coupling between classes

Be conscious of the consequences of using “new”

If you need loose coupling, replace “new” with Strategy Pattern

http://flic.kr/p/aN4Zv

Also works with WebForms

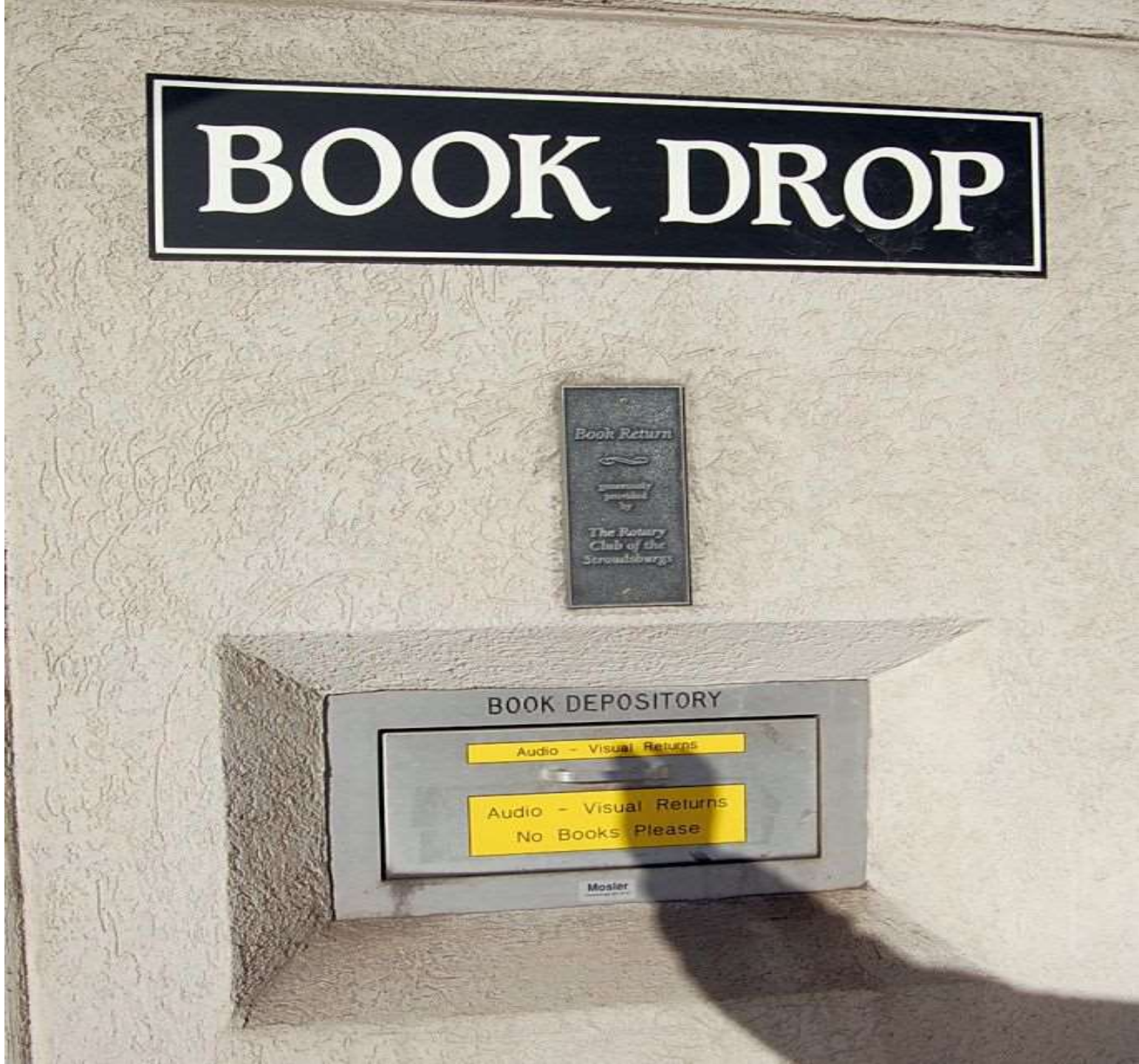
```
public class BasePage : System.Web.UI.Page
{
    public BasePage()
    {
        ObjectFactory.BuildUp(this);
    }
}
```

```
// for webforms
x.SetAllProperties(y =>
{
    y.OfType<IAlbumRepository>();
});
```

```
public partial class StructureMap : BasePage
{
    public IAlbumRepository AlbumRepository { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            //StructuremapMvc.Start();
            AlbumGrid.DataSource = AlbumRepository.GetTopSellingAlbums(5);
            AlbumGrid.DataBind();
        }
    }
}
```


"HOWEVER BEAUTIFUL THE STRATEGY, YOU SHOULD OCCASIONALLY LOOK AT THE RESULTS."
WINSTON CHURCHILL

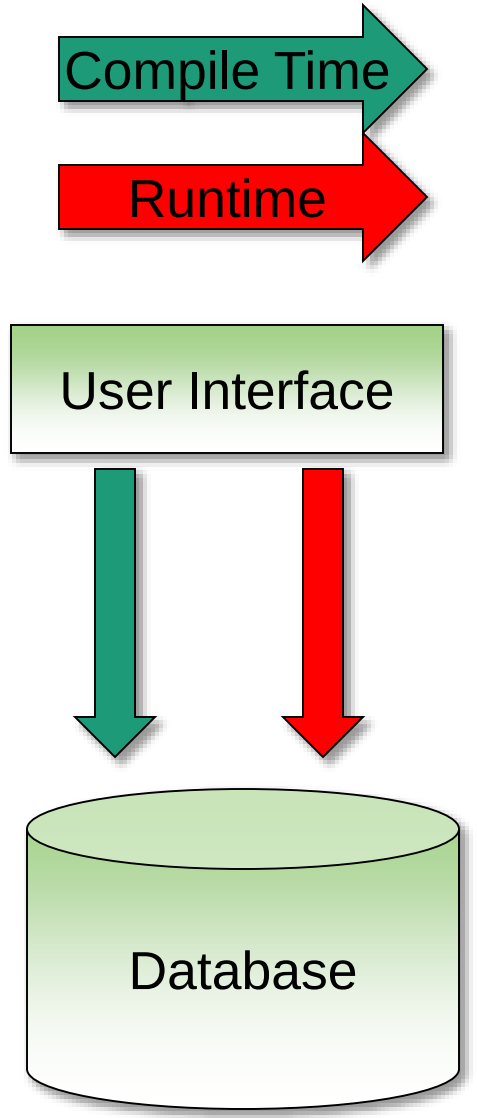
Repository


BOOK DROP
Book Return
The Rotary Club of the Stroudsburgs
BOOK DEPOSITORY
Audio - Visual Returns
Audio - Visual Returns
No Books Please
Mosler

Data Access Evolution

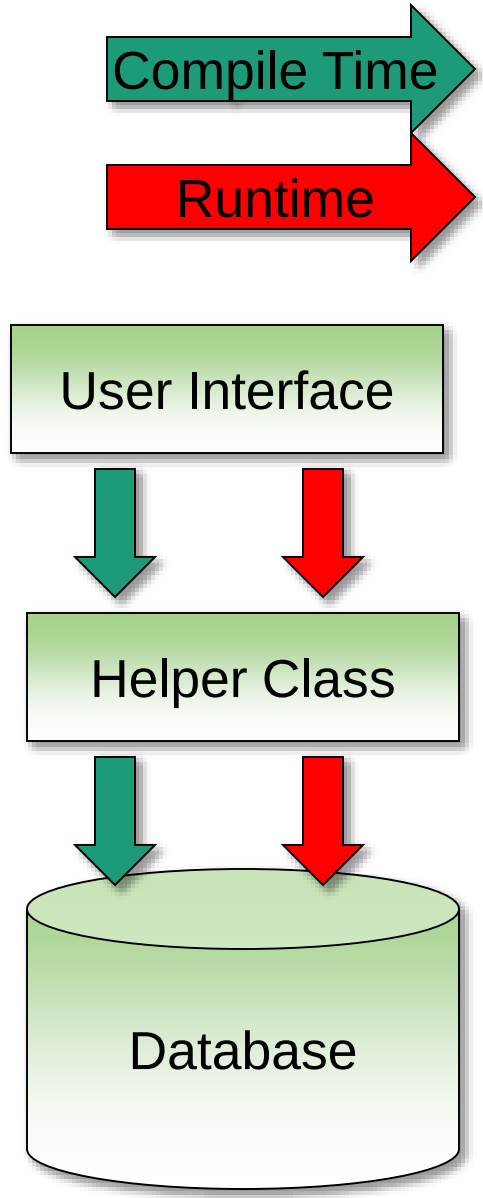
Initially no separation of concerns

- **Data access logic baked directly into UI**
  - ASP.NET Data Source Controls
  - Classic ASP scripts
- **Data access logic in UI layer via codebehind**
  - ASP.NET Page_Load event
  - ASP.NET Button_Click event



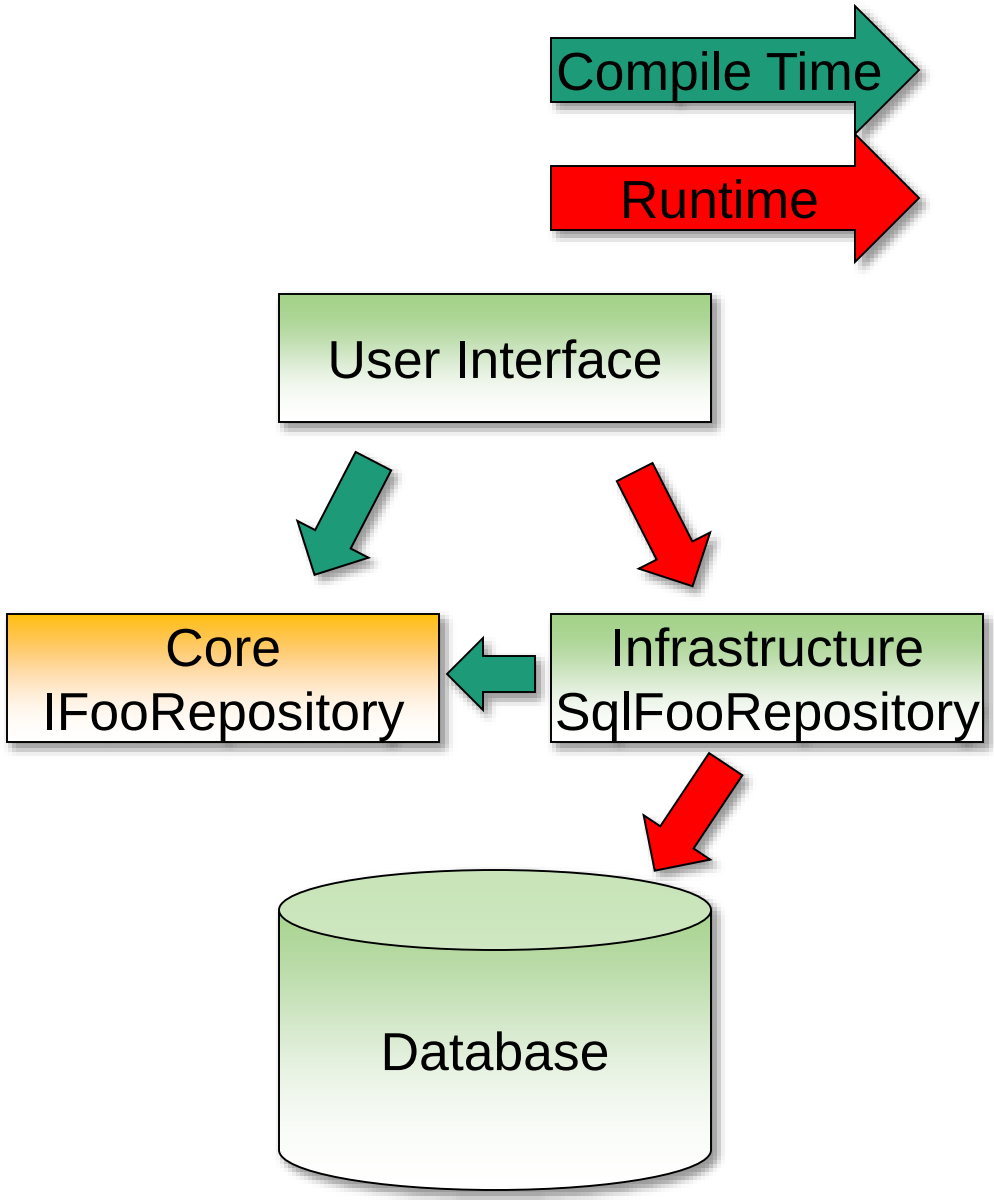
Data Access : Helper Classes

- Calls to data made through a utility
- Example: Data Access Application Block (SqlHelper)
- Logic may still live in UI layer
- Or a Business Logic Layer may make calls to a Data Access Layer which might then call the helper



What's Missing? Abstraction!

- **No way to abstract away data access**
- **Tight coupling**
- **Leads to *Big Ball of Mud* system**
- **Solution:**
  - Depend on interfaces, not concrete implementations
  - What should we call such interfaces? **Repositories!**



Repository

- **A Data Access Pattern**
  - Introduced as part of Domain-Driven Design, but very popular outside of DDD as well
- **Separates persistence responsibility from business classes**
- **Enables**
  - Single Responsibility Principle
  - Separation of Concerns
  - Testability

Repository

- **Frequently Separate Interfaces for Each Entity in Application**
  - E.g. CustomerRepository, OrderRepository, etc.
  - Or using Generics: IRepository<TEntity>

- **May also organize Repositories based on**
  - Reads vs. Writes
  - Bounded Contexts (a Domain-Driven Design concept)

Consider!

- **Repositories should return domain objects**
  - Don’t let persistence concerns leak through the repository interface
- **What should Repository List methods return?**
  - IEnumerable<T>
  - IQueryable<T>
- **What about deferred execution?**
- **What about lazy loading?**

Repository - Example

```
public class HomeController : Controller
{
    //
    // GET: /Home/

    MusicStoreEntities storeDB = new MusicStoreEntities();

    public ActionResult Index()
    {
        // Get most popular albums
        var albums = GetTopSellingAlbums(5);

        return View(albums);
    }

    private List<Album> GetTopSellingAlbums(int count)
    {
        // Group the order details by album and return
        // the albums with the highest count

        return storeDB.Albums
            .OrderByDescending(a => a.OrderDetails.Count())
            .Take(count)
            .ToList();
    }
}
```

Repository - Example

```
public interface IAlbumRepository
{
    IEnumerable<Album> GetTopSellingAlbums(int count);
}
```

(1)
Create/extend an interface to represent the data access you need

```
public class EfAlbumRepository : IAlbumRepository
{
    MusicStoreEntities storeDB = new MusicStoreEntities();

    public virtual IEnumerable<Album> GetTopSellingAlbums(int count)
    {
        Debug.Print("EfAlbumRepository:GetTopSellingAlbums");
        return storeDB.Albums
            .OrderByDescending(a => a.OrderDetails.Count())
            .Take(count)
            .ToList();
    }
}
```

(2)
Copy the implementation from your class into a new class implementing this interface.

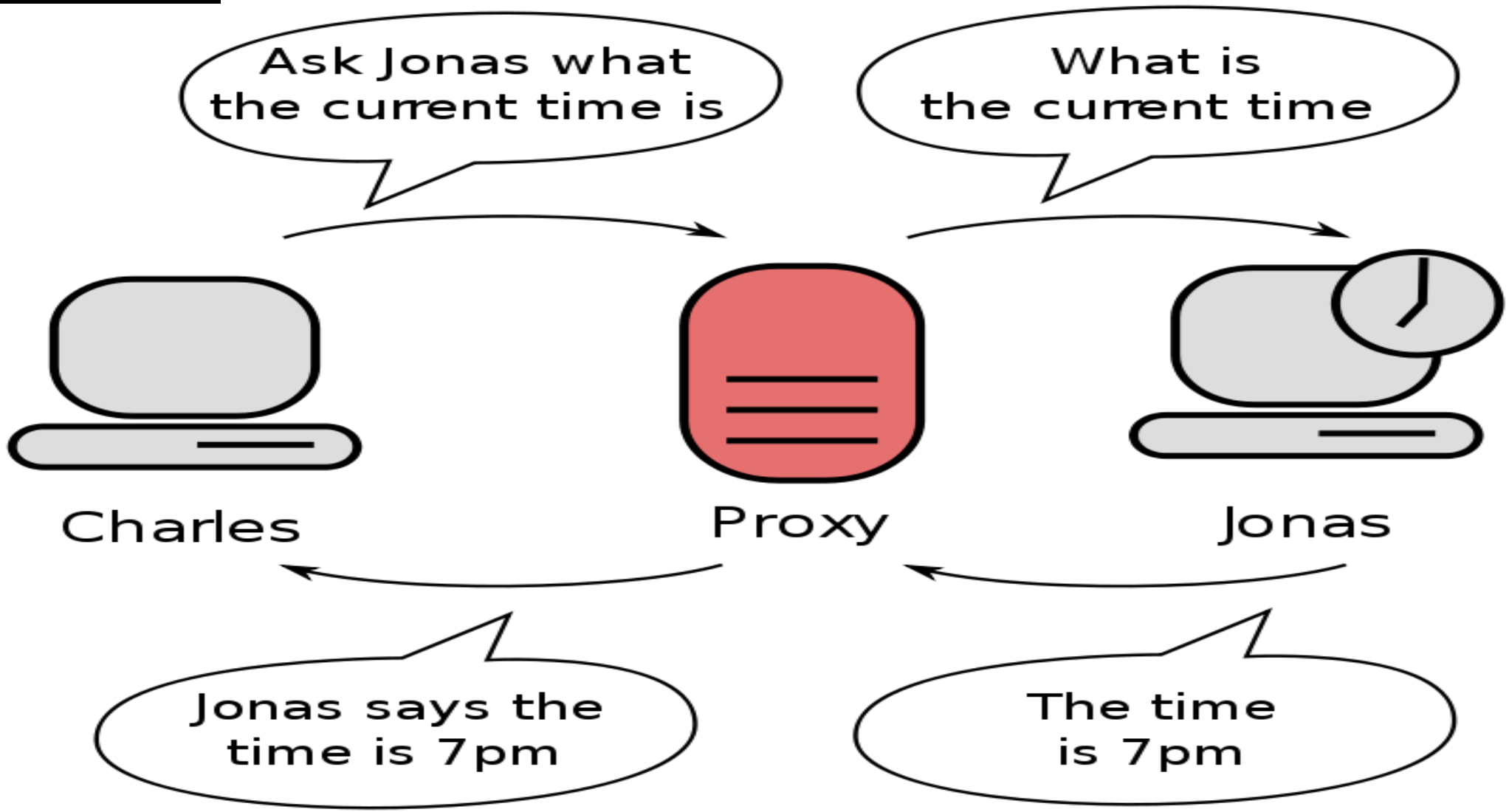
(3)
Inject the interface using the Strategy Pattern. Use the local field of this interface type in place of previous implementation code.

```
public class HomeController : Controller
{
    private readonly IAlbumRepository _albumRepository;

    public HomeController(IAlbumRepository albumRepository)
    {
        _albumRepository = albumRepository;
    }

    public ActionResult Index()
    {
        var albums = _albumRepository.GetTopSellingAlbums(5);

        return View(albums);
    }
}
```

Where do Repositories Live?

- **Place interfaces in Core**
  - Core includes Model classes and most business logic
  - Core has no dependencies (ideally)
- **Place implementation in Infrastructure**
  - Infrastructure references Core
- **UI layer (app entry point) references Core**
  - Accesses Infrastructure at runtime
  - Responsible for creating object graph, either manually or via an IOC Container

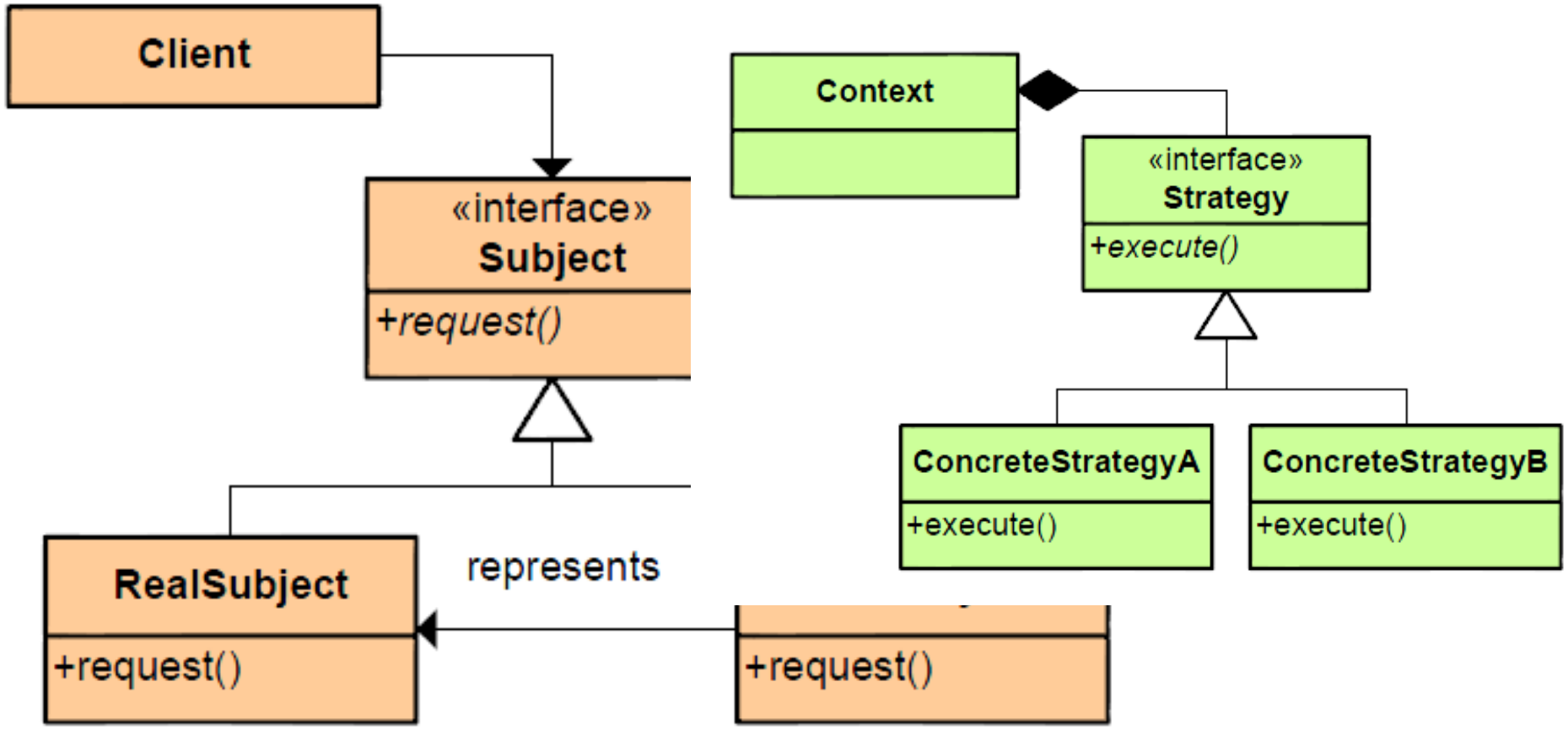
Proxy


Ask Jonas what
the current time is
What is
the current time
Charles
Proxy
Jonas
Jonas says the
time is 7pm
The time
is 7pm

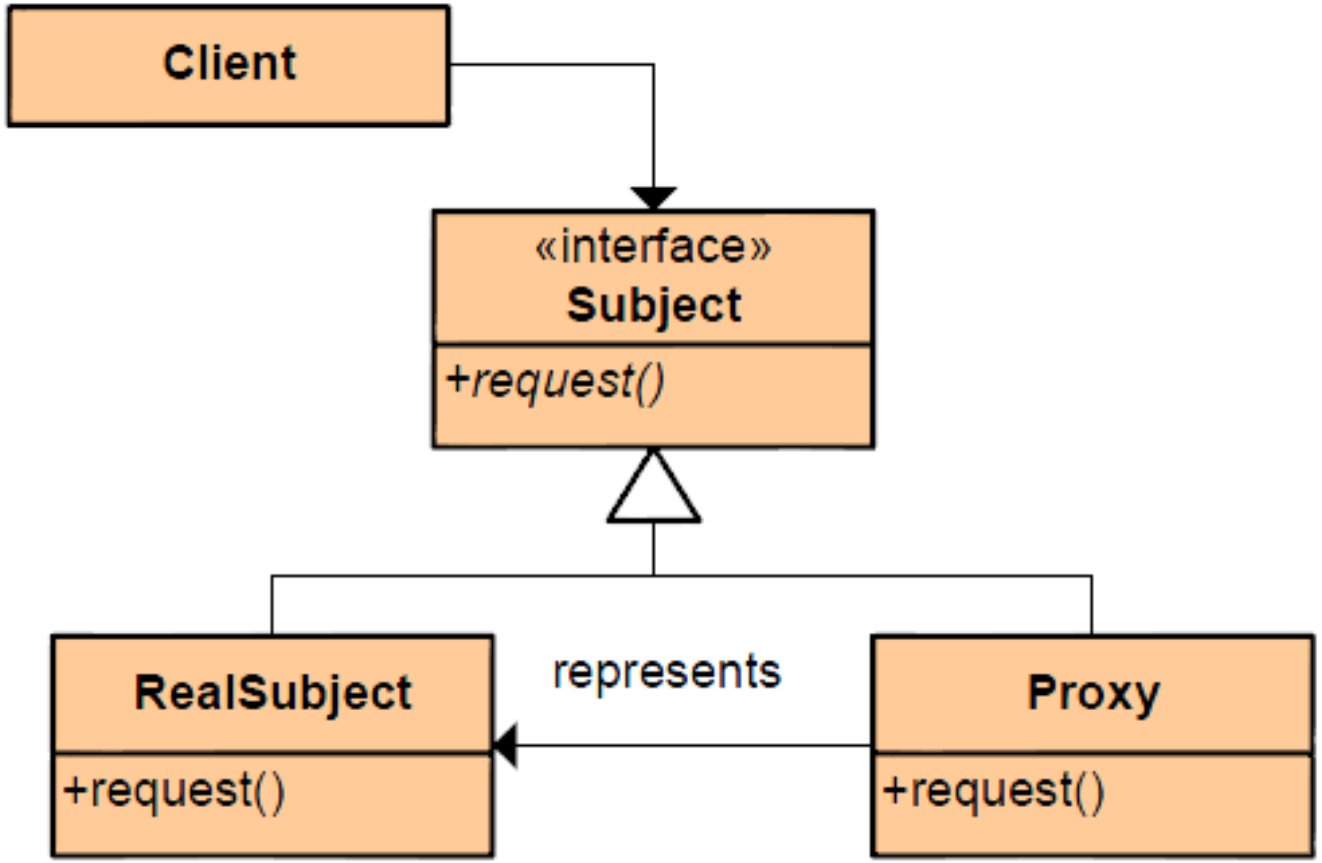
Proxy

- **A class that controls access to another**
- **Implemented via subclass or via delegation using a common interface**
- **Frequent use cases:**
  - Remote Proxy for web services / network calls
  - Lazy loading implementations
  - Security / access control

Proxy - Structure



Bonus Pattern: Decorator!



Stacking Patterns

Adding Caching to a Repository

- **Caching is frequently added to query methods in repositories**
- **Caching logic is a separate concern and should live in a separate class from the main data access logic**
- **Proxy pattern provides a straightforward means to control access to the “real” data, versus the cache**

Proxy – CachedRepository Implementation

```
public class CachedAlbumRepository : IAlbumRepository
{
    private readonly IAlbumRepository _albumRepository;

    public CachedAlbumRepository(IAlbumRepository albumRepository)
    {
        _albumRepository = albumRepository;
    }

    private static readonly object CacheLockObject = new object();
    public IEnumerable<Album> GetTopSellingAlbums(int count)
    {
        Debug.Print("CachedAlbumRepository:GetTopSellingAlbums");
        string cacheKey = "TopSellingAlbums-" + count;
        var result = HttpRuntime.Cache[cacheKey] as List<Album>;
        if (result == null)
        {
            lock (CacheLockObject)
            {
                result = HttpRuntime.Cache[cacheKey] as List<Album>;
                if (result == null)
                {
                    result = _albumRepository.GetTopSellingAlbums(count).ToList();
                    HttpRuntime.Cache.Insert(cacheKey, result, null,
                        DateTime.Now.AddSeconds(60), TimeSpan.Zero);
                }
            }
        }
        return result;
    }
}
```

Implementing with IOC (StructureMap)

// Doesn't work:

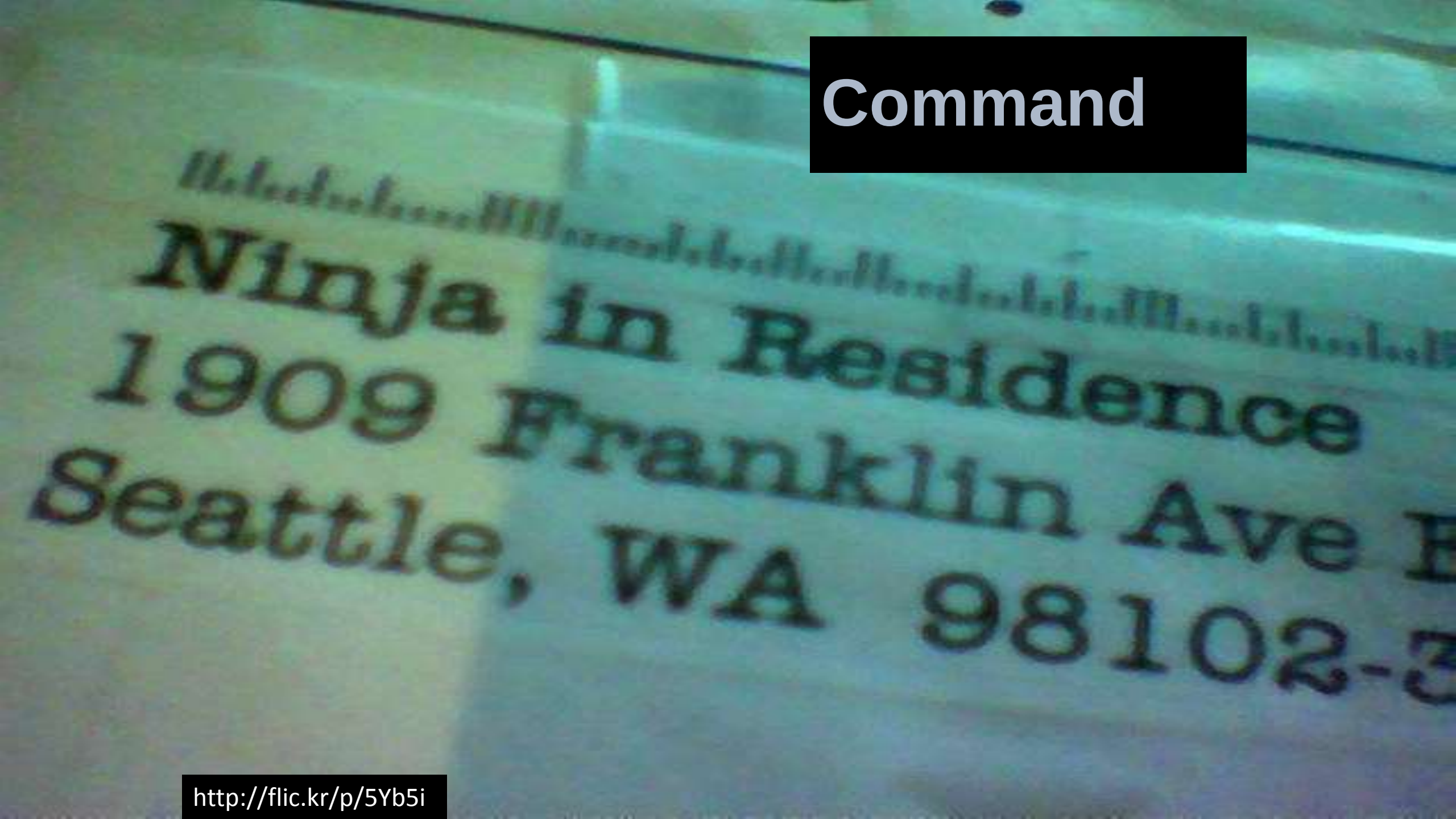
```
x.For<IAlbumRepository>().Use<CachedAlbumRepository>();
```

// Strategy Pattern with Proxy Pattern (using composition)

```
x.For<IAlbumRepository>().Use<CachedAlbumRepository>()
    .Ctor<IAlbumRepository>().Is<EfAlbumRepository>();
```

Implementing with IOC (.NET Core)

```
services.AddTransient<IDataService, DataService>((ctx) =>
{
   IOtherService svc = ctx.GetService<IOtherService>();
   return new DataService(svc);
});
```


Command
Ninja in Residence
1909 Franklin Ave E
Seattle, WA 98102-3
http://flic.kr/p/5Yb5i

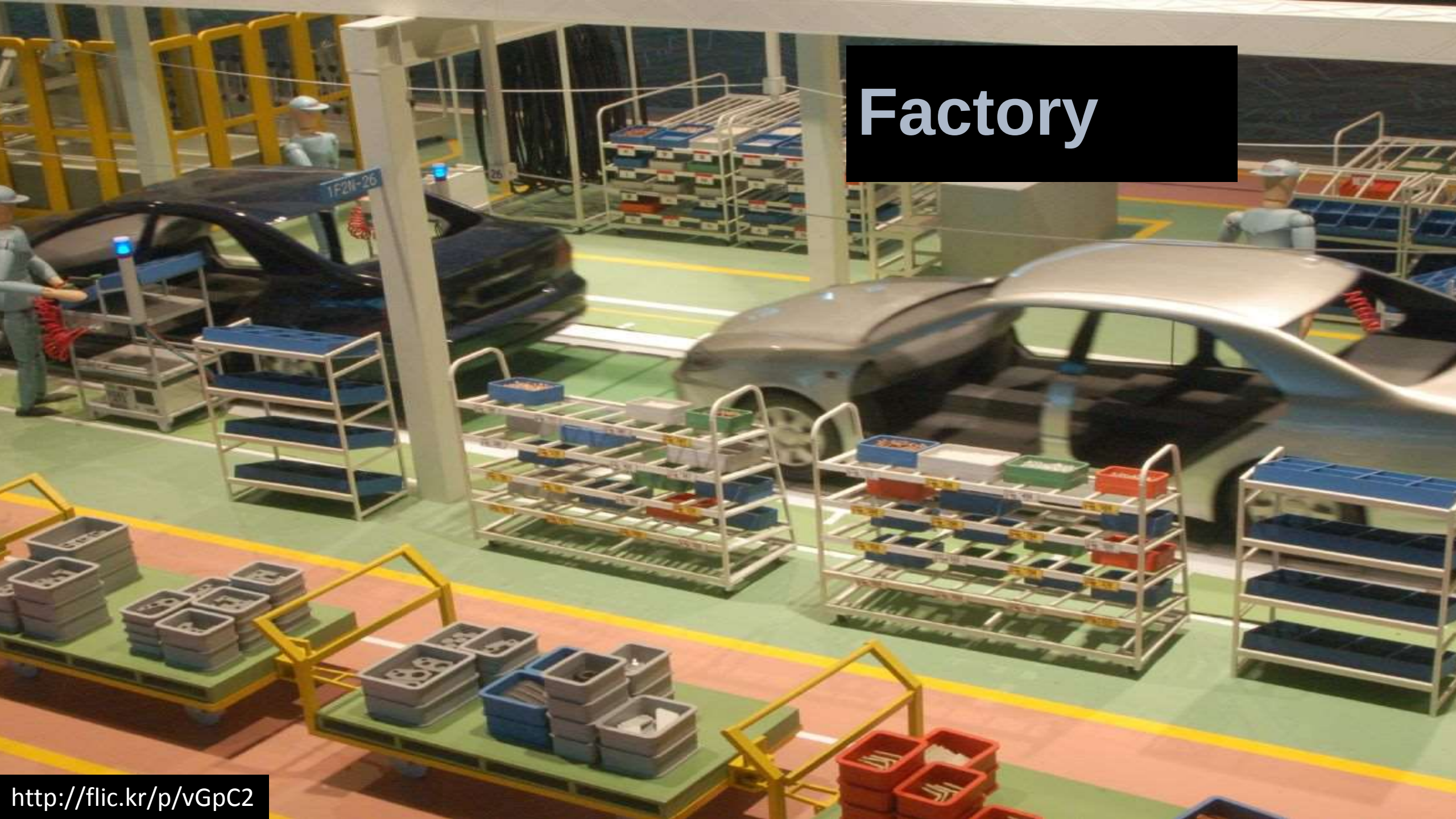
Command

- **Represent an action as an object**
- **Decouple performing the action from the client that is issuing the command**
- **A Command is a message**
- **Common scenarios:**
  - Delayed execution
  - Logging activity
  - Enabling Undo / Revoke Transaction functionality

Command : Usage

- **Combine with messaging for scalability**

- **What happens when you complete an order from Amazon?**


Factory
http://flic.kr/p/vGpC2

Factory: Intent

- **Separate object creation from the decision of which object to create**
- **Defer creation of objects**
  - Only create them if and when needed
- **Add new classes and functionality without breaking client code**
  - Only factory needs to know about new types available
- **Store possible objects to create outside of program**
  - Configuration
  - Persistent Storage
  - Plug-in assemblies

Lazy<T>

- **Provide simple built-in Factory behavior for lazy loading**
- **Available in .NET Framework 4, 4.5, later**
- **Learn more:**
- **http://msdn.microsoft.com/en-us/magazine/ff898407.aspx**

Without Lazy<T>

```
public class Order
{
    // other properties

    private Customer _customer;
    public Customer Customer
    {
        get
        {
            if (_customer == null)
            {
                _customer = new Customer();
            }
            return _customer;
        }
    }

    public string PrintLabel()
    {
        string result = _customer.CompanyName; // probably results in a NullReferenceException
        return result + "\n" + Customer.Address; // ok to access Customer
    }
}
```

With Lazy<T>

```
public class Order
{
    public Order()
    {
        _customerInitializer = new Lazy<Customer>(() => new Customer());
    }

    // other properties

    private Lazy<Customer> _customerInitializer;
    public Customer Customer
    {
        get
        {
            return _customerInitializer.Value;
        }
    }

    public string PrintLabel()
    {
        string result = Customer.CompanyName; // ok to access Customer
        return result + "\n" + _customerInitializer.Value.Address; // ok to access via .Value
    }
}
```

Alternate Factories

- **Write your own method**
- **Use an IOC Container**

> *"IOC Containers are basically just factories on steroids."*

- **Lambda expressions**
  - Func<ExpensiveObject> factory
  - () => new ExpensiveObject();

Null Object

Nulls

Nulls

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965.”

Sir Charles Antony Richard Hoare

Null Object

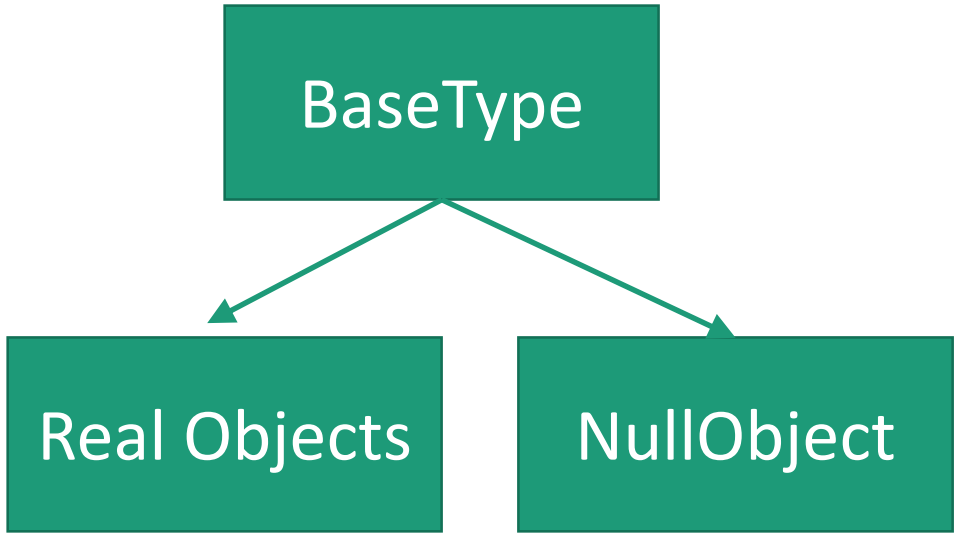
- Replace null with an actual instance object with default values
- Reduce number of null checks in application
- Allow methods on this “null” object to be called, without generating reference exceptions

Use When…

- **A method requires a collaborator**
  - And you want to ensure it is **never** null

- **It's necessary to support a "do nothing" option**
  - Especially with Command or Strategy

Implementation



The Null object is simply another implementation of a base type (class or interface)

For ease of use, it is often added as a static property on BaseType (e.g. BaseType.Null or BaseType.NotSet)

Nulls Break Polymorphism

- **Polymorphism is a key benefit of OO systems**

- **Null breaks polymorphism – calling methods on null instances results in exceptions**

```
public static void Main(string[] args)
{
    var employees = GetEmployees();

    foreach (var employee in employees)
    {
        Console.WriteLine("{0} {1}: {2} days",
          employee.FirstName,
    employee.LastName, employee.TenureInDays());
    }
    Console.ReadLine();
}
```

Null Checks Everywhere

```
var customer = repo.FindByPhone("555-555-5555");
string customerFormattedString = "";
if (customer != null)
{
    customerFormattedString = customer.ToFormattedString();
}
else
{
    customerFormattedString = "No customer found.";
}
Console.WriteLine(customerFormattedString);
```

With Null Obiect

Client code:

```
string customerDetails = repo.FindByPhone("555-555-5555").ToFormattedString();
Console.WriteLine(customerDetails);
```

```
public class Customer
{
    public static Customer NotFound = new NullCustomer();

    Other Properties and Methods
}
```

```
public class NullCustomer : Customer
{
    5 references
    public override string ToFormattedString()
    {
        return "Customer not found.";
    }
}
```

Repository:

```
public Customer FindByPhone(string phone)
{
    var customer = Customers.FirstOrDefault(c => c.Phone == phone);
    if (customer == null) return Customer.NotFound;
    return customer;
}
```

Practice PDD

- **Pain Driven Development**

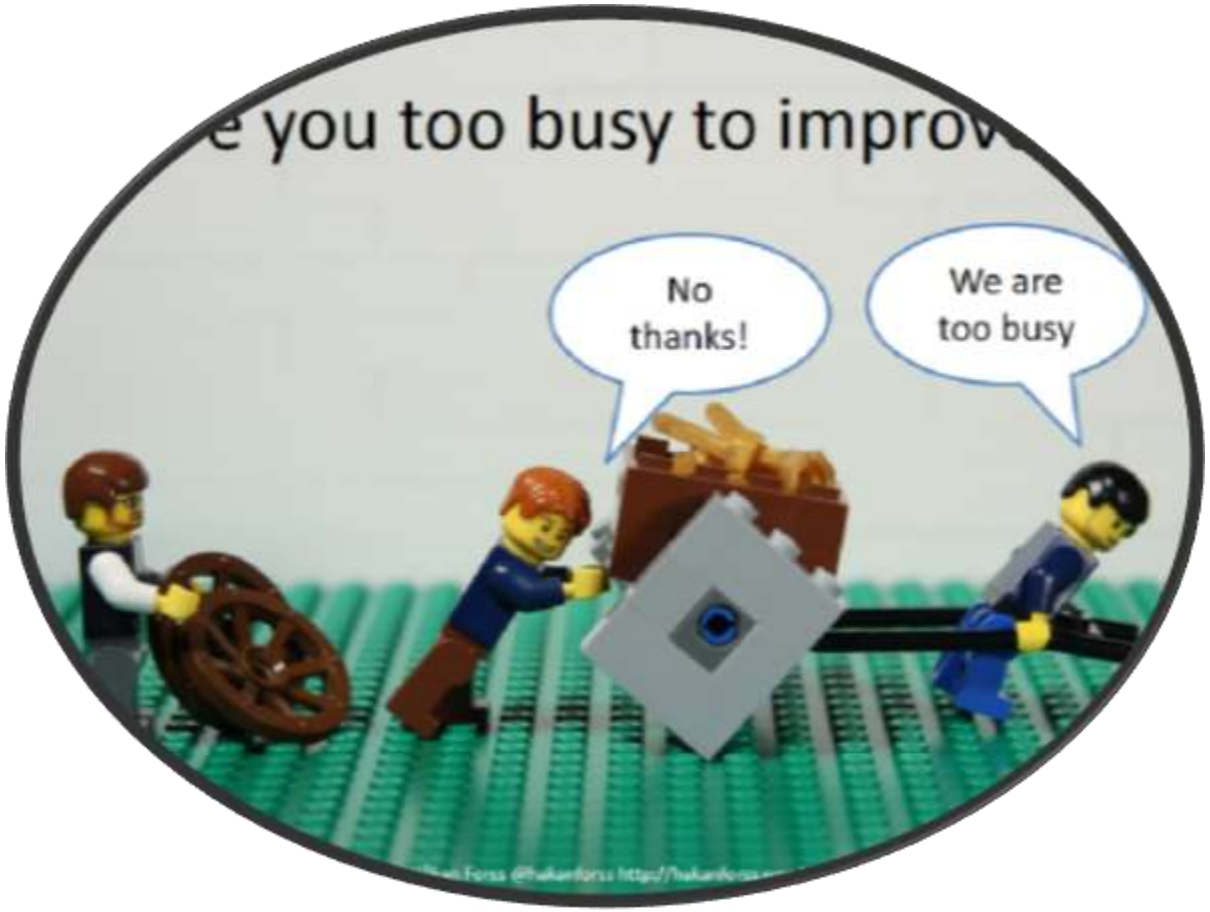
> *"Don't apply every pattern you know to a problem from the start; practice Pain Driven Development."*

If it's causing pain, fix it.

Let’s Review

Real World Design Patterns

1. Don’t be too busy to improve



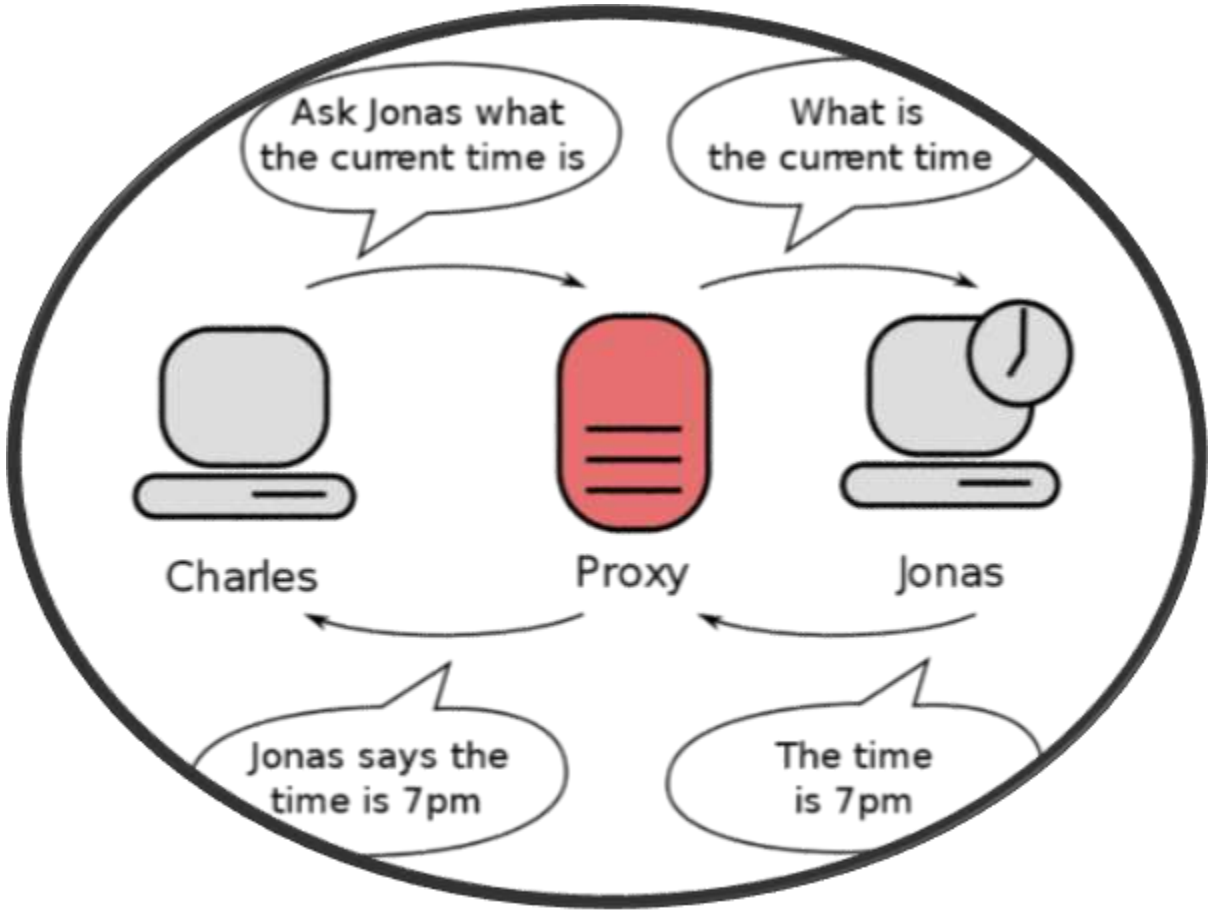
2. Avoid adding coupling via Singletons

3. Decouple specific implementation with Strategy

4. Avoid hidden dependencies – New is Glue!

5. Decouple your database with Repository

6. Decouple access control with Proxy



7. Decouple execution of other actions with Commands

8. Decouple construction with Factory

9. Reduce null checks with Null Object

10. Practice Pain Driven Development

References

- **Design Patterns, http://amzn.to/95q9ux**
- **Design Patterns Explained, http://amzn.to/cr8Vxb**
- **Design Patterns in C#, http://amzn.to/bqJgdU**
- **Head First Design Patterns, http://amzn.to/aA4RS6**

Pluralsight Resources

- **N-Tier Application Design in C# http://bit.ly/Msrwiq**
- **Design Patterns Library http://bit.ly/vyPEgK**
- **SOLID Principles of OO Design http://bit.ly/OWF4la**

- **My Blog**
  - http://ardalis.com

Discuss

Contact
steve@ardalis.com
Twitter: @ardalis
Web: ardalis.com
http://about.me/stevenasmith