Interfaces and Dynamic Loading

Jeremy Clark
DEVELOPER BETTERER

@jeremybytes www.jeremybytes.com

How & Why

- Focus on important functionality
- Remove details
- Run-time decisions
- Change behavior without recompiling
- Easier maintenance
- Easier unit testing

Program to an abstraction rather than a concrete type

Program to an interface rather than a concrete class

```
private void FetchButton_Click(object sender, RoutedEventArgs e)
{
  ClearListBox();

  IPersonRepository repository = RepositoryFactory.GetRepository();

  var people = repository.GetPeople();

  foreach (var person in people)
    PersonListBox.Items.Add(person);
}
```

Program to an Interface

No references to concrete repository types

Compile-time Factory

```
IPersonRepository GetRepository(string repositoryType) {

  IPersonRepository repository = null;

  switch (repositoryType) {
    case "Service": repository = new ServiceRepository();
      break;
    case "CSV": repository = new CSVRepository();
      break;
    case "SQL": repository = new SQLRepository();
      break;
  }
  return repository;
}
```

Factory Comparison

Compile-time Factory	Dynamic Factory
Has a parameter	No parameter
Caller picks the repository	Repository based on configuration
Compile-time reference	No compile-time references
	Decisions made at run-time

```
public static IPersonRepository GetRepository()
{
    string repositoryTypeName =
        ConfigurationManager.AppSettings["RepositoryType"];
    Type repositoryType = Type.GetType(repositoryTypeName);
    object repository = Activator.CreateInstance(repositoryType);
    IPersonRepository personRepository =
        repository as IPersonRepository;
    return personRepository;
}
```

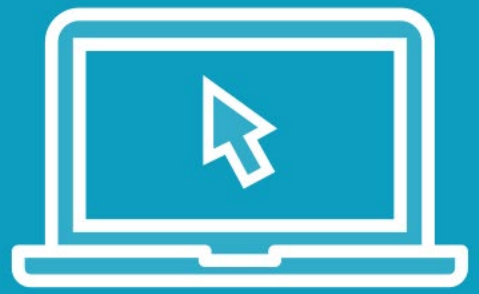
Dynamic Loading

Get Type and assembly from configuration

Load assembly through reflection

Create a repository instance with the Activator

Demo



Add dynamic loading code

No compile-time references

Change repository without recompiling

Unit Testing

Testing pieces of functionality in isolation

Interfaces help us isolate
code for easier unit testing.

What We Want to Test

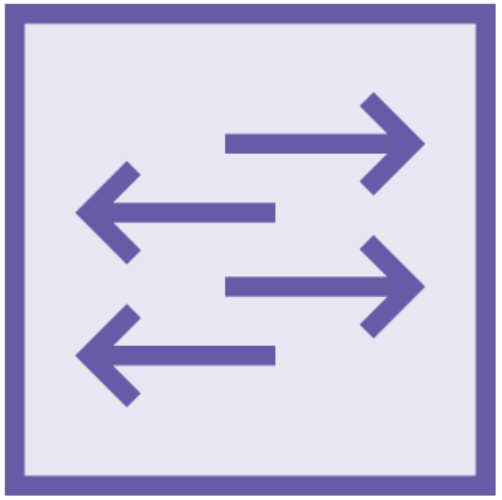
```
public partial class MainWindow : Window
{
    private void FetchButton_Click(object sender, RoutedEventArgs e)
    {
        ClearListBox();

        IPersonRepository repository =
            RepositoryFactory.GetRepository();

        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);

        ShowRepositoryType(repository);
    }
    ...
}
```

Dependent Objects

```
public partial class MainWindow : Window
{
    private void FetchButton_Click(object sender, RoutedEventArgs e)
    {
        ClearListBox();

        IPersonRepository repository =
            RepositoryFactory.GetRepository();

        var people = repository.GetPeople();
        foreach (var person in people)
            PersonListBox.Items.Add(person);

        ShowRepositoryType(repository);
    }
    ...
}
```

Current Application

Application



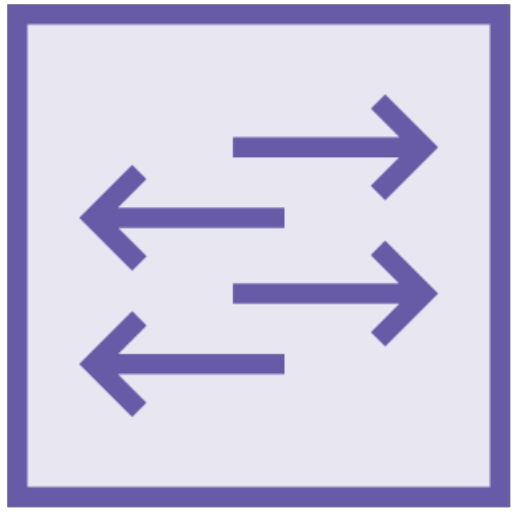
Repository

Data store

Application with View Model

View
(UI elements)

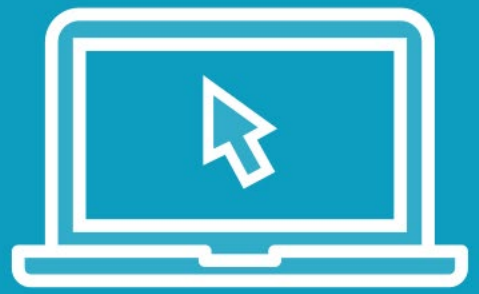
View model
(UI logic)



Repository

Data store

Demo



Move functionality to a view model

Add a fake repository for tests

Unit test the view model functionality

How & Why

- Focus on important functionality
- Remove details
- Run-time decisions
- Change behavior without recompiling
- Easier maintenance
- Easier unit testing