XAM250

Patterns for Cross Platform Mobile Development

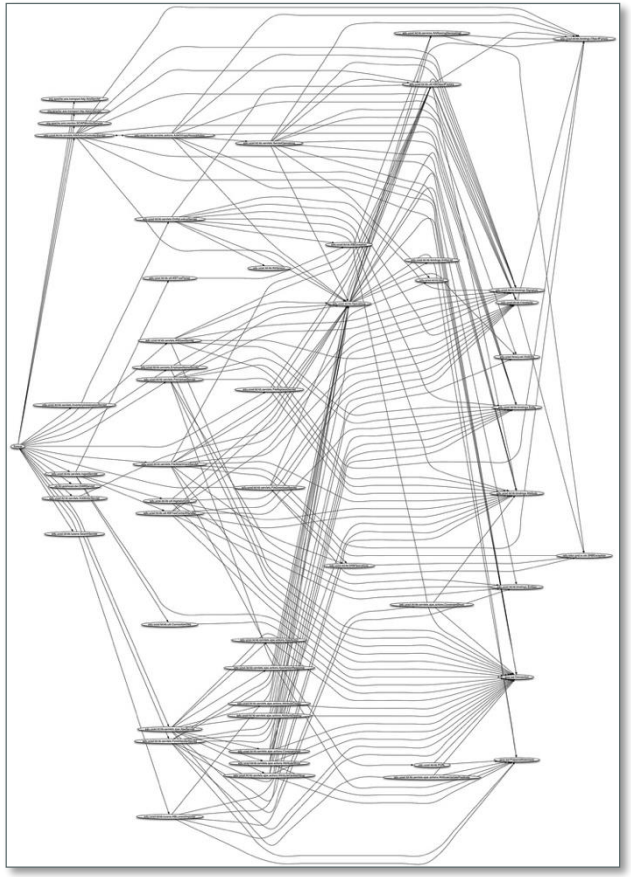
Download class materials from university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Locate dependencies using the Factory Pattern
2. Use a Service Locator to register and retrieve dependencies
3. Use an IoC container to automatically inject dependencies



Locate dependencies using the Factory Pattern

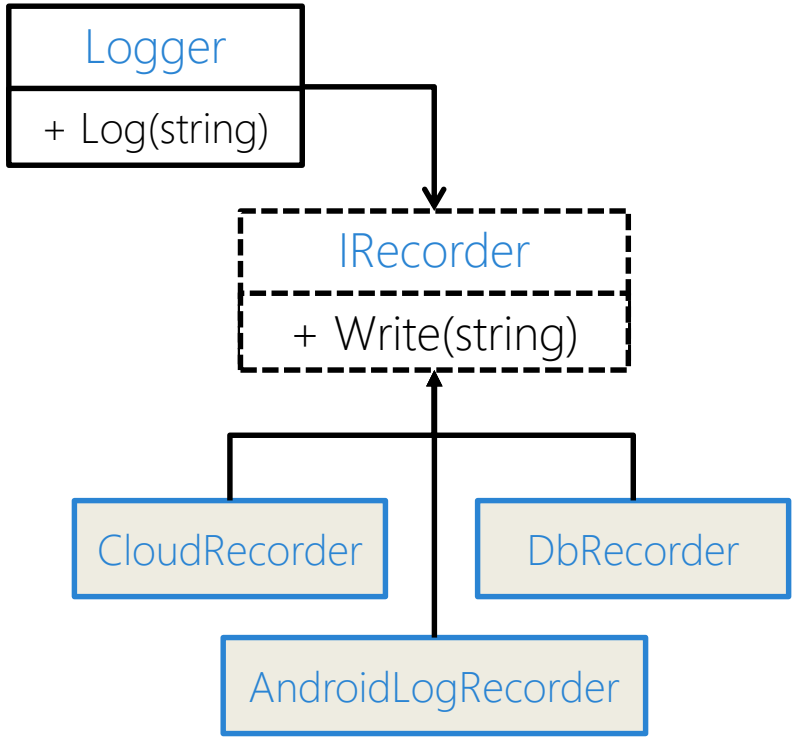
Xamarin University

Tasks

1. Define a Factory
2. Assign a dependency to a Factory
3. Access the Factory from shared code

Using Platform Features

- Common problem to require APIs which are platform-specific
  - alerts / notifications
  - file I/O
  - UI marshaling
  - …
- Use Bridge Pattern to decouple implementation; this also enables testing



Example: Alert Service

- For example – every platform has a unique way to notify the user that something has occurred
- Shared code will use the `IAlertService` *abstraction*
- Platform(s) must each *implement abstraction* using their own unique API

```
public interface IAlertService
{
    bool Show(string title,
              string message,
              string yesButton,
              string noButton);
}
```

Using Services from our Shared Code

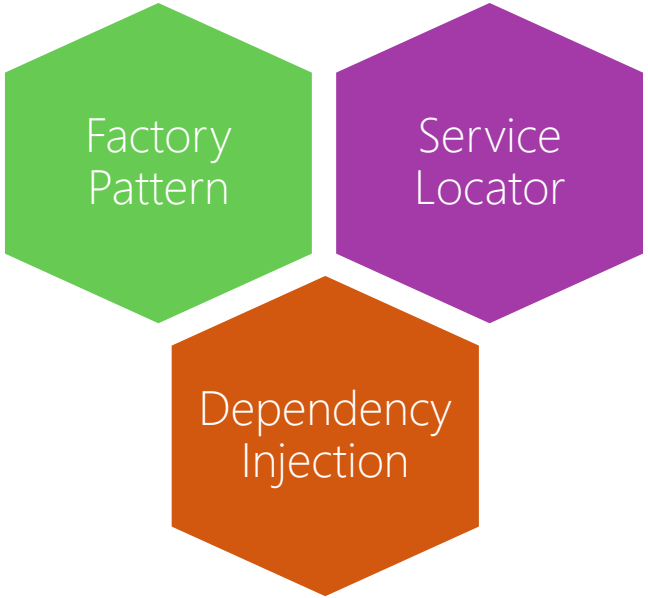
- Once we have abstractions and implementations we need to tie them together

Need to provide the **IAlertService** to the class or method

```
public class TerminatorViewModel
{
    ...
    public void TerminateJohnConner()
    {
        IAlertService alert = ??;
        if (!alert.Show("John Conner Located!",
             "Initiate termination sequence?",
             "Yes", "No")) { ... }
    }
}
```

Locating Services – Inversion of Control

- Several well-known patterns can be used to break dependencies and loosely-couple components together
  - referred to as "Inversion of Control" (IoC)
  - allow reusable components to call into platform-specific code (vs. the other way around)



Factory Pattern

- Dependencies can be located through factories which are responsible for creating the abstractions

Defining a Factory

Delegate is set by platform which returns implementation of the defined **AlertService**

```
public abstract class AlertService
{
    public static Func<AlertService> Create { get; set; }

    public abstract bool Show(string title,
                              string message,
                              string yesButton,
                              string noButton);
}
```

Note: this is just one way to build a Factory, as with any pattern, the implementation can be tailored to the language and platform capabilities

Setting up a Factory

- Each platform would implement the abstraction and then set the factory property to a delegate that returns the implementation

```
class AlertServiceiOS : AlertService
{
    public bool Show(string title, string message,
                     string yesButton, string noButton) { ... }
}
```

```
public override bool FinishedLaunching(...) {
    ...
    AlertService.Create = () => new AlertServiceiOS();
}
```

iOS

Using a Factory

- Then any code in the project that needed that feature would go to the known factory to create the object to be used

```
public void OnReceivedError(string errorMessage)
{
   var alertService = AlertService.Create();
   alertService.Show ("Error",
      $"Got error: {errorMessage}", "OK", null);
   ...
}
```

Now the client doesn't need to know or care about the implementation – it goes to the factory to get one and just uses it from anywhere in the app

Individual Exercise

Use the Factory Pattern to access a dependency from shared code

Xamarin University

Factory Pros and Cons

Pros	Cons
▪ Hides the implementation	▪ Requires separate “factory” for each abstraction (possible maintenance issue)
▪ Easy to use and understand	▪ Client must take dependency against factory
▪ Can decide implementation at runtime and return specific version based on environment	▪ Missing dependencies are not known until runtime

Use a Service Locator to register and retrieve dependencies

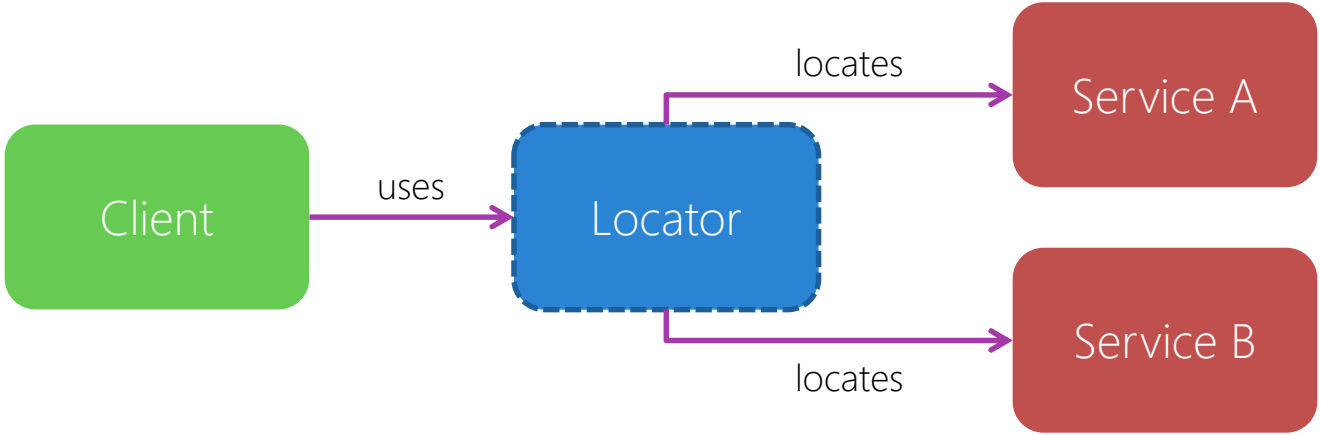
Xamarin University

Tasks

1. Define a Service Locator
2. Register dependencies with a Service Locator
3. Resolve dependencies from a Service Locator

Service Locator

- Service Locator pattern uses a container that maps abstractions (interfaces) to concrete, registered types – client then uses locator to find dependencies



Service Locator Example Definition

Uses Singleton pattern to provide global accessibility

```
public sealed class ServiceLocator
{
    public static ServiceLocator Instance { get; set; }

    public void Add(Type contractType, object value);
    public void Add(Type contractType, Type serviceType);
    public object Resolve(Type contractType);
    public T Resolve<T>();
}
```

Provide capability to register and locate types

Registering Dependencies

```
public partial class AppDelegate
{
    ...
    public override void FinishedLaunching(UIApplication application)
    {
        ...
        ServiceLocator.Instance.Add<IAlertService,MyAlertService>();
    }
}
```

Platform-specific code *registers* implementation for the abstraction

Using the Service Locator

```
public void DialNumber(string number)
{
   var alert = ServiceLocator.Instance.Resolve<IAlertService>();
   if (!alert.Show("Dial Number",
        "Are you sure you want to dial " + number,
        "Yes", "No")) { ... }
}
```

Client then *requests* the abstraction and locator returns the registered implementation

Service Locator implementations

- Easy to create your own service locator, but there are many usable 3rd-party implementations including:
  - Common Service Locator [`commonservicelocator.codeplex.com`]
  - Most Mvvm/Pattern libraries have a Service Locator
  - Xamarin.Forms `DependencyService`

Service Locator Pros and Cons

Pros	Cons
▪ Easy to use and understand ▪ Clients can JIT-request services ▪ Can be used with any client	▪ Clients must all have access to Locator ▪ Harder to identify dependencies in code ▪ Missing dependencies harder to detect

Group Exercise

Build a Service Locator

Xamarin University

Use an IoC container to automatically inject dependencies

Tasks

1. Register dependencies with an IoC container
2. Inject dependencies
3. Automate dependence injection

Dependency Injection

- Another option is to have the platform-specific code "inject" the dependency by passing it as a parameter or setting a property

```
public class DataAccessLayer
{
   public DataAccessLayer(
            IDbRepository db,
            IAlertService alerts) { ... }

   public ILogger Logger { get; set; }
   ...
}
```

Services this class depends on must be supplied ("injected") through constructor parameters, properties or method parameters

Using Dependency Injection

- Can then connect the client and required dependencies together manually in our code

```
public DataAccessLayer CreateDataLayer()
{
   var dataAccessLayer = new DataAccessLayer(
      new SqliteRepository(),                    // IDbRepository
      new WinRTAlertService());                  // IAlertService
   dataAccessLayer.Logger = new AzureLogger(); // ILogger

   return dataAccessLayer
}
```

Inversion of Control (IoC) container

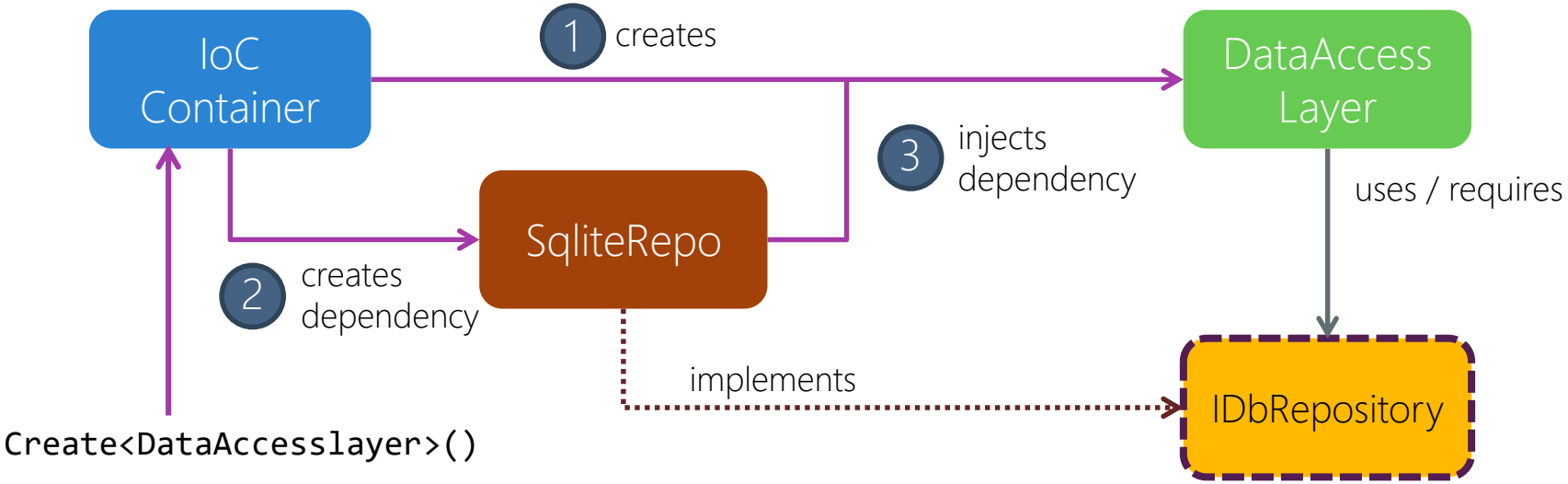
- An IoC container is a **dependency manager** used to create and control the lifetime of dependencies in your application; it has two purposes:

Registry of known dependencies

Creates objects and "injects" required dependencies

DI with an IoC Container

- Can automate DI with a *container* that dependencies are registered with which then *create* types – automatically supplying the dependencies



DI Container Example

Dependencies are typically registered in platform-specific code (but don't have to be!)

```
MyContainer container = new MyContainer();
container.Register<IDbRepository,SqliteRepository>();
container.Register<IAlertService,WinRTAlertService>();
container.Register<ILogger>(new AzureLogger(AzureToken));
container.Register< MessageBus>(new MessageBus(this));
```

```
var dataLayer = container.Create<DataAccessLayer>();
...
```

Can then ask container to *create* the **`DataAccessLayer`** from anywhere in our code – it will automatically supply the required dependencies

DI + Containers Pros and Cons

Pros	Cons
▪ Client only needs **real dependencies**, no container reference necessary	▪ Involves a bit of magic (!), the big picture can be harder to understand (what depends on what)
▪ Easier to identify dependencies being used since they are often passed to constructors or filled in properties	▪ Often requires some form of reflection; not generally a performance issue but could be

DI / IoC Containers

- Many popular 3rd-party IoC containers available:
  - TinyIoC
  - Ninject
  - AutoFac
  - Unity
  - MvvmCross
  - …

Individual Exercise

Use Dependency Injection

Xamarin University

Flash Quiz

Xamarin University

Flash Quiz

① Key to all these patterns is _____.

a) Custom attributes

b) Containers

c) Singletons

d) Abstractions

Flash Quiz

① Key to all these patterns is _____.

a) Custom attributes

b) Containers

c) Singletons

d) Abstractions

Flash Quiz

② Service Locator is where _____.

a) Services are found and set into properties on the client
b) Client request specific abstraction through a shared locator
c) Client creates service directly
d) You use *Accio* summoning charm to create the service

Flash Quiz

② Service Locator is where _____.

a) Services are found and set into properties on the client

b) Client request specific abstraction through a shared locator

c) Client creates service directly

d) You use *Accio* summoning charm to create the service

Flash Quiz

③ To inject dependencies the IoC container will often need to create the dependencies as well as the type that uses those dependencies

a) True

b) False

Flash Quiz

③ To inject dependencies the IoC container will often need to create the dependencies as well as the type that uses those dependencies

a) True

b) False

Flash Quiz

④ The best technique to manage dependencies is _____.

a) Factory Pattern
b) Service Locator Pattern
c) Dependency Injection
d) Depends on the project, team, and personal preference.

Flash Quiz

④ The best technique to manage dependencies is _____.

a) Factory Pattern

b) Service Locator Pattern

c) Dependency Injection

d) It depends on the project, team, and personal preference.

Summary

1. Register dependencies with an IoC container
2. Inject dependencies
3. Automate dependence injection

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft