Microsoft

Dependency Injection with Unity

Dominic Betts
Grigori Melnik
Fernando Simonazzi
Mani Subramanian

Foreword by Chris Tavares

patterns & practices

Contents

Foreword

The set of ideas that later congealed into the Unity container were originally conceived while I was working on the Web Client Software Factory project. Microsoft's patterns & practices team had been using the concept of dependency injection for several years at that point, most famously in the Composite Application Block (CAB). It was also core to the configuration story in Enterprise Library 2.0, and was again central when we started tackling composite applications for the web (a library that became known as CWAB).

Our goal had always been to promote the concepts of Dependency Injection as a way to build loosely coupled systems. However, the way p&p approached DI at the time was different then how we think about it now. Instead of a single reusable container it was felt that the DI implementation should be specialized to the system in which it was being used. We used a library called ObjectBuilder, which was described as "a framework to build DI containers." This would in theory let us write a container per project that did exactly what we wanted.

A lofty aspiration, but in practice it didn't work out so well. ObjectBuilder was a highly decoupled, abstract set of parts that had to be manually assembled. Combined with a lack of documentation it took a lot of time to understand what needed to go where and how to put it together into something useful. That turned into time spent writing, debugging, and optimizing the DI container instead of working on our actual project requirements.

It got even more fun when somebody wanted to use CAB (which used one DI container based on one version of ObjectBuilder) and Enterprise Library (with a separate container based on a different version of ObjectBuilder) in the same project. Integration was very difficult; just dealing with referencing two different versions of ObjectBuilder in the same project was a challenge. Also the one-off containers led to one-off extensibility and integration interfaces: what worked in Enterprise Library was useless in CAB and vice versa.

It finally came to a head when we'd just spent yet another week near the end of the Web Client Software Factory project fixing a bunch of bugs in CWAB: bugs that looked very similar to ones we'd fixed before in CAB. Wouldn't it be nice, we asked, if we could just have one container implementation and just use it instead of writing them over and over again?

From this frustration grew Unity. The Enterprise Library 4.0 team put the Dependency Injection Application Block (as Unity used to be known originally) on the product backlog. Our goals for the project were straightforward. First, introduce and promote the concepts of dependency injection to our community, unencumbered by a lot of low-level implementation details. Second, have a core container with an easy to use API that we, other teams at Microsoft, or anyone whose organization was uncomfortable using the available open source projects (for whatever reason) could just use. Third, have a variety of extensibility mechanisms so that new features could be added by anyone without having to rip open the core code.

In my opinion Unity has succeeded on all these goals. I'm particularly proud of how we affected the .NET developer community. Unity quickly became one of the most commonly used DI containers in the .NET ecosystem. More importantly, other DI container usage has increased as well. Unity introduced DI to a new set of people who would have otherwise never heard of it. Some of them later moved on to other containers that better suited their needs. That's not a loss for Unity: they're using the concepts, and that's the important part.

There's not a whole lot of evangelism published for DI containers anymore. In my opinion, this is because DI is no longer an "expert technique": it's now part of the mainstream. When even frameworks from Microsoft (ASP.NET MVC and WebAPI in particular) come with support for DI built in, you know that a concept has reached the core audience. I think Unity had a very large role in making this happen.

I'm thrilled to see this book published. For the first time, there's one place you can look for both the concepts of DI and how to apply those concepts using the Unity container. And there's coverage of the extensibility story, something I always wanted to write but never seemed to get started. I don't need to feel guilty about that anymore!

Read the book, embrace the concepts, and enjoy the world of loosely coupled, highly cohesive software that DI makes so easy to build!

Chris Tavares
Redmond, WA, USA
April 2013

Preface

About This Guide

This guide is one of the resources available with the Unity v3 release to help you to learn about Unity, learn about some of the problems and issues that Unity can help you to address, and get started using Unity in your applications. Unity is primarily a *dependency injection* container and so the guide also contains an introduction to dependency injection that you can read in isolation even if you don't plan to use Unity, although we hope you will.

The chapters are designed to be read in order, each one building on the previous one, and alternating chapters that cover some of the conceptual background material with chapters that address the specifics of using Unity in your own applications. If you're already familiar with concepts such as dependency injection and *interception*, you can probably focus on Chapter 3, "Dependency Injection with Unity," Chapter 5, "Interception with Unity," and Chapter 6, "Extending Unity."

The first two chapters introduce the conceptual background and explain what dependency injection is, what are its benefits and drawbacks, and when you should consider using it. Chapter 3 then applies this theoretical knowledge to the use of the Unity container and provides examples and guidance on how to use it in a variety of scenarios.

Chapter 4 then describes how you can implement interception using the Unity container.

Chapter 5 introduces the concepts associated with injection, again explains its benefits and drawbacks, discusses some alternatives, and offers some suggestions about when you should use it.

The final chapter introduces some of the ways that you can extend Unity such as creating container extensions or creating custom lifetime managers.

All of the chapters include references to additional resources such as books, blog posts, and papers that will provide additional detail if you want to explore some of the topics in greater depth. The majority of the code samples in the chapters come from a collection of sample applications that you can download and play with.

This guide does not include detailed information about every Unity feature or every class in the Unity assemblies. For that information, you should look at the Unity Reference Documentation and the Unity API Documentation.

Who This Book Is For

This book is intended for any architect, developer, or information technology (IT) professional who designs, builds, or operates applications and services and who wants to learn how to realize the benefits of using the Unity dependency injection container in his or her applications. You should be familiar with the Microsoft .NET Framework, and Microsoft Visual Studio to derive full benefit from reading this guide.

What Do You Need to Get Started?

The system requirements and prerequisites for using Unity are:

- Supported architectures: x86 and x64.
- Operating systems: Microsoft Windows 8, Microsoft Windows 7, Windows Server 2008 R2, Windows Server 2012.
- Supported .NET Frameworks: Microsoft .NET Framework 4.5, .NET for Windows Store Apps (previously known as WinRT).
- Rich development environment: Microsoft Visual Studio 2012, Professional, Ultimate, or Express editions.

You can use the NuGet package manager in Visual Studio to install the Unity assemblies in your projects.

Who's Who

The guide includes discussions and examples that relate to the use of Unity in a variety of scenarios and types of application. A panel of experts provides a commentary throughout the book, offering a range of viewpoints from developers with various levels of skill, an architect, and an IT professional. The following table lists the various experts who appear throughout the guide.

	Markus is a software developer who is new to Unity. He is analytical, detail-oriented, and methodical. He's focused on the task at hand, which is building a great LOB application. He knows that he's the person who's ultimately responsible for the code. "I want to get started using Unity quickly, so I want it to be simple to incorporate into my code and be easy to configure with plenty of sensible defaults."
	Beth is a developer who used Unity some time ago but abandoned it for her more recent projects. "I'm happy using libraries and frameworks but I don't want to get tied into dependencies that I don't need. I want to be able to use just the components I need for the task in hand."
	Jana is a software architect. She plans the overall structure of an application. Her perspective is both practical and strategic. In other words, she considers not only what technical approaches are needed today, but also what direction a company needs to consider for the future. Jana has worked on many projects that have used Unity as well as other dependency injection containers. Jana is comfortable assembling a solution using multiple libraries and frameworks. "It's not easy to balance the needs of the company, the users, the IT organization, the developers, and the technical platforms we rely on while trying to ensure component independence."
	Carlos is an experienced software developer and Unity expert. As a true professional, he is well aware of the common crosscutting concerns that developers face when building line-of-business (LOB) applications for the enterprise. His team is used to relying on Unity and he is happy to see continuity in Unity releases. Quality, support, and ease of migration are his primary concerns. "Our existing LOB applications use Unity for dependency and interception. This provides a level of uniformity across all our systems that make them easier to support and maintain. We want to be able to migrate our existing applications to the new version with a minimum of effort."

	Poe is an IT professional who's an expert in deploying and managing LOB applications. Poe has a keen interest in practical solutions; after all, he's the one who gets paged at 3:00 AM when there's a problem. Poe wants to be able to tweak application configuration without recompiling or even redeploying them in order to troubleshoot. "I want a consistent approach to configuration for all our applications both on-premises and in the cloud."

Chapter 1 - Introduction

Before you learn about dependency injection and Unity, you need to understand why you should use them. And in order to understand why you should use them, you should understand what types of problems dependency injection and Unity are designed to help you address. This introductory chapter will not say much about Unity, or indeed say much about dependency injection, but it will provide some necessary background information that will help you to appreciate the benefits of dependency injection as a technique and why Unity does things the way it does.

The next chapter, Chapter 2, "Dependency Injection," will show you how dependency injection can help you meet the requirements outlined in this chapter, and the following chapter, Chapter 3, "Dependency Injection with Unity," shows how Unity helps you to implement the dependency injection approach in your applications.


This chapter introduces a lot of requirements and principles. Don’t assume that they are all relevant all of the time. However, most enterprise systems have some of the requirements, and the principles all point towards good design and coding practices.

Motivations

When you design and develop software systems, there are many requirements to take into account. Some will be specific to the system in question and some will be more general in purpose. You can categorize some requirements as functional requirements, and some as non-functional requirements (or quality attributes). The full set of requirements will vary for every different system. The set of requirements outlined below are common requirements, especially for line-of-business (LOB) software systems with relatively long anticipated lifetimes. They are not all necessarily going to be important for every system you develop, but you can be sure that some of them will be on the list of requirements for many of the projects you work on.

Maintainability

As systems become larger, and as the expected lifetimes of systems get longer, maintaining those systems becomes more and more of a challenge. Very often, the original team members who developed the system are no longer available, or no longer remember the details of the system. Documentation may be out of date or even lost. At the same time, the business may be demanding swift action to meet some pressing new business need. *Maintainability* is the quality of a software system that determines how easily and how efficiently you can update it. You may need to update a system if a defect is discovered that must be fixed (in other words, performing *corrective maintenance*), if some change in the operating environment requires you to make a change in the system, or if you need to add new features to the system to meet a business requirement (*perfective maintenance*). Maintainable systems enhance the agility of the organization and reduce costs.


It is very hard to make existing systems more maintainable. It is much better to design for maintainability from the very start.

Therefore, you should include maintainability as one of your design goals, along with others such as reliability, security, and scalability.

Testability

A *testable* system is one that enables you to effectively test individual parts of the system. Designing and writing effective tests can be just as challenging as designing and writing testable application code, especially as systems become larger and more complex. Methodologies such as test-driven development (TDD) require you to write a unit test before writing any code to implement a new feature and the goal of such a design technique is to improve the quality of your application. Such design techniques also help to extend the coverage of your unit tests, reduce the likelihood of regressions, and make refactoring easier. However, as part of your testing processes you should also incorporate other types of tests such as acceptance tests, integration tests, performance tests, and stress tests.

Running tests can also cost money and be time consuming because of the requirement to test in a realistic environment. For example, for some types of testing on a cloud-based application you need to deploy the application to the cloud environment and run the tests in the cloud. If you use TDD, it may be impractical to run all the tests in the cloud all of the time because of the time it takes to deploy your application, even to a local emulator. In this type of scenario, you may decide to use *test doubles* (simple *stubs* or verifiable *mocks*) that replace the real components in the cloud environment with test implementations in order to enable you to run your suite of unit tests in isolation during the standard TDD development cycle.


Using test doubles is a great way to ensure that you can continuously run your unit tests during the development process. However, you must still fully test your application in a real environment.


For a great discussion on the use of test doubles, see the point/counterpoint debate by Steve Freeman, Nat Pryce and Joshua Kerievsky in IEEE Software (Volume: 24, Issue: 3), May/June 2007, pp.80-83.

Testability should be another of the design goals for your system along with maintainability and agility: a testable system is typically more maintainable, and vice versa.

Flexibility and Extensibility

Flexibility and *extensibility* are also often on the list of desirable attributes of enterprise applications. Given that business requirements often change, both during the development of an application and after it is running in production, you should try to design the application to make it flexible so that it can be adapted to work in different ways and extensible so that you can add new features. For example, you may need to convert your application from running on-premises to running in the cloud.

Late Binding

In some application scenarios, you may have a requirement to support *late binding*. Late binding is useful if you require the ability to replace part of your system without recompiling. For example, your application might support multiple relational databases with a separate module for each supported database type. You can use declarative configuration to tell the application to use a specific module at runtime. Another scenario where late binding can be useful is to enable users of the system to provide their own customization through a plug-in. Again, you can instruct the system to use a specific customization by using a configuration setting or a convention where the system scans a particular location on the file system for modules to use.


Not all systems have a requirement for late binding. It is typically required to support a specific feature of the application such as customization using a plug-in architecture.

Parallel Development

When you are developing large scale (or even small and medium scale) systems, it is not practical to have the entire development team working simultaneously on the same feature or component. In reality, you will assign different features and components to smaller groups to work on in parallel. Although this approach enables you to reduce the overall duration of the project, it does introduce additional complexities: you need to manage multiple groups and to ensure that you can integrate the parts of the application developed by different groups to work correctly together.


It can be a significant challenge to ensure that classes and components developed independently do work together.

Crosscutting Concerns

Enterprise applications typically need to address a range of *crosscutting concerns* such as validation, exception handling, and logging. You may need these features in many different areas of the application and you will want to implement them in a standard, consistent way to improve the maintainability of the system. Ideally, you want a mechanism that will enable you to efficiently and transparently add behaviors to your objects at either design time or run time without requiring you make changes to your

existing classes. Often, you need the ability to configure these features at runtime and in some cases, add features to address a new crosscutting concern to an existing application.

 For a large enterprise system, it's important to be able to manage crosscutting concerns such as logging and validation in a consistent manner. I often need to change the logging level on a specific component at run time to troubleshoot an issue without restarting the system.

Loose Coupling

You can address many of the requirements listed in the previous sections by ensuring that your design results in an application that loosely couples the many parts that make up the application. *Loose coupling*, as opposed to *tight coupling*, means reducing the number of dependencies between the components that make up your system. This makes it easier and safer to make changes in one area of the system because each part of the system is largely independent of the other.

 Loose coupling should be a general design goal for your enterprise applications.

A Simple Example

The following example illustrates tight coupling where the **ManagementController** class depends directly on the **TenantStore** class. These classes might be in different Visual Studio projects.

C#

```
public class TenantStore
{
  ...

  public Tenant GetTenant(string tenant)
  {
    ...
  }

  public IEnumerable<string> GetTenantNames()
  {
    ...
  }
}

public class ManagementController
{
  private readonly TenantStore tenantStore;

  public ManagementController()
```

```
  {
    tenantStore = new TenantStore(...);
  }

  public ActionResult Index()
  {
    var model = new TenantPageViewData<IEnumerable<string>>
      (this.tenantStore.GetTenantNames())
    {
      Title = "Subscribers"
    };
    return this.View(model);
  }


  public ActionResult Detail(string tenant)
  {
    var contentModel = this.tenantStore.GetTenant(tenant);
    var model = new TenantPageViewData<Tenant>(contentModel)
    {
      Title = string.Format("{0} details", contentModel.Name)
    };
    return this.View(model);
  }

  ...
}
```

The **ManagementController** and **TenantStore** classes are used in various forms throughout this guide. Although the **ManagementController** class is an ASP.NET MVC controller, you don't need to know about MVC to follow along. However, these examples are intended to look like the kinds of classes you would encounter in a real-world system, especially the examples in Chapter 3.

In this example, the **TenantStore** class implements a repository that handles access to an underlying data store such as a relational database, and the **ManagementController** is an MVC controller class that requests data from the repository. Note that the **ManagementController** class must either instantiate a **TenantStore** object or obtain a reference to a **TenantStore** object from somewhere else before it can invoke the **GetTenant** and **GetTenantNames** methods. The **ManagementController** class depends on the specific, concrete **TenantStore** class.

If you refer back to the list of common desirable requirements for enterprise applications at the start of this chapter, you can evaluate how well the approach outlined in the previous code sample helps you to meet them.

- Although this simple example shows only a single client class of the **TenantStore** class, in practice there may be many client classes in your application that use the **TenantStore** class. If you assume that each client class is responsible for instantiating or locating a **TenantStore**

object at runtime, then all of those classes are tied to a particular constructor or initialization method in that **TenantStore** class, and may all need to be changed if the implementation of the **TenantStore** class changes. This potentially makes maintenance of the **TenantStore** class more complex, more error prone, and more time consuming.

- In order to run unit tests on the **Index** and **Detail** methods in the **ManagementController** class, you need to instantiate a **TenantStore** object and make sure that the underlying data store contains the appropriate test data for the test. This complicates the testing process, and depending on the data store you are using, may make running the test more time consuming because you must create and populate the data store with the correct data. It also makes the tests much more brittle.
- It is possible to change the implementation of the **TenantStore** class to use a different data store, for example Windows Azure table storage instead of SQL Server. However, it might require some changes to the client classes that use **TenantStore** instances if it was necessary for them to provide some initialization data such as connection strings.
- You cannot use late binding with this approach because the client classes are compiled to use the **TenantStore** class directly.
- If you need to add support for a crosscutting concern such as logging to multiple store classes, including the **TenantStore** class, you would need to modify and configure each of your store classes independently.

The following code sample shows a small change, the constructor in the client **ManagementController** class now receives an object that implements the **ITenantStore** interface and the **TenantStore** class provides an implementation of the same interface.

C#

```
public interface ITenantStore
{
  void Initialize();
  Tenant GetTenant(string tenant);
  IEnumerable<string> GetTenantNames();
  void SaveTenant(Tenant tenant);
  void UploadLogo(string tenant, byte[] logo);
}

public class TenantStore : ITenantStore
{
  ...

  public TenantStore()
  {
    ...
  }
```

```
  ...
}

public class ManagementController : Controller
{
  private readonly ITenantStore tenantStore;

  public ManagementController(ITenantStore tenantStore)
  {
    this.tenantStore = tenantStore;
  }

  public ActionResult Index()
  {
    ...
  }

  public ActionResult Detail(string tenant)
  {
    ...
  }

  ...
}
```

This change has a direct impact on how easily you can meet the list of requirements.

- It is now clear that the **ManagementController** class, and any other clients of the **TenantStore** class are no longer responsible for instantiating **TenantStore** objects, although the example code shown doesn't show which class or component is responsible for instantiating them. From the perspective of maintenance, this responsibility could now belong to a single class rather than many.
- It’s now also clear what dependencies the controller has from its constructor arguments instead of being buried inside of the controller method implementations.
- To test some behaviors of a client class such as the **ManagementController** class, you can now provide a lightweight implementation of the **ITenantStore** interface that returns some sample data. This is instead of creating a **TenantStore** object that queries the underlying data store for sample data.
- Introducing the **ITenantStore** interface makes it easier to replace the store implementation without requiring changes in the client classes because all they expect is an object that implements the interface. If the interface is in a separate project to the implementation, then

the projects that contain the client classes only need to hold a reference to the project that contains the interface definition.

- It is now also possible that the class responsible for instantiating the store classes could provide additional services to the application. It could control the lifetime of the **ITenantStore** instances that it creates, for example creating a new object every time the client **ManagementController** class needs an instance, or maintaining a single instance that it passes as a reference whenever a client class needs it.
- It is now possible to use late binding because the client classes only reference the **ITenantStore** interface type. The application can create an object that implements the interface at runtime, perhaps based on a configuration setting, and pass that object to the client classes. For example, the application might create either a **SQLTenantStore** instance or a **BlobTenantStore** instance depending on a setting in the web.config file, and pass that to the constructor in the **ManagementController** class.
- If the interface definition is agreed, two teams could work in parallel on the store class and the controller class.
- The class that is responsible for creating the store class instances could now add support for the crosscutting concerns before passing the store instance on to the clients, such as by using the decorator pattern to pass in an object that implements the crosscutting concerns. You don't need to change either the client classes or the store class to add support for crosscutting concerns such as logging or exception handling.

The approach shown in the second code sample is an example of a loosely coupled design that uses interfaces. If we can remove a direct dependency between classes, it reduces the level of coupling and helps to increase the maintainability, testability, flexibility, and extensibility of the solution.


Loose coupling doesn’t necessarily imply dependency injection, although the two often do go together.

What the second code sample doesn't show is how dependency injection and the Unity container fit into the picture, although you can probably guess that they will be responsible for creating instances and passing them to client classes. Chapter 2 describes the role of dependency injection as a technique to support loosely coupled designs, and Chapter 3 describes how Unity helps you to implement dependency injection in your applications.

When Should You Use a Loosely Coupled Design?

Before we move on to dependency injection and Unity, you should start to understand where in your application you should consider introducing loose coupling, programming to interfaces, and reducing dependencies between classes. The first requirement we described in the previous section was

maintainability, and this often gives a good indication of when and where to consider reducing the coupling in the application. Typically, the larger and more complex the application, the more difficult it becomes to maintain, and so the more likely these techniques will be helpful. This is true regardless of the type of application: it could be a desktop application, a web application, or a cloud application.

At first sight, this perhaps seems counterintuitive. The second example shown above introduced an interface that wasn't in the first example, it also requires the bits we haven't shown yet that are responsible for instantiating and managing objects on behalf of the client classes. With a small example, these techniques appear to add to the complexity of the solution, but as the application becomes larger and more complex, this overhead becomes less and less significant.

Small examples of loosely coupled design, programming to interfaces, and dependency injection often appear to complicate the solution. You should remember that these techniques are intended to help you simplify and manage large and complex applications with many classes and dependencies. Of course small applications can often grow into large and complex applications.

The previous example also illustrates another general point about where it is appropriate to use these techniques. Most likely, the **ManagementController** class exists in the user interface layer in the application, and the **TenantStore** class is part of the data access layer. It is a common approach to design an application so that in the future it is possible to replace one tier without disturbing the others. For example, replacing or adding a new UI to the application (such as creating an app for a mobile platform in addition to a traditional web UI) without changing the data tier or replacing the underlying storage mechanism and without changing the UI tier. Building the application using tiers helps to decouple parts of the application from each other. You should try to identify the parts of an application that are likely to change in the future and then decouple them from the rest of the application in order to minimize and localize the impact of those changes.

The list of requirements in the previous section also includes crosscutting concerns that you might need to apply across a range of classes in your application in a consistent manner. Examples include the concerns addressed by the application blocks in Enterprise Library (http://msdn.microsoft.com/entlib) such as logging, exception handling, validation, and transient fault handling. Here you need to identify those classes where you might need to address these crosscutting concerns, so that responsibility for adding these features to these classes resides outside of the classes themselves. This helps you to manage these features consistently in the application and introduces a clear separation of concerns.

Principles of Object-Oriented Design

Finally, before moving on to dependency injection and Unity, we want to relate the five SOLID principles of object-oriented programming and design to the discussion so far. SOLID is an acronym that refers to the following principles:

- **S**ingle responsibility principle
- **O**pen/close principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

The following sections describe each of these principles and their relationship to loose coupling and the requirements listed at the start of this chapter.

Single Responsibility Principle

The single responsibility principle states that a class should have one, and only one, reason to change. For more information, see the article Principles of Object Oriented Design by Robert C. Martin[1].

In the first simple example shown in this chapter, the **ManagementController** class had two responsibilities: to act as a controller in the UI and to instantiate and manage the lifetime of **TenantStore** objects. In the second example, the responsibility for instantiating and managing **TenantStore** objects lies with another class or component in the system.

The Open/Closed Principle

The open/closed principle states that "software entities (classes, modules, functions, and so on) should be open for extension, but closed for modification" (Meyer, Bertrand (1988). Object-Oriented Software Construction.)

Although you might modify the code in a class to fix a defect, you should extend a class if you want to add any new behavior to it. This helps to keep the code maintainable and testable because existing behavior should not change, and any new behavior exists in new classes. The requirement to be able to add support for crosscutting concerns to your application can best be met by following the open/closed principle. For example, when you add logging to a set of classes in your application, you shouldn't make changes to the implementation of your existing classes.

[1] The Principles of OOD, Robert C Martin, http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

The Liskov Substitution Principle

The Liskov substitution principle in object-oriented programming states that in a computer program, if **S** is a subtype of **T**, then objects of type **T** may be replaced with objects of type **S** without altering any of the desirable properties, such as correctness, of that program.

In the second code sample shown in this chapter, the **ManagementController** class should continue to work as expected if you pass any implementation of the **ITenantStore** interface to it. This example uses an interface type as the type to pass to the constructor of the **ManagementController** class, but you could equally well use an abstract type.

Interface Segregation Principle

The interface segregation principle is a software development principle intended to make software more maintainable. The interface segregation principle encourages loose coupling and therefore makes a system easier to refactor, change, and redeploy. The principle states that interfaces that are very large should be split into smaller and more specific ones so that client classes only need to know about the methods that they use: no client class should be forced to depend on methods it does not use.

In the definition of the **ITenantStore** interface shown earlier in this chapter, if you determined that not all client classes use the **UploadLogo** method you should consider splitting this into a separate interface as shown in the following code sample:

C#

```
public interface ITenantStore
{
  void Initialize();
  Tenant GetTenant(string tenant);
  IEnumerable<string> GetTenantNames();
  void SaveTenant(Tenant tenant);
}

public interface ITenantStoreLogo
{
  void UploadLogo(string tenant, byte[] logo);
}


public class TenantStore : ITenantStore, ITenantStoreLogo
{
  ...

  public TenantStore()
  {
    ...
  }

  ...
```

```
}
```

Dependency Inversion Principle

The dependency inversion principle states that:

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

The two code samples in this chapter illustrate how to apply this principle. In the first sample, the high-level **ManagementController** class depends on the low-level **TenantStore** class. This typically limits the options for re-using the high-level class in another context.

In the second code sample, the **ManagementController** class now has a dependency on the **ITenantStore** abstraction, as does the **TenantStore** class.

Summary

In this chapter, you have seen how you can address some of the common requirements in enterprise applications such as maintainability and testability by adopting a loosely coupled design for your application. You saw a very simple illustration of this in the code samples that show two different ways that you can implement the dependency between the **ManagementController** and **TenantStore** classes. You also saw how the SOLID principles of object-oriented programming relate to the same concerns.

However, the discussion in this chapter left open the question of how to instantiate and manage **TenantStore** objects if the **ManagementController** is no longer responsible for this task. The next chapter will show how dependency injection relates to this specific question and how adopting a dependency injection approach can help you meet the requirements and adhere to the principles outlined in this chapter.

Chapter 2 - Dependency Injection

Introduction

Chapter 1 outlines how you can address some of the most common requirements in enterprise applications by adopting a loosely coupled design to minimize the dependencies between the different parts of your application. However, if a class does not directly instantiate the other objects that it needs, some other class or component must take on this responsibility. In this chapter, you'll see some alternative patterns that you can use to manage how objects are instantiated in your application before focusing specifically on dependency injection as the mechanism to use in enterprise applications.

Factories, Service Locators, and Dependency Injection

Factories, service locators, and dependency injection are all approaches you can take to move the responsibility for instantiating and managing objects on behalf of other client objects. In this section, you'll see how you can use them with the same example you saw in the previous chapter. You'll also see the pros and cons of the different approaches and see why dependency injection can be particularly useful in enterprise applications.

Factory Patterns

There are three common factory patterns. The Factory Method and Abstract Factory patterns from "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley Professional, 1994., and the Simple Factory pattern.

The Factory Method Pattern

The following code samples show how you could apply the factory method pattern to the example shown in the previous chapter. The first code sample shows how you could use a factory method to return an instance of the **TenantStore** class to use in the **ManagementController** class. In this example, the **CreateTenantStore** method is the factory method that creates the **TenantStore** instance and the **Index** method uses this instance as part of its logic.

C#

```
public class ManagementController: Controller
{
  protected ITenantStore tenantStore;

  public ManagementController()
  {
    this.tenantStore = CreateTenantStore();
  }

  protected virtual ITenantStore CreateTenantStore()
  {
    var storageAccount = AppConfiguration
```

```
      .GetStorageAccount("DataConnectionString");
    var tenantBlobContainer = new EntitiesContainer<Tenant>
      (storageAccount, "Tenants");
    var logosBlobContainer = new FilesContainer
      (storageAccount, "Logos",
      "image/jpeg");
    return new TenantStore(tenantBlobContainer,
                           logosBlobContainer);
  }

  public ActionResult Index()
  {
    var model = new TenantPageViewData<IEnumerable<string>>
      (this.tenantStore.GetTenantNames())
    {
      Title = "Subscribers"
    };
    return this.View(model);
  }

  ...
}
```

Using this approach does not remove the dependencies the **ManagementController** has on the **TenantStore** class, nor the **FilesContainer** and **EntitiesContainer** classes. However, it is now possible to replace the underlying storage mechanism without changing the existing **ManagementController** class as the following code sample shows.

C#

```
public class SQLManagementController : ManagementController
{
  protected override ITenantStore CreateTenantStore()
  {
    var storageAccount = ApplicationConfiguration
      .GetStorageAccount("DataConnectionString");
    var tenantSQLTable = ...
    var logosSQLTable = ....
    return new SQLTenantStore(tenantSQLTable, logosSQLTable);
  }

  ...
}
```

The factory method pattern enables you to modify the behavior of a class without modifying the class itself by using inheritance.

The application can use the **SQLManagementController** class to use a SQL-based store without you needing to make any changes to the original **ManagementController** class. This approach results in a flexible and extensible design and implements the open/closed principle described in the previous chapter. However, it does not result in a maintainable solution because all the client classes that use the **TenantStore** class are still responsible for instantiating **TenantStore** instances correctly and consistently.

It is also still difficult to test the **ManagementController** class because it depends on the **TenantStore** type, which in turn is tied to specific storage types (**FilesContainer** and **EntitiesContainer**). One approach to testing would be to create a **MockManagementController** type that derives from **ManagementController** and that uses a mock storage implementation to return test data: in other words you must create two mock types to manage the testing.

> In this example, there is an additional complication because of the way that ASP.NET MVC locates controllers and views based on a naming convention: you must also update the MVC routes to ensure that MVC uses the new **SQLManagementController** class.

Simple Factory Pattern

While the factory method pattern does not remove the dependencies from the high-level client class, such as the **ManagementController** class, on the low-level class, you can achieve this with the simple factory pattern. In this example, you can see that a new factory class named **TenantStoreFactory** is now responsible for creating the **TenantStore** instance on behalf of the **ManagementController** class.

C#

```
public class ManagementController : Controller
  {
    private readonly ITenantStore tenantStore;

    public ManagementController()
    {
      var tenantStoreFactory = new TenantStoreFactory();
      this.tenantStore = tenantStoreFactory.CreateTenantStore();
    }

    public ActionResult Index()
    {
      var model = new TenantPageViewData<IEnumerable<string>>
        (this.tenantStore.GetTenantNames())
      {
        Title = "Subscribers"
      };
      return this.View(model);
    }

  ...
}
```

The simple factory pattern removes the direct dependency of the **ManagementController** class on a specific store implementation. Instead of including the code needed to build a **TenantStore** instance directly, the controller class now relies on the **TenantStoreFactory** class to create the instance on its behalf.

This approach removes much of the complexity from the high-level **ManagementController** class, although in this example the **ManagementController** class is still responsible for selecting the specific type of tenant store to use. You could easily move this logic into the factory class that could read a configuration setting to determine whether to create a **BlobTenantStore** instance or a **SQLTenantStoreInstance**. Making the factory class responsible for selecting the specific type to create makes it easier to apply a consistent approach throughout the application.

Abstract Factory Pattern

One of the problems that can arise from using the simple factory pattern in a large application is that it can be difficult to maintain consistency. For example, the application may include multiple store classes such as **SurveyStore**, **LogoStore,** and **ReportStore** classes in addition to the **TenantStore** class you've seen in the examples so far. You may have a requirement to use a particular type of storage for all of the stores. Therefore, you could implement a **BlobStoreFactory** abstract factory class that can create multiple blob-based stores, and a **SQLStoreFactory** abstract factory class that can create multiple SQL based stores.

The abstract factory pattern is useful if you have a requirement to create families of related objects in a consistent way.

The abstract factory pattern is described in "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, et al.

Service Locator Pattern

Using a service locator provides another variation to this general approach of using another class to create objects on your behalf. You can think of a service locator as a registry that you can look up an instance of an object or service that another class in your application created and registered with the service locator. The service locator might support querying for objects by a string key or by interface type. Often, in contrast to the factory patterns where the factory creates the object but gives responsibility for managing its lifetime to the client class, the service locator is responsible for managing the lifetime of the object and simply returns a reference to the client. Also, factories are typically responsible for creating instances of specific types or families of types as in the case of the abstract factory pattern, while a service locator may be capable of returning a reference to an object of any type in the application.

> The section "Object Lifetime" later in this chapter discusses object lifetimes in more detail.

Any classes that retrieve object references or service references from the service locator will have a dependency on the service locator itself.

When using a service locator, every class will have a dependency on your service locator. This is not the case with dependency injection.

For a description of the service locator pattern, see the section "Using a Service Locator" in the article "Inversion of Control Containers and the Dependency Injection pattern" by Martin Fowler.

For a discussion of why the service locator may be considered an anti-pattern, see the blog post "Service Locator is an Anti-Pattern" by Mark Seeman.

For a shared interface for service location that application and framework developers can reference, see the Common Service Locator library . The library provides an abstraction over dependency injection containers and service locators. Using the library allows an application to indirectly access the capabilities without relying on hard references.

Dependency Injection

A common feature of the all the factory patterns and the service locator pattern, is that it is still the high-level client object's responsibility to resolve its own dependencies by requesting the specific instances of the types that it needs. They each adopt a pull model of varying degrees of sophistication, assigning various responsibilities to the factory or service locator. The pull model also means that the high-level client class has a dependency on the class that is responsible for creating or locating the object it wants to use. This also means that the dependencies of the high-level client classes are hidden inside of those classes rather specified in a single location, making them harder to test.

Figure 1 shows the dependencies in the simple factory pattern where the factory instantiates a **TenantStore** object on behalf of the **ManagementController** class.

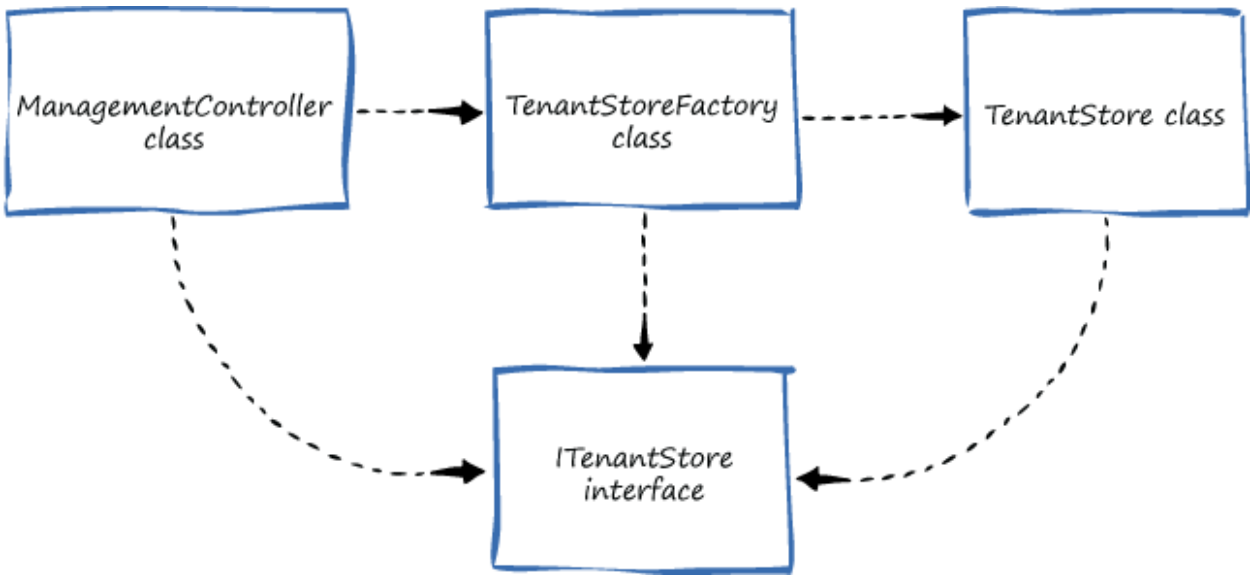

Figure 1 - Dependencies in the factory pattern

Dependency injection takes the opposite approach, adopting a push model in place of the pull model. *Inversion of Control* is a term that's often used to describe this push model and dependency injection is one specific implementation of the inversion of control technique.

Martin Fowler states: "With service locator the application class asks for it explicitly by a message to the locator. With injection there is no explicit request, the service appears in the application class—hence the inversion of control." (Inversion of Control Containers and the Dependency Injection pattern.)

With dependency injection, another class is responsible for injecting (pushing) the dependencies into the high-level client classes, such as the **ManagementController** class, at runtime. The following code sample shows what the high-level **ManagementController** class looks like if you decide to use dependency injection.

C#

```
public class ManagementController : Controller
{
  private readonly ITenantStore tenantStore;

  public ManagementController(ITenantStore tenantStore)
  {
    this.tenantStore = tenantStore;
  }

  public ActionResult Index()
  {
    var model = new TenantPageViewData<IEnumerable<string>>
        (this.tenantStore.GetTenantNames())
    {
      Title = "Subscribers"
```

```
    };
    return this.View(model);
  }

  ...
}
```

As you can see in this sample, the **ManagementController** constructor receives an **ITenantStore** instance as a parameter, injected by some other class. The only dependency in the **ManagementContoller** class is on the interface type. This is better because it doesn't have any knowledge of the class or component that is responsible for instantiating the **ITenantStore** object.

In Figure 2, the class that is responsible for instantiating the **TenantStore** object and inserting it into the **ManagementController** class is called the **DependencyInjectionContainer** class.

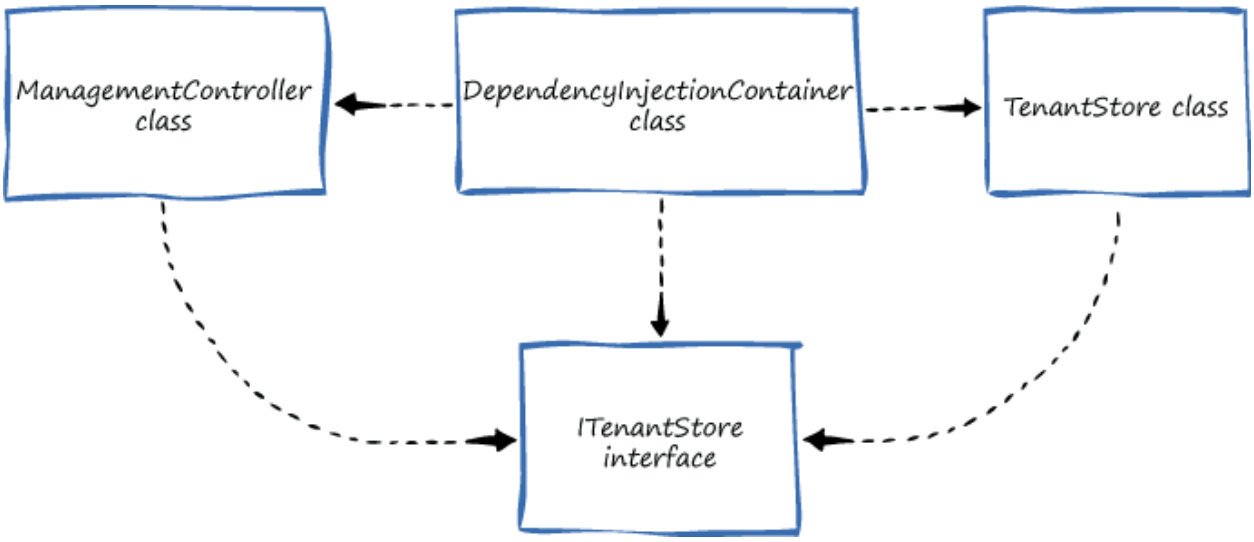

Figure 2 - Dependencies when using dependency injection

> **Note:** Chapter 3, "Dependency Injection with Unity," will describe in more detail what happens in the **DependencyInjectionContainer** class.

The key difference between the Figure 1 and Figure 2 is the direction of the dependency from the **ManagementController** class. In Figure 2, the only dependency the **ManagementController** class has is on the **ITenantStore** interface.

In Figure 2, the **DependencyInjectionContainer** class may manage the dependencies of multiple high level client classes such as the **ManagementController** class on multiple service classes such as the **TenantStore** class.

You can use either a dependency injection container or implement dependency injection manually using factories. As you'll see in the next chapter, using a container is easier and provides additional capabilities such as lifetime management, interception, and registration by convention.

Object Composition

So far in this chapter, you have seen how dependency injection can simplify classes such as the **ManagementController** class and minimize the number of dependencies between classes in your application. The previous chapter explained some of the benefits of this approach, such as maintainability and testability, and showed how this approach relates to the SOLID principles of object-oriented programming. You will now see how this might work in practice: in particular, how and where you might use dependency injection in your own applications.

If you adopt the dependency injection approach, you will have many classes in your application that require some other class or component to pass the necessary dependencies into their constructors or methods as parameters or as property values before you can use them. This implies that your application requires a class or component that is responsible for instantiating all the required objects and passing them into the correct constructors, methods, and properties: your application must know how to compose its object graph before it can perform any work. This must happen very early in the application's lifecycle: for example, in the **Main** method of a console application, in the **Global.asax** in a web application, in a role's **OnStart** method in a Windows Azure application, or in the initialization code for a test method.


Typically, you should place all the code tells the application how to build its object graph in a single location; this is known as the Composition Root pattern. This makes it much easier to maintain and update the application.

Object Lifetime

You should determine when to create the objects in your application based on criteria such as which object is responsible for managing the state, is the object shared, and how long the object will live for. Creating an object always takes a finite amount of time that is determined by the object's size and complexity, and once you have created an object, it occupies some of your system's memory.


Whichever way you create an object, there is always a trade-off between performance and resource utilization when you decide where to instantiate it.

In the example, you've seen in this chapter, there is a single **ManagementController** client class that uses an implementation of the **ITenantStore** interface. In a real application, there may be many other client classes that all need **ITenantStore** instances. Depending on the specific requirements and

structure of your application, you might want each client class to have its own **ITenantStore** object, or have all the client classes share the same **ITenantStore** instance, or for different groups of client classes each have their own **ITenantStore** instance.

If every client object has its own **ITenantStore** instance, then the **ITenantStore** instance can be garbage collected along with the client object. If multiple client objects share an **ITenantStore** instance, then the class or component that instantiates the shared **ITenantStore** object must responsible for tidying it up when all the clients are finished with it.

Types of Injection

Typically, when you instantiate an object you invoke a class constructor and pass any values that the object needs as parameters to the constructor. In the example that you saw earlier in this chapter, the constructor in the **ManagementController** class expects to receive an object that implements the **ITenantStore** interface. This is an example of *constructor injection* and is the type of injection you will use most often. There are other types of injection such as *property setter injection* and *method call injection*, but they are less commonly used.

Property Setter Injection

As an alternative or in addition to passing a parameter to a constructor, you may want to set a property value when you instantiate an object in your application. The following code sample shows part of a class named **AzureTable** in an application that uses property injection to set the value of the **ReadWriteStrategy** property when it instantiates **AzureTable** object.

C#

```
public class AzureTable<T> : ...
{
  public AzureTable(StorageAccount account)
  : this(account, typeof(T).Name)
  {
  }

  ...

  public IAzureTableRWStrategy ReadWriteStrategy
    { get; set; }

  ...
}
```

Notice that the constructors are not responsible for setting the read/write strategy and that the type of the **ReadWriteStrategy** property is an interface type. You can use property setter injection to provide an instance of the **IAzureTableRWStrategy** type when your dependency injection container constructs an instance of **AzureTable<T>**.

You should only use property setter injection if the class has a usable default value for the property. While you cannot forget to call a constructor, you can forget to set a property such as the **ReadWriteStrategy** property in the example above.

You should use property setter injection when the dependency is optional. However don't use property setter injection as a technique to avoid polluting your constructor with multiple dependencies; too many dependencies might be an indicator of poor design because it is placing too much responsibility in a single class. See the single responsibility principle discussed in Chapter 1.

However, dependencies are rarely optional when you are building a LOB application. If you do have an optional dependency, consider using constructor injection and injecting an empty implementation (the Null Object Pattern.)

Method Call Injection

In a similar way to using property setter injection, you might want to invoke a method when the application instantiates an object to perform some initialization that is not convenient to perform in a constructor. The following code sample shows part of a class named **MessageQueue** in an application that uses method injection to initialize the object.

C#

```
public class MessageQueue<T> : ...
{
  ...

  public MessageQueue(StorageAccount account)
  : this(account, typeof(T).Name.ToLowerInvariant())
  {
  }

  public MessageQueue(StorageAccount account,
                      string queueName)
  {
    ...
  }

  public void Initialize(TimeSpan visibilityTimeout,
                     IRetryPolicyFactory retryPolicyFactory)
  {
    ...
  }

  ...
}
```

In this example, the **Initialize** method has one concrete parameter type and one interface parameter type. You can use method injection to provide an instance of the **IRetryPolicyFactory** type when your dependency injection container constructs an instance of **MessageQueue<T>**.

Method call injection is useful when you want to provide some additional information about the context that the object is being used in that can't be passed in as a constructor parameter.


Both property setter and method injection may be useful when you need to support legacy code that uses properties and methods to configure instances.

When You Shouldn't Use Dependency Injection

Dependency injection is not a silver bullet. There are reasons for not using it in your application, some of which are summarized in this section.

- Dependency injection can be overkill in a small application, introducing additional complexity and requirements that are not appropriate or useful.
- In a large application, it can make it harder to understand the code and what is going on because things happen in other places that you can't immediately see, and yet they can fundamentally affect the bit of code you are trying to read. There are also the practical difficulties of browsing code like trying to find out what a typical implementation of the **ITenantStore** interface actually does. This is particularly relevant to junior developers and developers who are new to the code base or new to dependency injection.
- You need to carefully consider if and how to introduce dependency injection into a legacy application that was not built with inversion of control in mind. Dependency injection promotes a specific style of layering and decoupling in a system that may pose challenges if you try to adapt an existing application, especially with an inexperienced team.
- Dependency injection is far less important in functional as opposed to object-oriented programming. Functional programming is becoming a more common approach when testability, fault recovery, and parallelism are key requirements.
- Type registration and resolving do incur a runtime penalty: very negligible for resolving, but more so for registration. However, the registration should only happen once.


Programming languages shape the way we think and the way we code. For a good exploration of the topic of dependency injection when the functional programming model is applied, see the article "Dependency Injection Without the Gymnastics" by Tony Morris.

According to Mark Seeman, using dependency injection “can be dangerous for your career because it may increase your overall knowledge of good API design. Once you learn how proper loosely coupled code can look like, it may turn out that you will have to decline lots of job offers because you would otherwise have to work with tightly coupled legacy apps.”

What are the downsides to using Dependency Injection? On StackOverflow.

Summary

In this chapter, you've seen how dependency injection differs from patterns such as the factory patterns and the service locator pattern by adopting a push model, whereby some other class or component is responsible for instantiating the dependencies and injecting them into your object’s constructor, properties, or methods. This other class or component is now responsible for composing the application by building the complete object graph, and in some cases it will also be responsible for managing the lifetime of the objects that it creates. In the next chapter, you'll see how you can use the Unity container to manage the instantiation of dependent objects and their lifetime.

Chapter 3 - Dependency Injection with Unity

Introduction

In previous chapters, you saw some of the reasons to use dependency injection and learned how dependency injection differs from other approaches to decoupling your application. In this chapter you'll see how you can use the Unity dependency injection container to easily add a dependency injection framework to your applications. On the way, you'll see some examples that illustrate how you might use Unity in a real-world application.

The Dependency Injection Lifecycle: Register, Resolve, Dispose

In the previous chapter, you saw how the **ManagementController** class has a constructor that expects to be injected with an object of type **ITenantStore**. The application must know at run time which implementation of the **ITenantStore** interface it should instantiate before it can go ahead and instantiate a **ManagementController** object. There are two things happening here: something in the application is making a decision about how to instantiate an object that implements the **ITenantStore** interface, and then something in the application is instantiating both that object and the **ManagementController** object. We will refer to the first task as *registration* and the second as *resolution*. At some point in the future, the application will finish using the **ManagementController** object and it will become available for garbage collection. At this point, it may also make sense for the garbage collector to dispose of the **ITenantStore** instance if other client classes do not share the same instance.


Typically, you perform the registration of the types that require dependency injection in a single method in your application; you should invoke this method early in your application's lifecycle to ensure that the application is aware of all of the dependencies between its classes. Unity also supports configuring the container declaratively from a configuration file.

The Unity container can manage this register, resolve, dispose cycle making it easy to use dependency injection in your applications. The following sections illustrate this cycle using a simple example. Later in this chapter you will see a more sophisticated real-world sample and learn about some alternative approaches.


You should always try to write container-agnostic code (except for the one place at the root of the application where you configure the container) in order to decouple your application from the specific dependency injection container you are using.

Register

Using the Unity container, you can register a set of mappings that determine what concrete type you require when a constructor (or property or method) identifies the type to be injected by an interface type or base class type. As a reminder, here is a copy of the constructor in the **ManagementController** class showing that it requires an injection of an object that implements the **ITenantStore** interface.

C#

```
public ManagementController(ITenantStore tenantStore)
{
  this.tenantStore = tenantStore;
}
```

The following code sample shows how you could create a new Unity container and then register the concrete type to use when a **ManagementController** instance requires an **ITenantStore** instance.

C#

```
var container = new UnityContainer();
container.RegisterType<ITenantStore, TenantStore>();
```

The **RegisterType** method shown here tells the container to instantiate a **TenantStore** object when it instantiates an object that requires an injection of an **ITenantStore** instance through a constructor, or method, or property. This example represents one of the simplest types of mapping that you can define using the Unity container. As you continue through this chapter, you will see other ways to register types and instances in the Unity container, that handle more complex scenarios and that provide greater flexibility.


You'll see later that with Unity you can also register a class type directly without a mapping from an interface type.

Resolve

The usage of the **RegisterType** method shown in the previous section defines the mapping between the interface type used in the client class and the concrete type that you want to use in the application. To instantiate the **ManagementController** and **TenantStore** objects, you must invoke the **Resolve** method.

C#

```
var controller = container.Resolve<ManagementController>();
```

Note that in this example, you do not instantiate the **ManagementController** object directly, rather you ask the Unity container to do it for you so that the container can resolve any dependencies. In this simple example, the dependency to resolve is for an **ITenantStore** object. Behind the scenes, the Unity container first constructs a **TenantStore** object and then passes it to the constructor of the **ManagementController** class.

Dispose

In the simple example shown in the previous two sections on registering and resolving types, the application stores a reference to the **ManagementController** object in the **controller** variable and the Unity container creates a new **TenantStore** instance to inject whenever you call the **Resolve** method. When the **controller** variable goes out of scope and becomes eligible for garbage collection, the **TenantStore** object will also be eligible for garbage collection.


By default, the Unity container doesn't hold a reference to the objects it creates: to change this default behavior you need to use one of the Unity lifetime managers.

Registering and Resolving in your Code

One of the original motivations, discussed in Chapter 1, for a loosely coupled design and dependency injection was maintainability. One of the ways that dependency injection can help you to create more maintainable solutions is by describing, in a single location, how to compose your application from all of its constituent classes and components. From the perspective of Unity, this is the type registration information. Therefore, it makes sense to group all of the type registrations together in a single method that you invoke very early on in your application's lifecycle; usually, directly in the application's entry point. For example, in a web application, you could invoke the method that performs all of the registrations from within the **Application_Start** method in the global.asax.cs or global.asax.vb file, in a desktop application you invoke it from the **Main** method.


You should perform all the registrations in a single location in your code or in a configuration file. This makes it easy to manage the dependencies in your application. In a highly modular application, each module might be responsible for its own registration and manage its own container.

Using a configuration file for registrations can be a brittle and error prone solution. It can also lead to the illusion that this configuration can be changed without proper testing. Consider which settings, if any, need to be configurable after your solution is deployed.

Typically, you can call the **Resolve** method when you need an instance of a particular type in your application. The section "Lifetime Management" later in this chapter discusses the options for controlling the lifetime of objects resolved from the container: for example, do you want the container return a new instance each time you resolve a particular type, or should the container maintain a reference to the instance.

Adding Unity to Your Application

As a developer, before you can write any code that uses Unity, you must configure your Visual Studio project with all of the necessary assemblies, references, and other resources that you'll need. For

information about how you can use NuGet to prepare your Visual Studio project to work with Unity, see the topic "Adding Unity to Your Application."

 NuGet makes it very easy for you to configure your project with all of the prerequisites for using Unity.

A Real-World Example

The following example is taken from a web role implemented using ASP.NET MVC. You may find it useful to open the sample application, "DIwithUnitySample," that accompanies this guide in Visual Studio while you read this section. At first sight the contents of this **RegisterTypes** method (in the **ContainerBootstrapper** class in the **Surveys** project) might seem to be somewhat complex; the next section will discuss the various type registrations in detail, and the following section will describe how the application uses these registrations to resolve the types it needs at runtime. This example also illustrates how you should perform all of the type registration in a single method in your application.

 It's useful to adopt a standard name for the class that contains your type registration code; for example **ContainerBootstrapper**.

C#

```
public static void RegisterTypes(IUnityContainer container)
  {
    var storageAccountType = typeof(StorageAccount);
    var retryPolicyFactoryType = typeof(IRetryPolicyFactory);

    // Instance registration
    StorageAccount account =
      ApplicationConfiguration.GetStorageAccount("DataConnectionString");
    container.RegisterInstance(account);

    // Register factories
    container
      .RegisterInstance<IRetryPolicyFactory>(
        new ConfiguredRetryPolicyFactory())
      .RegisterType<ISurveyAnswerContainerFactory,
        SurveyAnswerContainerFactory>(
          new ContainerControlledLifetimeManager());

    // Register table types
    container
      .RegisterType<IDataTable<SurveyRow>, DataTable<SurveyRow>>(
        new InjectionConstructor(storageAccountType,
```

```
          retryPolicyFactoryType, Constants.SurveysTableName))
      ...

    // Register message queue type, use typeof with open generics
    container
      .RegisterType(
          typeof(IMessageQueue<>),
          typeof(MessageQueue<>),
          new InjectionConstructor(storageAccountType,
            retryPolicyFactoryType, typeof(String)));

    ...

    // Register store types
    container
      .RegisterType<ISurveyStore, SurveyStore>()
      .RegisterType<ITenantStore, TenantStore>()
      .RegisterType<ISurveyAnswerStore, SurveyAnswerStore>(
        new InjectionFactory((c, t, s) => new SurveyAnswerStore(
          container.Resolve<ITenantStore>(),
          container.Resolve<ISurveyAnswerContainerFactory>(),
          container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
            new ParameterOverride(
              "queueName", Constants.StandardAnswerQueueName)),
          container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
            new ParameterOverride(
              "queueName", Constants.PremiumAnswerQueueName)),
          container.Resolve<IBlobContainer<List<String>>>())));
  }
}
```

To see the complete **ContainerBootstrapper** class, you can open the DIwithUnitySample sample application that accompanies this guidance.

Figure 1 illustrates the object graph that the container will generate if a client resolves the **ISurveyAnswerStore** type from the container with the type registrations shown in the previous code sample.

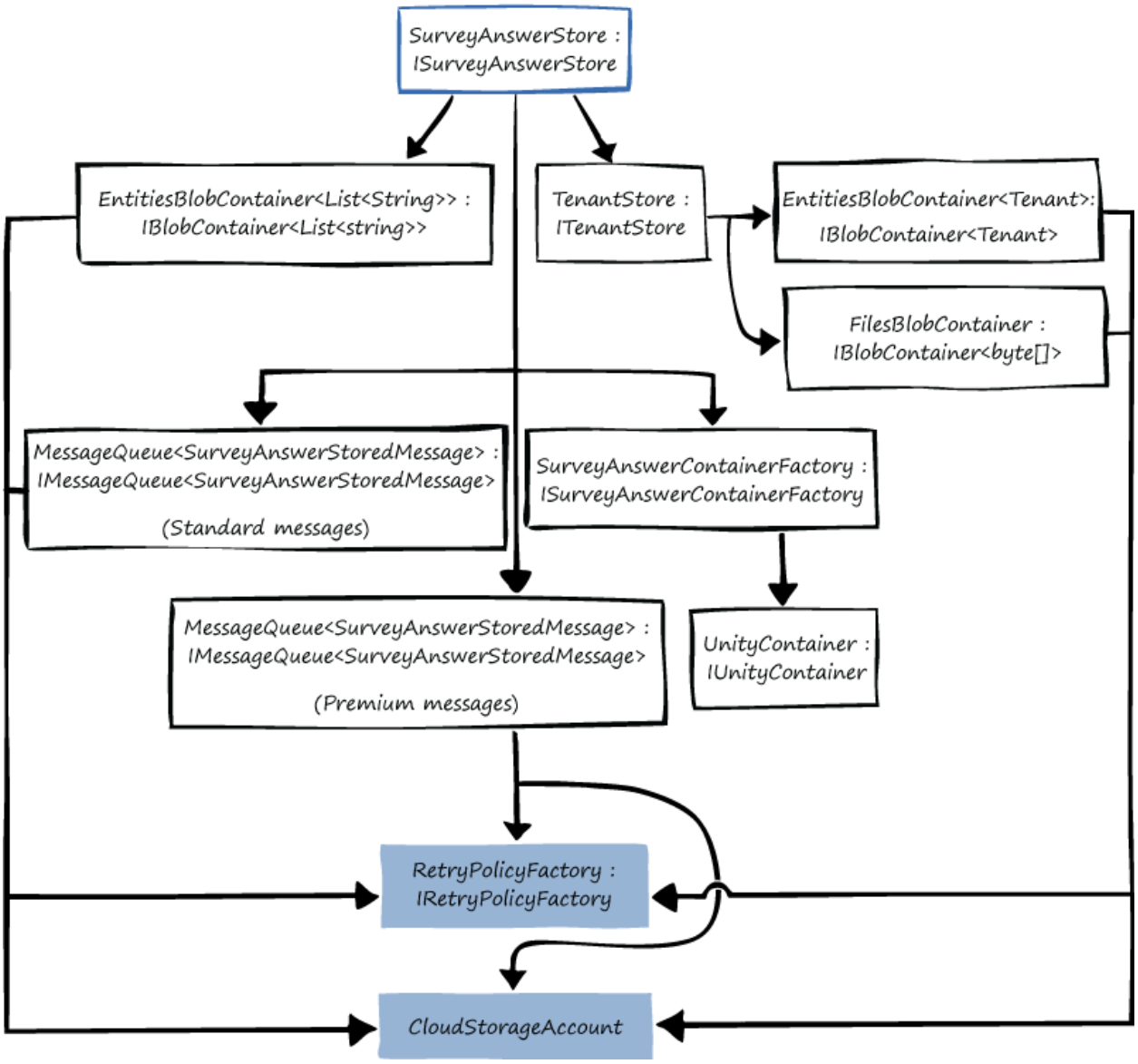

Figure 1 - Resolving the ISurveyAnswerStore type

Figure 1 illustrates the object graph that the container creates when you resolve the **ISurveyAnswerStore** type from the registrations shown in the previous code listing. There are some important points to note from Figure 1.

- The container injects the **SurveyAnswerStore** with five objects that the container itself resolves: a **TenantStore** object, a **SurveyAnswerContainerFactory** object, an **EntitiesBlobContainer** object, and two **MessageQueue** objects. Note that an explicit factory delegate is used to determine what must be injected to create the store.
- The container also resolves additional objects such as an **EntitiesBlobContainer** object and a **FilesBlobContainer** object to inject into the **TenantStore** instance.

- Many of the objects instantiated by the container share the same instances of the **RetryPolicyFactory** and **CloudStorageAccount** objects which are registered using the **RegisterInstance** method. Instance registration is discussed in more detail later in this chapter.
- The container injects the **SurveyAnswerContainerFactory** instance with an instance of the Unity container. Note that as a general rule, this is not a recommended practice.

The following sections discuss all of these points (and more) in detail. Figure 1 is intended to give an idea of what you can achieve with dependency injection in your applications.

Type Registrations in the Example

The previous code listing gives examples of many of the different types of registration that you can perform with the Unity container. This section examines each part of the registration individually.


Bear in mind, that if your registrations start to become too complicated or fragile, you are probably doing it wrong.

Instance Registration

The simplest type of registration is instance registration where the container is responsible for maintaining a reference to a singleton instance of a type. For example:

C#

```
StorageAccount account =
  ApplicationConfiguration.GetStorageAccount("DataConnectionString");
container.RegisterInstance(account);
```

Here, instead of registering a mapping for a type to resolved later, the application creates a **CloudStorageAccount** object and registers the instance with the container. This means that the **CloudStorageAccount** object is created at registration time, and that only a single instance of the object exists in the container. This single instance is shared by many of the other objects that the container instantiates. Figure 1 shows that many of the objects that the container creates when a client resolves the **ISurveyAnswerStore** type share this **CloudStorageAccount** object instance.

You can also use the **ContainerControlledLifetimeManager** class with the **RegisterType** method to create a singleton instance where the container maintains a reference to the object. The section "Lifetime Management" later in this chapter covers this in more detail.

Simple Type Registration

The most common type registration you will see maps an interface type to a concrete type. For example:

C#

```
container.RegisterType<ISurveyStore, SurveyStore>();
```

Later, you can resolve the **ISurveyStore** type as shown in the following example, and the container will inject any of the required dependencies into the **SurveyStore** object that it creates.

C#

```
var surveyStore = container.Resolve<ISurveyStore>();
```

If the **SurveyStore** class has multiple constructors with the same number of parameters you can use either the **InjectionConstructor** attribute, the API, or the configuration file to disambiguate between the different **SurveyStore** constructors. However, although **InjectionConstructor** attributes are easy to use, they do couple your code to the container.

In most cases, components should have a single constructor and the constructor defines the dependencies of that component.

Constructor Injection

The following code sample shows the constructor for the **DataTable** class that takes three parameters.

C#

```
public DataTable(StorageAccount account,
  IRetryPolicyFactory retryPolicyFactory, string tableName)
{
  ...
}
```

The registrations for the **DataTable** types in the container includes an **InjectionConstructor** that defines how the container should resolve the parameter types. The container passes to the constructor references to the registered **StorageAccount** and **RetryPolicyFactory** instances, and a string that specifies the name of the table to use.

C#

```
container
  .RegisterType<IDataTable<SurveyRow>, DataTable<SurveyRow>>(
    new InjectionConstructor(storageAccountType,
      retryPolicyFactoryType, Constants.SurveysTableName))
  .RegisterType<IDataTable<QuestionRow>, DataTable<QuestionRow>>(
    new InjectionConstructor(storageAccountType,
      retryPolicyFactoryType, Constants.QuestionsTableName));
```

The sample application uses a similar approach to register the blob container types it uses:

C#

```
container
```

```
  .RegisterType<IBlobContainer<List<string>>,
    EntitiesBlobContainer<List<string>>>(
      new InjectionConstructor(storageAccountType,
        retryPolicyFactoryType, Constants.SurveyAnswersListsBlobName))
  .RegisterType<IBlobContainer<Tenant>,
    EntitiesBlobContainer<Tenant>>(
      new InjectionConstructor(storageAccountType,
        retryPolicyFactoryType, Constants.TenantsBlobName))
  .RegisterType<IBlobContainer<byte[]>,
    FilesBlobContainer>(
      new InjectionConstructor(storageAccountType,
        retryPolicyFactoryType, Constants.LogosBlobName, "image/jpeg"))
  .RegisterType<IBlobContainer<SurveyAnswer>,
    EntitiesBlobContainer<SurveyAnswer>>(
      new InjectionConstructor(storageAccountType,
        retryPolicyFactoryType, typeof(string)));
```

Unity supports property and method injection in addition to the constructor injection shown in this example. If you use property injection then, as with any property, you should ensure that any properties have a useful default value. It is easy to forget to set a property.

Both the storage account and retry policy factory are singletons. The storage account is registered using the **RegisterInstance** method, the retry policy factory is registered using the **ContainerControlledLifetimeManager** class that you'll learn more about later in this chapter.

Registering Open Generics

The example code uses a slightly different approach to register the message queue types: it uses an overload of the **RegisterTypes** method that takes types as standard parameters instead of using type parameters.

C#

```
container
  .RegisterType(
    typeof(IMessageQueue<>),
    typeof(MessageQueue<>),
    new InjectionConstructor(storageAccountType,
      retryPolicyFactoryType, typeof(string)));
```

This approach enables you to resolve the message queue type with any type parameter. The example uses the **SurveyAnswerStoredMessage** type:

C#

```
container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(...);
```

Parameter Overrides

The **ContainerBootstrapper** class contains several examples where one of the **InjectionConstructor** constructor parameters is **typeof(string)**. For example, in the message queue registration and in the blob container for survey answers registration:

C#

```
container.RegisterType(
  typeof(IMessageQueue<>),
  typeof(MessageQueue<>),
  new InjectionConstructor(storageAccountType,
    retryPolicyFactoryType, typeof(string)));

...

container.RegisterType<IBlobContainer<SurveyAnswer>,
  EntitiesBlobContainer<SurveyAnswer>>(
    new InjectionConstructor(storageAccountType,
      retryPolicyFactoryType, typeof(string)));
```

The container does not include a registration that it can use to resolve this type. Therefore, when you resolve either the **IMessageQueue<>** or **IBlobContainer<SurveyAnswer>** types, you must provide a value for this parameter otherwise the resolution will fail. This provides a convenient method to pass parameter values that aren't known at registration time to instances created by the container using the **ParameterOverride** type.

Resolving Types in the Example

The example solution performs the type registration described in the previous section in three locations: in a standalone application that initializes the storage, in the web application's start-up phase, and in a factory class.

Simple Resolve

The usage in this simple standalone application is straightforward: it calls the **RegisterTypes** method to perform all the registration, resolves a number of objects, and then invokes the **Initialize** method on each of them to perform some initialization work before the application terminates. The following code sample shows this.

C#

```
static void Main(string[] args)
{
  using (var container = new UnityContainer())
  {
    ContainerBootstrapper.RegisterTypes(container);

    container.Resolve<ISurveyStore>().Initialize();
    container.Resolve<ISurveyAnswerStore>().Initialize();
```

```
    container.Resolve<ITenantStore>().Initialize();

    Console.WriteLine("Done");
    Console.ReadLine();
  }

}
```

In this example, after the **Initialization** methods have run, the container is disposed.

Resolving in an MVC Application

The usage in the MVC application is more sophisticated: the application configures a container that the application will use at start-up, and then resolves the various types as and when it needs them. Remember that this is an ASP.NET MVC application; therefore, the container must be able to inject the MVC controller classes with the various store and queue objects that they need. The "Unity bootstrapper for ASP.NET MVC" NuGet package (search for Unity3 in the NuGet package manager) simplifies this by adding libraries and source code to the project in Visual Studio. The following code sample shows the **RegisterTypes** method in the **UnityConfig** class that the NuGet package added to the project; you can choose to load the Unity configuration from your configuration file or add the registrations directly.

C#

```
public static void RegisterTypes(IUnityContainer container)
{
    // NOTE: To load from web.config uncomment the line below...
    // container.LoadConfiguration();

    // TODO: Register your types here
    // container.RegisterType<IProductRepository, ProductRepository>();
}
```

The "Unity bootstrapper for ASP.NET MVC" provides a **UnityDependencyResolver** class that resolves controllers from the container. If you need to configure injection for the controller classes then you need to add the registration manually or add injection attributes to the controller classes

The following code sample shows part of the **ManagementController** class that custom factory class can resolve along with its dependency on the **ITenantStore** type from the registration information.

C#

```
public class ManagementController : Controller
{
  private readonly ITenantStore tenantStore;

  public ManagementController(ITenantStore tenantStore)
  {
    this.tenantStore = tenantStore;
```

```
  }
    ...
}
```

Using the Per Request Lifetime Manager in MVC and WebAPI Application

The previous example showed how to use the "Unity bootstrapper for ASP.NET MVC" NuGet package to handle registering and resolving controllers in an MVC application. The package also includes a **PerRequestLifetime** manager that you can use in an MVC application. This lifetime manager enables you to create instances of registered types that behave like singletons within the scope of an HTTP request.

If you are working with an ASP.NET Web API project, there is a "Unity bootstrapper for ASP.NET WebApi" NuGet package that offers equivalent features (search for Unity3 in the NuGet package manager). You can use both the "Unity bootstrapper for ASP.NET WebApi" and "Unity bootstrapper for ASP.NET MVC" packages in the same project and they will share a single container configuration class.

> There are third-party solutions available that offer similar support for ASP.NET WCF applications.

Resolving with Run Time Information

You don't always know the values that you need to construct a dependency at design time. In the example shown below, a user provides the name of the blob container that the application must create at run time. In this example, type resolution occurs in a factory class that determines the value of a constructor parameter at registration time. The following code sample shows this factory class.

C#

```
public class SurveyAnswerContainerFactory : ISurveyAnswerContainerFactory
{
  private readonly IUnityContainer unityContainer;

  public SurveyAnswerContainerFactory(IUnityContainer unityContainer)
  {
    this.unityContainer = unityContainer;
  }

  public IBlobContainer<SurveyAnswer> Create(string tenant, string surveySlug)
  {
    var blobContainerName = string.Format(
        CultureInfo.InvariantCulture,
        "surveyanswers-{0}-{1}",
        tenant.ToLowerInvariant(),
        surveySlug.ToLowerInvariant());
    return this.unityContainer.Resolve<IBlobContainer<SurveyAnswer>>(
        new ParameterOverride("blobContainerName", blobContainerName));
  }
}
```

In this example, the **Resolve** method uses a parameter override to provide a value for the **blobContainerName** parameter to the constructor of the **EntitiesBlobContainer** class that is registered in the container instance injected into the **SurveyAnswerContainerFactory** object. Figure 1 shows how the **SurveyAnswerContainerFactory** object is injected with a Unity container instance when it is resolved.


Because the application supplies the name of the blob container to create at run time, the factory class uses a parameter override to supply this value to the **Resolve** method.

You saw previously how the application registered the **IBlobContainer<SurveyAnswer>>** type using a string parameter in the injection constructor. Without the parameter override, this registration would fail because the container cannot resolve the string type.

You can also see the parameter overrides in use in the **ContainerBootstrapper** class as part of the registration of the survey answer store. In this example, the parameter overrides provide the name of the message queues to create. The parameter overrides are supplied at resolve time when the registered **InjectionFactory** executes.

C#

```
container
  .RegisterType<ISurveyAnswerStore, SurveyAnswerStore>(
    new InjectionFactory((c, t, s) => new SurveyAnswerStore(
      container.Resolve<ITenantStore>(),
      container.Resolve<ISurveyAnswerContainerFactory>(),
      container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
        new ParameterOverride(
          "queueName", Constants.StandardAnswerQueueName)),
      container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
        new ParameterOverride(
          "queueName", Constants.PremiumAnswerQueueName)),
      container.Resolve<IBlobContainer<List<string>>>())));
```

Registration

In this section, you'll learn more about how you can register types with the Unity container and the advantages and disadvantages of the different approaches. All of the examples you've seen so far have registered types with the Unity container programmatically by using methods such as **RegisterType** and **RegisterInstance**.

Programmatically configuring a Unity container at runtime is convenient, and if you keep all of your registration code together, it makes it easy to change those registrations when you modify the application. However, it does mean that you must recompile the application if you need to change your registrations. In some scenarios, you may want a mechanism that enables you to change the

registrations in a configuration file and cause your application to use a different set of concrete classes at run time.

Named Type Registrations

Previously, you saw an example of how you can use parameter overrides to provide the name of the message queue when you are resolving the **IMessageQueue** type. An alternative approach for this scenario is to use named type registrations. In this case the message queue registration looks like the following where the two alternative registrations are named "Standard" and "Premium":

C#

```
container
  .RegisterType<IMessageQueue<SurveyAnswerStoredMessage>,
    MessageQueue<SurveyAnswerStoredMessage>>(
    "Standard",
    new InjectionConstructor(storageAccountType, retryPolicyFactoryType,
      Constants.StandardAnswerQueueName))
  .RegisterType<IMessageQueue<SurveyAnswerStoredMessage>,
    MessageQueue<SurveyAnswerStoredMessage>>(
    "Premium",
    new InjectionConstructor(storageAccountType, retryPolicyFactoryType,
      Constants.PremiumAnswerQueueName));
```

With this registration, you can now resolve the message queue parameters to the **SurveyAnswerStore** constructor as follows using the named registrations:

C#

```
container
  .RegisterType<ISurveyAnswerStore, SurveyAnswerStore>(
    new InjectionFactory((c, t, s) => new SurveyAnswerStore(
      container.Resolve<ITenantStore>(),
      container.Resolve<ISurveyAnswerContainerFactory>(),
      container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
      "Standard"),
      container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
      "Premium"),
      container.Resolve<IBlobContainer<List<string>>>())));
```

Design-Time Configuration

Unity enables you to load a collection of registrations from a configuration file into a container. For example, you could add the following sections to your app.config or web.config file to register mapping from the **ITenantStore** interface to the **TenantStore** class.

> You cannot use design-time configuration for Unity containers in Windows Store apps. For more information, see Appendix A – Unity and Windows Store apps.

XML

```
<configuration>
  <configSections>
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration" />
  </configSections>
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <namespace name="Tailspin.Web.Survey.Shared.Stores" />
    <container>
      <register type="ITenantStore" mapTo="TenantStore" />
    </container>
  </unity>
</configuration>
```

For more information about the structure of this configuration file, see the topic Design Time Configuration.

To load the registration details from the configuration file, you can use the following code. The **LoadConfiguration** extension method is defined in the Microsoft.Practices.Unity.Configuration namespace.

C#

```
IUnityContainer container = new UnityContainer();
container.LoadConfiguration();
```

Defining the registrations in a configuration file means that it's possible to make changes without recompiling the application. This can be useful for customized deployments and troubleshooting. You can also adopt a hybrid approach.

If your Unity configuration includes any sensitive information, you should encrypt it. For more information, see Encrypting Configuration Information Using Protected Configuration on MSDN.

Registration by Convention

This feature (also known as auto-registration) is intended to minimize the amount of type registration code that you need to write. You may find it useful to open the sample application, "OtherUnitySamples," that accompanies this guide in Visual Studio while you read this section. Rather than specify each type mapping individually, you can direct the Unity container to scan a collection of assemblies and then automatically register multiple mappings based on a set of rules. If you only have a handful of simple registrations, it doesn't make sense to use this feature, but if you have many types to register it will save you a considerable amount of effort.


In many cases, you don't need to register a type to resolve it because Unity's auto-wiring will figure it out for you. You do need to explicitly configure mappings though, and for this the registration by convention feature may prove useful.

This feature is only supported when you are configuring Unity programmatically. You can't specify registration by convention in the configuration file.

The registration by convention feature can scan a collection of assemblies, and then create a set of mappings in a Unity container for some or all of the types it discovers in the assemblies. Additionally, you can specify the lifetime managers to use (these are described in more detail later in this chapter) and the details of any injection parameters. You can easily modify the set of rules included out-of-the-box to implement more sophisticated scenarios.


Registration by convention is intended to simplify registering types with the Unity container when you have a large number of types that must be registered with similar settings.

The following samples illustrate how you can use the registration by convention feature to create mappings in the Unity container using the **RegisterTypes** method. These examples are based on the code in the OtherUnitySamples Visual Studio solution included with this guidance. The **RegisterTypes** method has the following parameters:

Parameter	Description
Types	This parameter is an enumerable collection of types that you want to register with the container. These are the types that you want to register directly or create mappings to. You can create this collection by providing a list of types directly or by using one of the methods of the built-in AllClasses helper class: for example, the method FromLoadedAssemblies loads all of the available types from the currently loaded assemblies. You can use LINQ to filter this enumeration.
getFromTypes	This optional parameter identifies the types you want to map from in the container. The built-in WithMappings helper class provides several options for this mapping strategy: for example, the MatchingInterface property creates mappings where there are interfaces and implementations that follow the naming convention ITenant and Tenant.
getName	This optional parameter enables you to control whether to create default registrations or named registrations for the types. The built-in helper class WithName, enables you to choose between using default registrations or named registrations that use the type name.

getLifeTimeManager	This optional parameter enables you to select from the built-in lifetime managers.
getInjectionMembers	This optional parameter enables you to provide definitions for any injection members for the types that you are registering.
overwriteExistingMappings	This optional parameter enables you to control how the method behaves if it detects an attempt to overwrite an existing mapping in the Unity container. By default, the RegisterTypes method throws an exception if it detects such an attempt. If this parameter is true, the method silently overwrites an existing mapping with a new one based on the values of the other parameters.

The first example shows how you can create registrations for all the types that implement an interface where the **ITenant**/**Tenant** naming convention is in use.

C#

```
using (var container = new UnityContainer())
{
  container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),
    WithMappings.MatchingInterface,
    WithName.Default);
}
```

This creates a set of transient registrations that map interfaces to types in the container.

The following example creates named registrations and uses the **ContainerControlledLifetimeManager** type to ensure that resolving from the container results in singletons.

C#

```
using (var container = new UnityContainer())
      {
        container.RegisterTypes(
          AllClasses.FromLoadedAssemblies(),
          WithMappings.MatchingInterface,
          WithName.TypeName,
          WithLifetime.ContainerControlled);
      }
```

Because this example is using the lifetime manager, it registers all loaded types in addition to mapping any interfaces to their matching types.

Lifetime managers are discussed later in this chapter in the section "Lifetime Management."

In some scenarios, you may need to combine registration by convention with explicit registration. You can use registration by convention to perform the basic registration of multiple types, and then explicitly add information for specific types. The following example adds an **InjectionConstructor** to the type registration for the **TenantStore** class that was one of the types registered by calling the **RegisterTypes** method.

C#

```
using (var container = new UnityContainer())
{
  container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),
    WithMappings.MatchingInterface,
    WithName.Default,
    WithLifetime.ContainerControlled);

  // Provide some additional information for this registration
  container.RegisterType<TenantStore>(new InjectionConstructor("Adatum"));
}
```

The examples you've seen so far use the **FromLoadedAssemblies** method to provide a list of assemblies to scan for types; you may want to filter this list so that you only register a subset of the types from these assemblies. The following sample shows how to create a filter based on the namespace containing the type. In this example, only types in the **OtherUnitySamples** namespace are registered in the container.

C#

```
using (var container = new UnityContainer())
{
  container.RegisterTypes(
    AllClasses.FromLoadedAssemblies().Where(
      t => t.Namespace == "OtherUnitySamples"),
    WithMappings.MatchingInterface,
    WithName.Default,
    WithLifetime.ContainerControlled);
}
```

The next example illustrates the use of the **getInjectionMembers** parameter: this enables you specify types that should be injected when the registered type is resolved from the container. Note that any types to be injected will be injected into all the types registered in the container by the call to the **RegisterTypes** method. The following example assumes that all of the types registered have a constructor with a string parameter: any attempt to resolve a type without such a constructor parameter will result in an exception being thrown from the container.

C#

```
using (var container = new UnityContainer())
{
  container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),
    WithMappings.MatchingInterface,
    getInjectionMembers: t => new InjectionMember[]
    {
      new InjectionConstructor("Adatum")
    });
```

```
}
```

A more practical use of the **getInjectionMembers** method is to use it to configure interception for all of the registered types (for more information about interception in Unity, see Chapter 5, "Interception with Unity"). In the following example, the registration by convention injects all of the registered types with the custom **LoggingInterceptionBehavior** type using virtual method interception.

C#

```
using (var container = new UnityContainer())
{
  container.AddNewExtension<Interception>();
  container.RegisterTypes(
    AllClasses.FromLoadedAssemblies().Where(
      t => t.Namespace == "OtherUnitySamples"),
    WithMappings.MatchingInterface,
    getInjectionMembers: t => new InjectionMember[]
    {
      new Interceptor<VirtualMethodInterceptor>(),
      new InterceptionBehavior<LoggingInterceptionBehavior>()
    });
}
```

You can keep the list of classes in the **OtherUnitySamples** namespace to use in other calls to the **RegisterTypes** method to avoid the need to re-scan the assemblies.

Note how this example uses a filter to ensure that only types in the **OtherUnitySamples** namespace are registered in the container. Without this filter, the container will try to inject the interceptor into all the types from the loaded assemblies: this includes the **LoggingInterceptionBehavior** type itself and this results in a stack overflow.

The block provides helper classes such as the **AllClasses** class that you can use as parameters to the **RegisterTypes** method. You can create your own helper classes, or use a lambda such the one used for the **getInjectionMembers** parameter in the previous example, to customize the behavior of registration by convention to your own requirements.

The following artificial example shows how the **overwriteExistingMappings** parameter prevents the **RegisterTypes** method from throwing an exception if you attempt to overwrite an existing mapping. The result is that the container contains a mapping of the **ITenantStore** interface type to the **TenantStore2** type.

C#

```
using (var container = new UnityContainer())
{
```

```
  container.RegisterTypes(
    new[] { typeof(TenantStore), typeof(TenantStore2) },
    t => new[] { typeof(ITenantStore) },
    overwriteExistingMappings: true);
```

The mapping for the last type overwrites the previous mapping in the container.

 Be cautious when you use registration by convention and consider the implications of registering all of the types discovered by the **RegisterTypes** method using the same options.

You can extend the abstract **RegistrationConvention** class to define a reusable convention that you can pass to the **RegisterTypes** method as a single parameter. The following sample shows a how you can extend the **RegistrationConvention** class.

C#

```
class SampleConvention : RegistrationConvention
{
  public override Func<Type, IEnumerable<Type>> GetFromTypes()
  {
    return t => t.GetTypeInfo().ImplementedInterfaces;
  }

  public override Func<Type, IEnumerable<InjectionMember>> GetInjectionMembers()
  {
    return null;
  }

  public override Func<Type, LifetimeManager> GetLifetimeManager()
  {
    return t => new ContainerControlledLifetimeManager();
  }

  public override Func<Type, string> GetName()
  {
    return t => t.Name;
  }

  public override IEnumerable<Type> GetTypes()
  {
    yield return typeof(TenantStore);
    yield return typeof(SurveyStore);
  }
}
```

Registration by Convention and Generic Types

You can use registration by convention to register generic types. The following interface and class definitions use a generic type and you can use the **WithMappings.FromMatchingInterface** helper method to create a mapping between these types in the container.

C#

```
public interface ICustomerStore<T>
{
  ...
}

public class CustomerStore<T> : ICustomerStore<T>
{
  ...
}
```

This registers the open generic types but enables you to resolve using a closed generic. For example, you could resolve instances using the following code.

C#

```
var blobCustomerStore = container.Resolve<ICustomerStore<BlobStorage>>();
var tableCustomerStore = container.Resolve<ICustomerStore<TableStorage>>();
```

It's also possible to combine using registration by convention to register mappings for open generic types with a specific registration a closed generic type as shown in the following sample. The closed type registration will always take priority.

C#

```
container.RegisterTypes(
  // Registers open generics
  AllClasses.FromLoadedAssemblies(),
  WithMappings.FromMatchingInterface,
  WithName.Default);

// Add a registration for a closed generic type
container.RegisterType<ICustomerStore<TableStorage>,
                       CustomerStore<TableStorage>>();
```

Using Child Containers

Although you can use named registrations to define different mappings for the same type, an alternative approach is to use a child container. The following code sample illustrates how to use a child container to resolve the **IMessageQueue<SurveyAnswerStoredMessage>** type with a different connection string.

C#

```
var storageAccountType = typeof(StorageAccount);

var storageAccount = ApplicationConfiguration
```

```
  .GetStorageAccount("DataConnectionString");
container.RegisterInstance(storageAccount);

container
  .RegisterType<IMessageQueue<SurveyAnswerStoredMessage>,
      MessageQueue<SurveyAnswerStoredMessage>>(
        "Standard",
        new InjectionConstructor(storageAccountType,
          "StandardAnswerQueue"),
        retryPolicyFactoryProperty);

var childContainer = container.CreateChildContainer();
var alternateAccount = ApplicationConfiguration
  .GetStorageAccount("AlternateDataConnectionString");
childContainer.RegisterInstance(alternateAccount);

childContainer
  .RegisterType<IMessageQueue<SurveyAnswerStoredMessage>,
    MessageQueue<SurveyAnswerStoredMessage>>(
      "Standard",
        new InjectionConstructor(storageAccountType,
          "StandardAnswerQueue"),
        retryPolicyFactoryProperty);
```

You can now resolve the type **IMessageQueue<SurveyAnswerStoredMessage>** from either the original parent container or the child container. Depending on which container you use, the **MessageQueue** instance is injected with a different set of account details.

The advantage of this approach over using different named registrations, is that if you attempt to resolve a type from the child container and that type is not registered in the child container, then Unity will automatically fall back to try and resolve the type from the parent container.

> You can also use child containers to manage the lifetime of objects. This use of child containers is discussed later in this chapter.

Viewing Registration Information

You can access the registration data in the container programmatically if you want to view details of the registration information. The following code sample shows a basic approach to viewing the registrations in a container.

C#

```
Console.WriteLine("Container has {0} Registrations:",
  container.Registrations.Count());
foreach (ContainerRegistration item in container.Registrations)
{
  Console.WriteLine(item.GetMappingAsString());
}
```

This example uses an extension method called **GetMappingAsString** to display formatted output. The output using the registrations shown at the start of this chapter looks like the following:

Output

```
Container has 14 Registrations:
+ IUnityContainer  '[default]'  Container
+ StorageAccount  '[default]'  ContainerControlled
+ IRetryPolicyFactory  '[default]'  ContainerControlled
+ IDataTable`1<SurveyRow> -> DataTable`1<SurveyRow>  '[default]'  Transient
+ IDataTable`1<QuestionRow> -> DataTable`1<QuestionRow>  '[default]'  Transient
+ IMessageQueue`1<SurveyAnswerStoredMessage> ->
  MessageQueue`1<SurveyAnswerStoredMessage>  'Standard'  Transient
+ IMessageQueue`1<SurveyAnswerStoredMessage> ->
  MessageQueue`1<SurveyAnswerStoredMessage>  'Premium'  Transient
+ IBlobContainer`1<List`1<String>> -> EntitiesBlobContainer`1<List`1<String>>
  '[default]'  Transient
+ IBlobContainer`1<Tenant> -> EntitiesBlobContainer`1<Tenant>  '[default]'  Transient
+ IBlobContainer`1<Byte[]> -> FilesBlobContainer  '[default]'  Transient
+ ISurveyStore -> SurveyStore  '[default]'  Transient
+ ITenantStore -> TenantStore  '[default]'  Transient
+ ISurveyAnswerStore -> SurveyAnswerStore  '[default]'  Transient
+ ISurveyAnswerContainerFactory -> SurveyAnswerContainerFactory  '[default]'
  Transient
```

Having a utility that can display your registrations can help troubleshoot issues. This is especially useful if you define the registrations in a configuration file. However, getting registration information is expensive and you should only do it to troubleshoot issues.

The following code sample shows the extension method that creates the formatted output.

C#

```
static class ContainerRegistrationsExtension
{
  public static string GetMappingAsString(
    this ContainerRegistration registration)
  {
    string regName, regType, mapTo, lifetime;

    var r = registration.RegisteredType;
    regType = r.Name + GetGenericArgumentsList(r);

    var m = registration.MappedToType;
    mapTo = m.Name + GetGenericArgumentsList(m);

    regName = registration.Name ?? "[default]";
```

```
    lifetime = registration.LifetimeManagerType.Name;
    if (mapTo != regType)
    {
      mapTo = " -> " + mapTo;
    }
    else
    {
      mapTo = string.Empty;
    }
    lifetime = lifetime.Substring(
      0, lifetime.Length - "LifetimeManager".Length);
    return String.Format(
      "+ {0}{1}  '{2}'  {3}", regType, mapTo, regName, lifetime);
  }

  private static string GetGenericArgumentsList(Type type)
  {
    if (type.GetGenericArguments().Length == 0) return string.Empty;
    string arglist = string.Empty;
    bool first = true;
    foreach (Type t in type.GetGenericArguments())
    {
      arglist += first ? t.Name : ", " + t.Name;
      first = false;
      if (t.GetGenericArguments().Length > 0)
      {
        arglist += GetGenericArgumentsList(t);
      }
    }
    return "<" + arglist + ">";
  }
}
```

Resolving

You have already seen some sample code that shows some simple cases of resolving types from the Unity container. For example:

C#

```
var surveyStore = container.Resolve<ISurveyStore>();
container.Resolve<IMessageQueue<SurveyAnswerStoredMessage>>(
  "Premium");
```

The second of these two examples shows how to resolve a named type registration from the container.

You can override the registration information in the container when you resolve a type. The following code sample uses a dependency override to specify the type of controller object to create:

C#

```
protected override IController GetControllerInstance(
  RequestContext requestContext, Type controllerType)
{
  return this.container.Resolve(controllerType,
    new DependencyOverride<RequestContext>(requestContext)) as IController;
}
```

This example assumes that there is no registration in the container for the controller type that is passed to the **GetControllerInstance** method, so the dependency override defines a registration as the type is resolved. This enables the container to resolve any other dependencies in the controller class. You can use a dependency override to override an existing registration in addition to providing registration information that isn't registered with the container.

The **SurveyAnswerContainerFactory** class uses a parameter override to specify the value of a constructor parameter that the application cannot know until run time.

Resolving in an ASP.NET Web Application

In the example shown earlier in this chapter, you saw how to integrate Unity into an MVC application, so that you can use Unity to resolve any dependencies in your MVC controller classes by creating a custom MVC controller factory.

Standard ASP.NET web applications face a similar problem: how do you resolve any dependencies in your web page classes when you have no control over how and when ASP.NET instantiates your page objects. The aExpense reference implementation demonstrates how you can address this issue.

The following code sample shows part of a page class in the aExpense web application.

C#

```
public partial class Default : Page
{
    [Dependency]
    public IExpenseRepository Repository { get; set; }

    protected void Page_Init(object sender, EventArgs e)
    {
        this.ViewStateUserKey = this.User.Identity.Name;
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        var expenses = Repository.GetExpensesByUser(this.User.Identity.Name);
        this.MyExpensesGridView.DataSource = expenses;
        this.DataBind();
    }
...
}
```

This example shows a property, of type **IExpenseRepository** decorated with the **Dependency** attribute, and some standard page life-cycle methods, one of which uses the **Repository** property. The **Dependency** attribute marks the property for property setter injection by the Unity container.

The following code sample shows the registration of the **IExpenseRepository** type.

C#

```
public static void Configure(IUnityContainer container)
{
    container
        .RegisterInstance<IExpenseRepository>(new ExpenseRepository())
        .RegisterType<IProfileStore, SimulatedLdapProfileStore>()
        .RegisterType<IUserRepository, UserRepository>(new
            ContainerControlledLifetimeManager());
    ...
}
```

The following code sample, from the Global.asax.cs file, shows how the web application performs the type registration in the **Application_Start** method, and uses the **BuildUp** method in the **Application_PreRequestHandlerExecute** method to perform the type resolution.

C#

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        IUnityContainer container = Application.GetContainer();
        ContainerBootstrapper.Configure(container);
    }
    ...

    protected void Application_PreRequestHandlerExecute(
        object sender, EventArgs e)
    {
        System.Web.UI.Page handler =
            HttpContext.Current.Handler as System.Web.UI.Page;

        if (handler != null)
        {
            Microsoft.Practices.Unity.IUnityContainer container =
                Application.GetContainer();

            if (container != null)
            {
                container.BuildUp(handler.GetType(), handler);
            }
        }
    }
}
```

The **BuildUp** method passes an existing object, in this case the ASP.NET page object, through the container so that the container can inject any dependencies into the object.

Using the **BuildUp** method, you can only perform property and method injection. You cannot perform constructor injection because the object has already been created.

Resolving in a WCF Service

If you want to use Unity to automatically resolve types in a WCF service, you need to modify the way that WCF instantiates the service so that Unity can inject any dependencies. The example in this section is based on a simple WCF calculator sample, and the following example code shows the interface and part of the service class. You may find it useful to open the sample application, "UnityWCFSample," that accompanies this guide in Visual Studio while you read this section. The example uses constructor injection, but you could just as easily use method or property setter injection if required.

C#

```
[ServiceContract(Namespace = "http://Microsoft.ServiceModel.Samples")]
public interface ICalculator
{
  [OperationContract]
  double Add(double n1, double n2);
  [OperationContract]
  double Subtract(double n1, double n2);
  [OperationContract]
  double Multiply(double n1, double n2);
  [OperationContract]
  double Divide(double n1, double n2);
}

public class CalculatorService : ICalculator
{
  private ICalculationEngine calculatorEngine;

  public CalculatorService(ICalculationEngine calculatorEngine)
  {
    this.calculatorEngine = calculatorEngine;
  }

  public double Add(double n1, double n2)
  {
    double result = calculatorEngine.Add(n1, n2);
    Console.WriteLine("Received Add({0},{1})", n1, n2);
    Console.WriteLine("Return: {0}", result);
    return result;
  }
```

```
  ...
}
```

To modify WCF to use Unity to instantiate the service, you must provide a custom **ServiceHost** class that can pass a Unity container instance into the WCF infrastructure as shown in the following example.

C#

```
public class UnityServiceHost : ServiceHost
{
  public UnityServiceHost(IUnityContainer container,
    Type serviceType, params Uri[] baseAddresses)
    : base(serviceType, baseAddresses)
  {
    if (container == null)
    {
      throw new ArgumentNullException("container");
    }

    foreach (var cd in this.ImplementedContracts.Values)
    {
      cd.Behaviors.Add(new UnityInstanceProvider(container));
    }
  }
```

The following code sample shows the **UnityInstanceProvider** class that resolves the service type (the **CalculatorService** type in this example) from the Unity container.

C#

```
public class UnityInstanceProvider
  : IInstanceProvider, IContractBehavior
{
  private readonly IUnityContainer container;

  public UnityInstanceProvider(IUnityContainer container)
  {
    if (container == null)
    {
      throw new ArgumentNullException("container");
    }

    this.container = container;
  }

  #region IInstanceProvider Members

  public object GetInstance(InstanceContext instanceContext,
    Message message)
  {
```

```
    return this.GetInstance(instanceContext);
  }

  public object GetInstance(InstanceContext instanceContext)
  {
    return this.container.Resolve(
      instanceContext.Host.Description.ServiceType);
  }

  public void ReleaseInstance(InstanceContext instanceContext,
    object instance)
  {
  }

  #endregion

  #region IContractBehavior Members

  public void AddBindingParameters(
    ContractDescription contractDescription,
    ServiceEndpoint endpoint,
    BindingParameterCollection bindingParameters)
  {
  }

  public void ApplyClientBehavior(
    ContractDescription contractDescription,
    ServiceEndpoint endpoint, ClientRuntime clientRuntime)
  {
  }

  public void ApplyDispatchBehavior(
    ContractDescription contractDescription,
    ServiceEndpoint endpoint,
    DispatchRuntime dispatchRuntime)
  {
    dispatchRuntime.InstanceProvider = this;
  }

  public void Validate(
    ContractDescription contractDescription,
    ServiceEndpoint endpoint)
  {
  }

  #endregion
}
```

Now that you have defined the new **UnityServiceHost** class that uses Unity to instantiate your WCF service, you must create an instance of the **UnityServiceHost** class in your run time environment. How you do this for a self-hosted service is different from how you do it for a service hosted in IIS or WAS.

Using the UnityServiceHost Class with a Self-hosted Service

If you are self-hosting the service, you can instantiate the **UnityServiceHost** class directly in your hosting application and pass it a Unity container as shown in the following code sample.

C#

```
class Program
{
  static void Main(string[] args)
  {
    // Register types with Unity
    using (IUnityContainer container = new UnityContainer())
    {
      RegisterTypes(container);

      // Step 1 Create a URI to serve as the base address.
      Uri baseAddress = new Uri("http://localhost:8000/GettingStarted/");

      // Step 2 Create a ServiceHost instance
      ServiceHost selfHost = new UnityServiceHost(container,
        typeof(CalculatorService.CalculatorService), baseAddress);


      try
      {
        // Step 3 Add a service endpoint.
        selfHost.AddServiceEndpoint(typeof(ICalculator), new WSHttpBinding(),
          "CalculatorService");

        // Step 4 Enable metadata exchange.
        ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
        smb.HttpGetEnabled = true;
        selfHost.Description.Behaviors.Add(smb);

        // Step 5 Start the service.
        selfHost.Open();
        Console.WriteLine("The service is ready.");
        Console.WriteLine("Press <ENTER> to terminate service.");
        Console.WriteLine();
        Console.ReadLine();

        // Close the ServiceHostBase to shutdown the service.
        selfHost.Close();
      }
      catch (CommunicationException ce)
```

```
      {
        Console.WriteLine("An exception occurred: {0}", ce.Message);
        selfHost.Abort();
      }
    }
  }

  private static void RegisterTypes(IUnityContainer container)
  {
    container.RegisterType<ICalculationEngine, SimpleEngine>();
  }
}
```

Using the UnityServiceHost Class with Service Hosted in IIS or WAS

If you are hosting your WCF service in IIS or WAS it is a little more complex because you can no longer directly create a service host and by default, IIS will create a **ServiceHost** and not a **UnityServiceHost** instance. To get around this problem, you must create a service host factory as shown in the following code sample.

C#

```
class UnityServiceHostFactory : ServiceHostFactory
{
  private readonly IUnityContainer container;

  public UnityServiceHostFactory()
  {
    container = new UnityContainer();
    RegisterTypes(container);
  }

  protected override ServiceHost CreateServiceHost(
    Type serviceType, Uri[] baseAddresses)
  {
    return new UnityServiceHost(this.container,
      serviceType, baseAddresses);
  }

  private void RegisterTypes(IUnityContainer container)
  {
    container.RegisterType<ICalculationEngine, SimpleEngine>();
  }
}
```

This factory class creates a Unity container instance and passes it in to the constructor of the new **UnityServiceHost** class.

The final step is to instruct IIS or WAS to use the factory to create a service host. You can do this in the .svc file for the service as shown in the following example.

SVC

```
<%@ ServiceHost Language="C#" Debug="true"
Service="CalculatorService.CalculatorService"
Factory="CalculatorService.UnityServiceHostFactory" %>
```

Automatic Factories

Sometimes, your application does not know all the details of the objects to construct until run time. For example, a class called **SurveyAnswerStore** uses one of two queues, depending on whether the tenant is a premium or standard tenant. A simple approach is to use Unity to resolve both queue types as shown in the following sample.

C#

```
public class SurveyAnswerStore : ISurveyAnswerStore
{
  ...
  private readonly IMessageQueue<SurveyAnswerStoredMessage>
    standardSurveyAnswerStoredQueue;
  private readonly IMessageQueue<SurveyAnswerStoredMessage>
    premiumSurveyAnswerStoredQueue;

  public SurveyAnswerStore(
  ITenantStore tenantStore,
  ISurveyAnswerContainerFactory surveyAnswerContainerFactory,
  IMessageQueue<SurveyAnswerStoredMessage> standardSurveyAnswerStoredQueue,
  IMessageQueue<SurveyAnswerStoredMessage> premiumSurveyAnswerStoredQueue,
  IBlobContainer<List<string>> surveyAnswerIdsListContainer)
  {
    ...
    this.standardSurveyAnswerStoredQueue = standardSurveyAnswerStoredQueue;
    this.premiumSurveyAnswerStoredQueue = premiumSurveyAnswerStoredQueue;

  }
  public void SaveSurveyAnswer(SurveyAnswer surveyAnswer)
  {
    var tenant = this.tenantStore.GetTenant(surveyAnswer.Tenant);
    ...
      (SubscriptionKind.Premium.Equals(tenant.SubscriptionKind)
      ? this.premiumSurveyAnswerStoredQueue
      : this.standardSurveyAnswerStoredQueue)
      .AddMessage(new SurveyAnswerStoredMessage
      {
        ...
      });
    }
  }
  ...
}
```

In this example, when the container resolves the **SurveyAnswerStore** type it will inject two **IMessageQueue<SurveyAnswerStoredMessage>** instances. If you know that only one of these instances will be used, you might consider optimizing the solution to create only the instance you need.

One approach is to write a factory class that will instantiate the correct instance, and then take a dependency on the factory. The following code sample shows this approach.

C#

```
class SurveyAnswerStore : ISurveyAnswerStore

{
  ...
  private readonly ISurveyAnswerQueueFactory surveyAnswerQueueFactory;

  public SurveyAnswerStore(
  ITenantStore tenantStore,
  ISurveyAnswerContainerFactory surveyAnswerContainerFactory,
  ISurveyAnswerQueueFactory surveyAnswerQueueFactory,
  IBlobContainer<List<string>> surveyAnswerIdsListContainer)
  {
    ...
    this.surveyAnswerQueueFactory = surveyAnswerQueueFactory;
  }

  public void SaveSurveyAnswer(SurveyAnswer surveyAnswer)
  {
    var tenant = this.tenantStore.GetTenant(surveyAnswer.Tenant);
    ...

      ((tenant.SubscriptionKind == "Premium")
      ? this.surveyAnswerQueueFactory.GetPremiumQueue()
      : this.surveyAnswerQueueFactory.GetStandardQueue())
      .AddMessage(new SurveyAnswerStoredMessage
      {
        ...
      });
    }
  }
  ...
}
```

For this approach to work, in addition to writing the factory class, you must register the factory class with the container so that the container can inject it when it resolves the **SurveyAnswerStore** type.

A further refinement is to use Unity's automatic factory approach. Using this approach you do not need to write and register a factory class, Unity creates a lightweight factory and registers it on your behalf. The following code sample shows this approach.

C#

```
class SurveyAnswerStore : ISurveyAnswerStore

{
  ...
  private readonly Func<IMessageQueue<SurveyAnswerStoredMessage>>
    standardSurveyAnswerQueueFactory;
  private readonly Func<IMessageQueue<SurveyAnswerStoredMessage>>
    premiumSurveyAnswerQueueFactory;


  public SurveyAnswerStore(
  ITenantStore tenantStore,
  ISurveyAnswerContainerFactory surveyAnswerContainerFactory,
  [Dependency("Standard")]Func<IMessageQueue<SurveyAnswerStoredMessage>>
    standardSurveyAnswerQueueFactory,
  [Dependency("Premium")]Func<IMessageQueue<SurveyAnswerStoredMessage>>
    premiumSurveyAnswerQueueFactory,
  IBlobContainer<List<string>> surveyAnswerIdsListContainer)
  {
    ...
    this.standardSurveyAnswerQueueFactory = standardSurveyAnswerQueueFactory;
    this.premiumSurveyAnswerQueueFactory = premiumSurveyAnswerQueueFactory;
  }
  public void SaveSurveyAnswer(SurveyAnswer surveyAnswer)
  {
    var tenant = this.tenantStore.GetTenant(surveyAnswer.Tenant);
    ...

      ((tenant.SubscriptionKind == "Premium")
      ? premiumSurveyAnswerQueueFactory()
      : standardSurveyAnswerQueueFactory())
      .AddMessage(new SurveyAnswerStoredMessage
      {
        ...
      });
    }
  }
  ...
}
```

In this example, the dependencies of the **SurveyAnswerStore** class are on values of the form **Func<T>**. This enables the container to generate delegates that perform the type resolution when they are invoked: in the sample code, the delegates are called **premiumSurveyAnswerQueueFactory** and **standardSurveyAnswerQueueFactory**.


You don't need to change the registrations in the container to make this approach work.

One drawback of this specific example is that because the two queues use named registrations in the container, you must use the **Dependency** attribute to specify which named registration to resolve. This means that the **SurveyAnswerStore** class has a dependency on Unity.

Deferred Resolution

Sometimes, you may want to resolve an object from the container, but defer the creation of the object until you need to use it. You can achieve this with Unity by using the **Lazy<T>** type from the .NET Framework; this type provides support for the lazy initialization of objects.


Lazy<T> doesn't work very well with value types, and it is better to avoid in this case. You should use the **Lazy<T>** type very cautiously.

To use this approach with Unity, you can register the type you want to use in the standard way, and then use the **Lazy<T>** type when you resolve it. The following code sample shows this approach.

C#

```
// Register the type
container.RegisterType<MySampleObject>(new InjectionConstructor("default"));

// Resolve using Lazy<T>
var defaultLazy = container.Resolve<Lazy<MySampleObject>>();

// Use the resolved object
var mySampleObject = defaultLazy.Value;
```

This example is adapted from the sample application, "OtherUnitySamples," included with this guidance.

You can use lazy resolution with the Unity lifetime managers. The following example, again adapted from the sample application illustrates this with the **ContainerManagedLifetime** class.

C#

```
// Register the type with a lifetime manager
container.RegisterType<MySampleObject>(
  "other", new ContainerControlledLifetimeManager(),
  new InjectionConstructor("other"));

// Resolve the lazy type
var defaultLazy1 = container.Resolve<Lazy<MySampleObject>>("other");

// Resolve the lazy type a second time
var defaultLazy2 = container.Resolve<Lazy<MySampleObject>>("other");
```

```
// defaultLazy1 == defaultLazy2 is false
// defaultLazy1.Value == defaultLazy2.Value is true
```

For more information about **Lazy<T>**, see the topic Lazy<T> Class on MSDN.

You can also use the **Resolve** method to resolve registered types by using **Func<T>** in a similar way.

Lifetime Management

When you resolve an object that you registered using the **RegisterType** method, the container instantiates a new object when you call the **Resolve** method: the container does not hold a reference to the object. When you create a new instance using the **RegisterInstance** method, the container manages the object and holds a reference to it for the lifetime of the container.

Lifetime Managers manage the lifetimes of objects instantiated by the container. The default lifetime manager for the **RegisterType** method is the **TransientLifetimeManager** and the default lifetime manager for the **RegisterInstance** method is the **ContainerControlledLifetimeManager**. If you want the container to create or return a singleton instance of a type when you call the **Resolve** method, you can use the **ContainerControlledLifetimeManager** type when you register your type or instance. The following example shows how you could tell the container to create a singleton instance of the **TenantStore**.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new ContainerControlledLifetimeManager());
```

The first time that you resolve the **ITenantStore** type the container creates a new **TenantStore** object and keeps a reference to it. On subsequent times when you resolve the **ITenantStore** type, the container returns a reference to the **TenantStore** object that it created previously. Some lifetime managers, such as the **ContainerControlledLifetimeManager**, are used to dispose the created objects when the container is disposed.

Lifetime managers enable you to control for how long the objects created by the container should live in your application. You can override the default lifetime managers that the **RegisterType** and **RegisterInstance** methods use.

Unity includes five other lifetime managers, described in the following sections, that you can use to address specific scenarios in your applications.

Hierarchical Lifetime Management

This type of lifetime management is useful if you have a hierarchy of containers. Earlier in this chapter, you saw how to use child containers to manage alternative mappings for the same type. You can also use child containers to manage the lifetime of resolved objects. Figure 2 illustrates a scenario where you

have created two child containers and registered a type using the **ContainerControlledLifetimeManager** type to create a singleton.

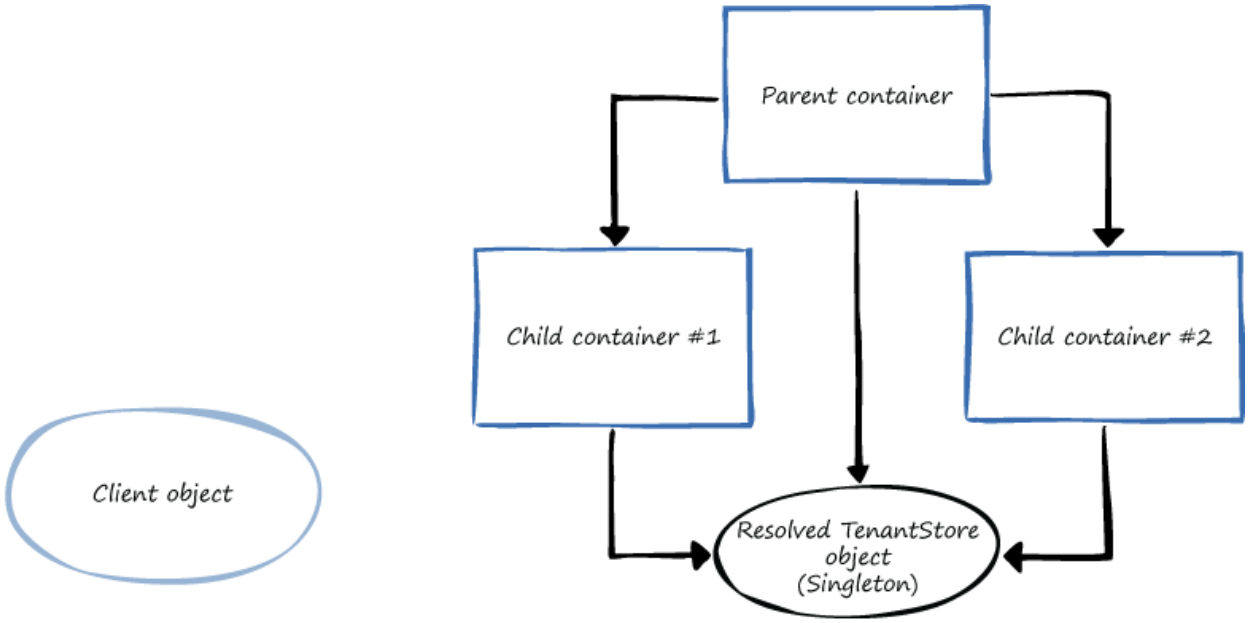

Figure 2 - Container hierarchy with ContainerControlledLifetimeManager lifetime manager

If the client object executes the following code that creates the containers, performs the registrations, and then resolves the types, the three variables (**tenant1**, **tenant2**, and **tenant3**) all refer to the same instance managed by the containers.

C#

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ITenantStore, TenantStore>(
  new ContainerControlledLifetimeManager());
IUnityContainer child1 = container.CreateChildContainer();
IUnityContainer child2 = container.CreateChildContainer();

var tenant1 = child1.Resolve<ITenantStore>();
var tenant2 = child2.Resolve<ITenantStore>();
var tenant3 = container.Resolve<ITenantStore>();
```

However, if you use the **HierarchicalLifetimeManager** type, the container resolves the object as shown in Figure 3.

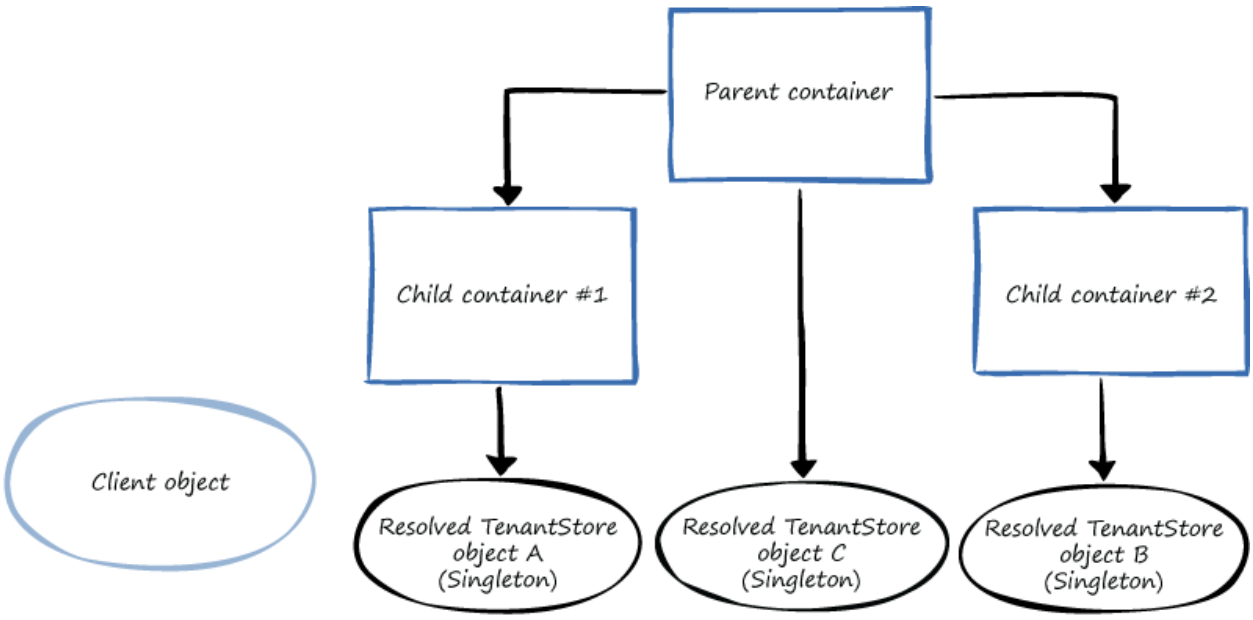

Figure 3 - Container hierarchy with HierarchicalLifetimeManager lifetime manager

If the client executes the following code, the three variables (**tenant1**, **tenant2**, and **tenant3**) each refer to different **TenantStore** instances.

C#

```
IUnityContainer container = new UnityContainer();
container.RegisterType<ITenantStore, TenantStore>(
  new HierarchyLifetimeManager());
IUnityContainer child1 = container.CreateChildContainer();
IUnityContainer child2 = container.CreateChildContainer();
var tenant1 = child1.Resolve<ITenantStore>();
var tenant2 = child2.Resolve<ITenantStore>();
var tenant3 = container.Resolve<ITenantStore>();
```

Although you register the type with the parent container, each child container now resolves its own instance. Each child container manages its own singleton instance of the **TenantStore** type; therefore, if you resolve the same type from container #1 a second time, the container returns a reference to the instance it created previously.

This approach is useful in web applications where you want to register your types once, but then resolve separate instances for each client session. This assumes that it is possible, with your design, to map one child container to each session.

Per Resolve Lifetime Management

Figure 4 shows part of the dependency tree for an application: the **SurveysController** type depends on the **SurveyStore** and **SurveyAnswerStore** types, both the **SurveyStore** and **SurveyAnswerStore** types depend on the **TenantStore** type.

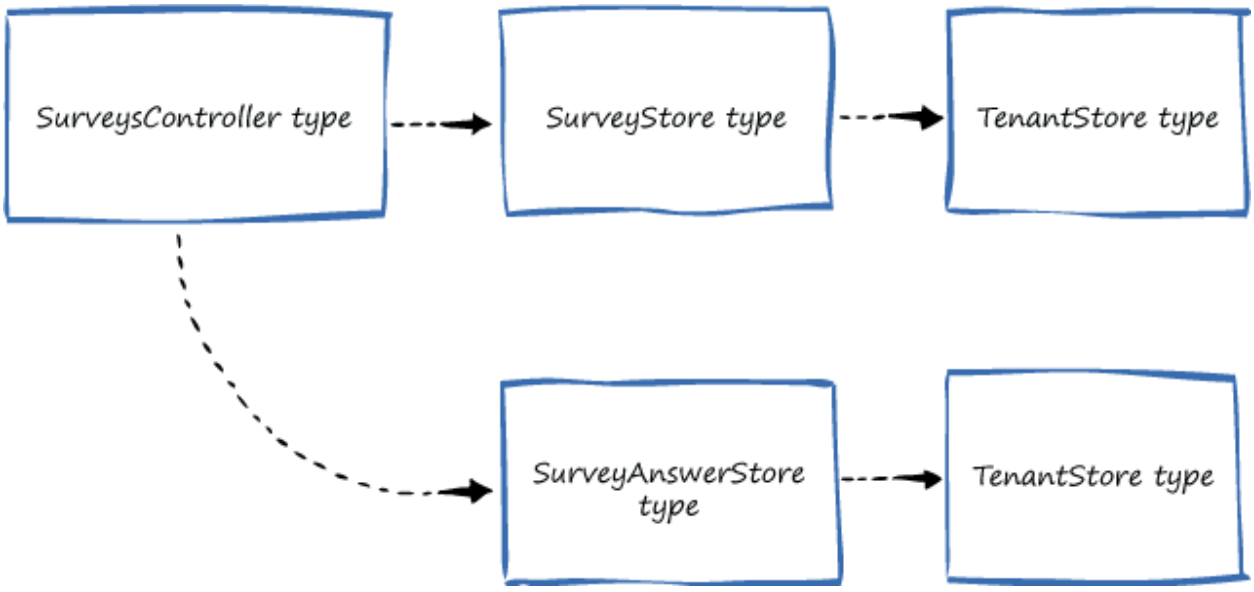

Figure 4 - Sample dependency tree

If you use the default **TransientLifetimeManager** class when you register the **SurveysController** type, then when you resolve the **SurveysController** type, the container builds the object graph shown in Figure 5.

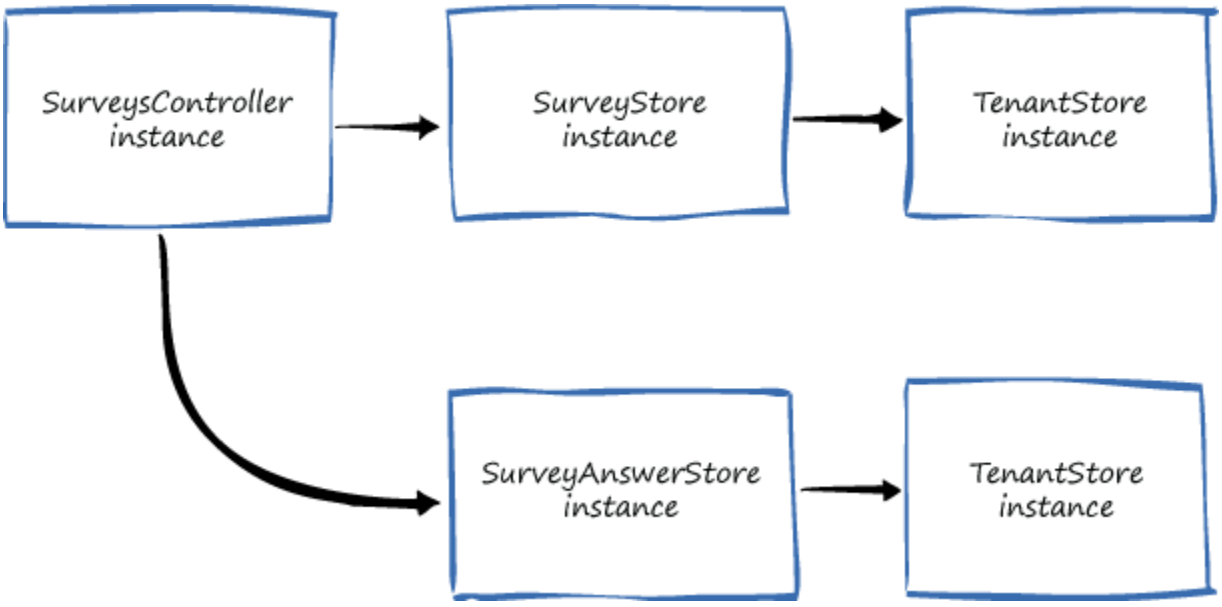

Figure 5 - Object graph generated using TransientLifetimeManager lifetime manager

However, if you use the **PerResolveLifetimeManager** class in place of the **TransientLifetimeManager** class, then the container builds the object graph shown in Figure 6. With the **PerResolveLifetimeManager** class, the container reuses any instances it resolves during a call to the **Resolve** method in any other types it resolves during the same call.

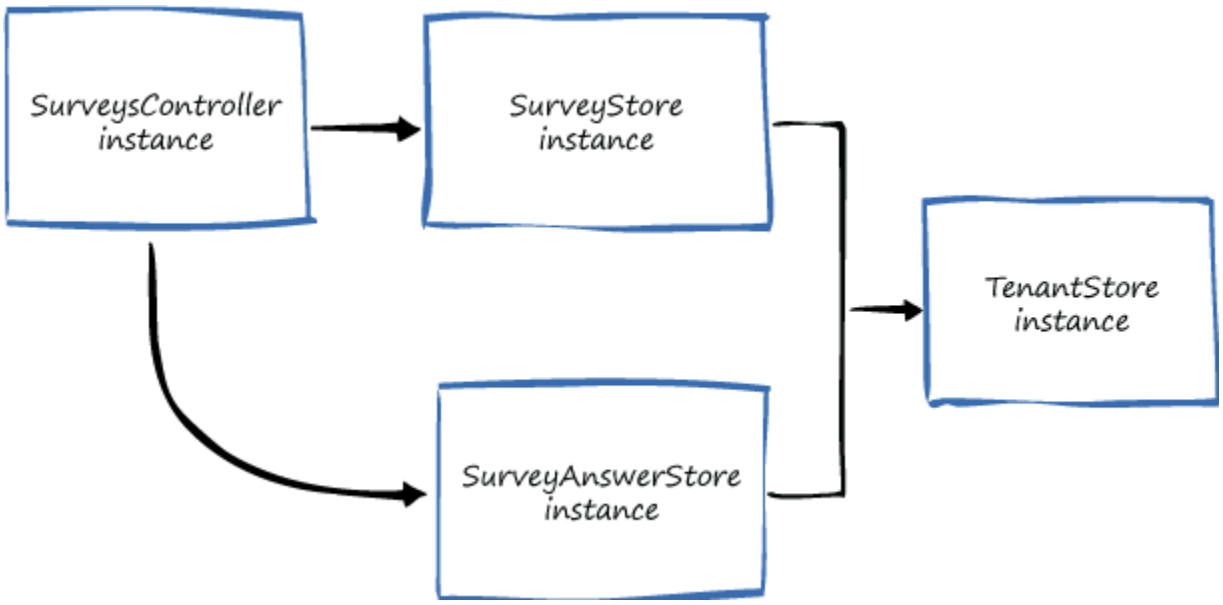

Figure 6 - Object graph generated using the PerResolveLifetimeManager class

Externally Controlled Lifetime Management

If you resolve a type that was registered using the **ContainerControlledLifetimeManager** class, the container creates a singleton instance and holds a strong reference to it: this means that the instance lives at least as long as the container. However, if you use the **ExternallyControlledLifetimeManager** class, when you resolve the type, the container creates a singleton instance but holds only a weak reference to it. In this case, you can directly manage the lifetime of the object: because of the weak reference, you can dispose of the object when you no longer need it. This enables you to inject objects that are not owned by the container; for example you might need to inject objects whose lifetime is managed by ASP.NET into instances created by the container. Also, the **ExternallyControlledLifetimeManager** class does not dispose the instances it holds references to when the container is disposed.

Per Request Lifetime Management

This lifetime manager is only available for use in Web applications when you've added the "Unity bootstrapper for ASP.NET MVC" NuGet package to your project. The **PerRequestLifetimeManager** class enables the container to create new instances of registered types for each HTTP request in an ASP.NET MVC application or an ASP.NET Web API application. Each call to **Resolve** a type within the context of a single HTTP request will return the same instance: in effect, the Unity container creates singletons for registered types for the duration of the HTTP request.

Although the **PerRequestLifetimeManager** class works correctly and can help you to work with stateful or thread-unsafe dependencies within the scope of an HTTP request, it is generally not a good idea to use it if you can avoid it. Using this lifetime manager can lead to bad practices or hard to find bugs in the end user's application code when used incorrectly. The dependencies you register with the Unity container should be stateless, and if you have a requirement to share common state between several objects during the lifetime of an HTTP request, then you can have a stateless service that explicitly stores and retrieves this state using the **System.Web.HttpContext.Items** collection of the **System.Web.HttpContext.Current** object.


Think carefully about the implications for state management in your application if you plan to use the **PerRequestLifetimeManager** lifetime manager class.

Per Thread Lifetime Management

The final lifetime manger included with Unity enables you to resolve instances on a per thread basis. All calls to the **Resolve** method from the same thread return the same instance.

For more information about Unity's lifetime managers, see the topic Understanding Lifetime Managers.

Dependency Injection and Unit Testing

In Chapter 1, one of the motivations for adopting a loosely coupled design was that it facilitates unit testing. In the example used in this chapter, one of the types registered with the container is the **TenantStore** class. The following code sample shows an outline of this class.

C#

```
public class TenantStore : ITenantStore
{
  ...

  public TenantStore(IBlobContainer<Tenant> tenantBlobContainer,
    IBlobContainer<byte[]> logosBlobContainer)
  {
    ...
  }


  public Tenant GetTenant(string tenant)
  {
    ...
  }

  public IEnumerable<string> GetTenantNames()
  {
    ...
```

```
  }

  public void SaveTenant(Tenant tenant)
  {
    ...
  }

  public void UploadLogo(string tenant, byte[] logo)
  {
    ...
  }
}
```

This class has dependencies on the **IBlobContainer<Tenant>** and **IBlobContainer<byte[]>** types which the container resolves when it instantiates a **TenantStore** object. However, to test this class in a unit test, you don't want to have to create these blob containers: now it's easy to replace them with mocks for the purpose of the tests. The following code sample shows some example tests.

C#

```
[TestMethod]
public void GetTenantReturnsTenantFromBlobStorage()
{
  var mockTenantBlobContainer = new Mock<IBlobContainer<Tenant>>();
  var store = new TenantStore(mockTenantBlobContainer.Object, null);
  var tenant = new Tenant();
  mockTenantBlobContainer.Setup(c => c.Get("tenant")).Returns(tenant);

  var actualTenant = store.GetTenant("tenant");

  Assert.AreSame(tenant, actualTenant);
}

[TestMethod]
public void UploadLogoGetsTenantToUpdateFromContainer()
{
  var mockLogosBlobContainer = new Mock<IBlobContainer<byte[]>>();
  var mockTenantContainer = new Mock<IBlobContainer<Tenant>>();
  var store = new TenantStore(
    mockTenantContainer.Object, mockLogosBlobContainer.Object);
  mockTenantContainer.Setup(c => c.Get("tenant"))
    .Returns(new Tenant() { Name = "tenant" }).Verifiable();
  mockLogosBlobContainer.Setup(
    c => c.GetUri(It.IsAny<string>())).Returns(new Uri("http://bloburi"));

  store.UploadLogo("tenant", new byte[1]);

  mockTenantContainer.Verify();
}
```

These two example tests provide mock objects that implement the **IBlobContainer<Tenant>** and **IBlobContainer<byte[]>** interfaces when they create the **TenantStore** instances to test.

> These examples use the Moq mocking library to create the mock objects. For more information, see http://code.google.com/p/moq/. Moq is also available as a NuGet package.

Summary

In this chapter, you saw how to use the Unity container to add support for dependency injection to a real-world application and how you can use a Unity container to register types, resolve types at runtime, and manage the lifetime of the resolved objects. In addition to seeing how the Unity container made it possible to build the application's object graph at startup, you also saw how this approach facilitated designing and running unit tests.

Chapter 4 - Interception

Introduction

In Chapter 1, you learned about some of the motivations for adopting a loosely coupled design, and in the following chapters, you saw how dependency injection (DI) and the Unity container can help you to realize these benefits in your own applications. One of the motivations that Chapter 1 described for adopting a loosely coupled design was crosscutting concerns. Although DI enables you to inject dependencies when you instantiate the objects that your application will use at run time and helps you to ensure that the objects that Unity instantiates on your behalf address your crosscutting concerns, you will still want to try and follow the single responsibility and open/closed principles in your design. For example, the classes that implement your business behaviors should not also be responsible for logging or validation, and you should not need to modify your existing classes when you add support for a new crosscutting concern to your application. Interception will help you to separate out the logic that handles cross-cutting concerns from your LOB classes.

Crosscutting Concerns

Crosscutting concerns are concerns that affect many areas of your application. For example, in a LOB application, you may have a requirement to write information to a log file from many different areas in your application. The term crosscutting concerns, refers to the fact that these types of concern typically cut across your application and do not align with the modules, inheritance hierarchies, and namespaces that help you to structure your application in ways that align with your application's business domain. Common crosscutting concerns for LOB applications include:

- **Logging**. Writing diagnostic messages to a log for troubleshooting, tracing, or auditing purposes.
- **Validation**. Checking that the input from users or other systems complies with a set of rules.
- **Exception handling**. Using a common approach to exception handling in the application.
- **Transient fault handling**. Using a common approach to identifying transient faults and retrying operations.
- **Authentication and authorization**. Identifying the caller and determining if that caller should be allowed to perform an operation.
- **Caching**. Caching frequently used objects and resources to improve performance.
- **Performance monitoring**. Collecting performance data in order to measure SLA compliance.
- **Encryption**. Using a common service to encrypt and decrypt messages within the application.
- **Mapping**. Providing a mapping or translation service for data as it moves between classes or components.
- **Compression**. Providing a service to compress data as it moves between classes or components.

It's possible that many different classes and components within your application will need to implement some of these behaviors (Unity often uses the term *behaviors* to refer to the logic that implements cross-cutting concerns in your code). However, implementing support for these crosscutting concerns in a LOB application introduces a number of challenges such as how you can:

- **Maintain consistency**. You want to be sure that all the classes and components that need to implement one of these behaviors do so in a consistent way. Also, if you need to modify the way that your application supports one of these crosscutting concerns, then you want to be sure that the change is applied everywhere.
- **Create maintainable code**. The single responsibility principle helps to make your code more maintainable. A class that implements a piece of business functionality should not also be responsible for implementing a crosscutting concern such as logging.
- **Avoid duplicate code**. You don't want to have the same code duplicated in multiple locations within your application.

Before examining how interception and Unity's implementation of interception can help you to address cross-cutting concerns in your applications, it's worth examining some alternative approaches. The decorator pattern offers an approach that doesn't require a container and that you can implement yourself without any dependencies on special frameworks or class libraries. Aspect oriented programming (AOP) is another approach that adopts a different paradigm for addressing cross-cutting concerns.

The Decorator Pattern

A common approach to implementing behaviors to address crosscutting concerns in your application is to use the decorator pattern. The basis of the decorator pattern is to create wrappers that handle the crosscutting concerns around your existing objects. In previous chapters, you saw the **TenantStore** class that is responsible for retrieving tenant information from, and saving tenant information to a data store. You will now see how you can use the decorator pattern to add logic to handle crosscutting concerns such as logging and caching without modifying or extending the existing **TenantStore** class.


If you extended the **TenantStore** class to add support for logging you would be adding an additional responsibility to an existing class, breaking the single responsibility principle.

The following code sample shows the existing **TenantStore** class and **ITenantStore** interface.

C#

```
public interface ITenantStore
{
  void Initialize();
  Tenant GetTenant(string tenant);
```

```
  IEnumerable<string> GetTenantNames();
  void SaveTenant(Tenant tenant);
  void UploadLogo(string tenant, byte[] logo);
}

public class TenantStore : ITenantStore
{
  ...

  public TenantStore(IBlobContainer<Tenant> tenantBlobContainer,
    IBlobContainer<byte[]> logosBlobContainer)
  {
    ...
  }

  ...
}
```

The following code sample shows a decorator class that adds logging functionality to the existing **TenantStore** class:

C#

```
class LoggingTenantStore : ITenantStore
{
  private readonly ITenantStore tenantStore;
  private readonly ILogger logger;
  public LoggingTenantStore(ITenantStore tenantstore, ILogger logger)
  {
    this.tenantStore = tenantstore;
    this.logger = logger;
  }
  public void Initialize()
  {
    tenantStore.Initialize();
  }

  public Surveys.Models.Tenant GetTenant(string tenant)
  {
    return tenantStore.GetTenant(tenant);
  }

  public IEnumerable<string> GetTenantNames()
  {
    return tenantStore.GetTenantNames();
  }

  public void SaveTenant(Surveys.Models.Tenant tenant)
  {
    tenantStore.SaveTenant(tenant);
```

```
    logger.WriteLogMessage("Saved tenant" + tenant.Name);
  }

  public void UploadLogo(string tenant, byte[] logo)
  {
    tenantStore.UploadLogo(tenant, logo);
    logger.WriteLogMessage("Uploaded logo for " + tenant);
  }
}
```

Note how this decorator class also implements the **ITenantStore** interface and accepts an **ITenantStore** instance as a parameter to the constructor. In each method body, it invokes the original method before it performs any necessary work related to logging. You could also reverse this order and perform the work that relates to the crosscutting concern before you invoke the original method. You could also perform work related to the crosscutting concern both before and after invoking the original method.

If you had another decorator class called **CachingTenantStore** that added caching behavior you could create an **ITenantStore** instance that also handles logging and caching using the following code.

C#

```
var basicTenantStore = new TenantStore(tenantContainer, blobContainer);
var loggingTenantStore = new LoggingTenantStore(basicTenantStore, logger);
var cachingAndLoggingTenantStore = new CachingTenantStore(loggingTenantStore, cache);
```

If you invoke the **UploadLogo** method on the **cachingAndLoggingTenantStore** object, then you will first invoke the **UploadLogo** method in the **CachingTenantStore** class that will in turn invoke the **UploadLogo** method in the **LoggingTenantStore** class, which will in turn invoke the original **UploadLogo** method in the **TenantStore** class before returning through the sequence of decorators.

Figure 1 shows the relationships between the various objects at run time after you have instantiated the classes. You can see how each **UploadLogo** method performs its crosscutting concern functionality before it invokes the next decorator in the chain, until it gets to the end of the chain and invokes the original **UploadLogo** method on the **TenantStore** instance.

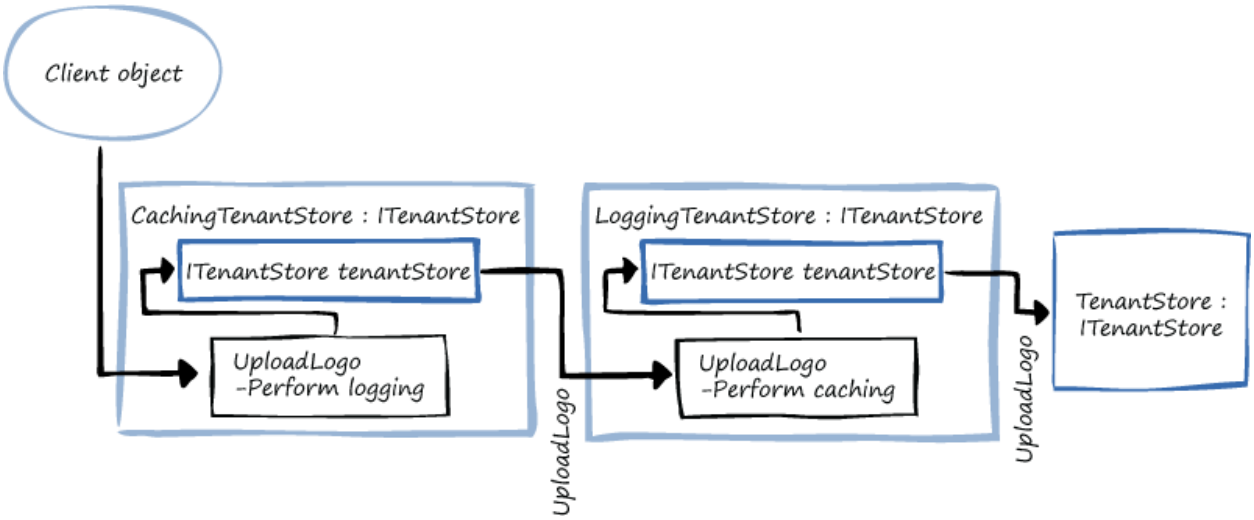

Figure 1 - The decorator pattern at run time

Using Unity to Wire Up Decorator Chains

Instead of wiring the decorators up manually, you could use the Unity container to do it for you. The following set of registrations will result in the same chain of decorators shown in figure 1 when you resolve the default **ITenantStore** type.

C#

```
container.RegisterType<ILogger, Logger>();
container.RegisterType<ICacheManager, SimpleCache>();

container.RegisterType<ITenantStore, TenantStore>(
  "BasicStore");

container.RegisterType<ITenantStore, LoggingTenantStore>(
  "LoggingStore",
  new InjectionConstructor(
    new ResolvedParameter<ITenantStore>("BasicStore"),
    new ResolvedParameter<ILogger>()));

// Default registration
container.RegisterType<ITenantStore, CachingTenantStore>(
  new InjectionConstructor(
    new ResolvedParameter<ITenantStore>("LoggingStore"),
    new ResolvedParameter<ICacheManager>()));
```

Aspect Oriented Programming

Aspect Oriented Programming (AOP) is closely related to the issue of handling crosscutting concerns in your application. In AOP, you have some classes in your application that handle the core business logic and other classes that handle aspects or crosscutting concerns. In the example shown in the previous section, the **TenantStore** class is responsible for some of the business logic in your LOB application, and the classes that implement the **ILogger** and **ICacheManager** interfaces are responsible for handling the aspects (or crosscutting concerns) in the application.

The previous example shows how you can use the decorator pattern to explicitly wire-up the classes responsible for the crosscutting concerns. While this approach works, it does require you write the decorator classes (**CachingTenantStore** and **LoggingTenantStore**) in addition to wiring everything together, either explicitly or using DI.

AOP is a mechanism that is intended to simplify how you wire-up the classes that are responsible for the crosscutting concerns to the business classes: there should be no need to write the decorator classes and you should be able to easily attach the aspects that handle the crosscutting concerns to your standard classes. The way that AOP frameworks are implemented is often technology dependent because it requires a dynamic mechanism to wire-up the aspect classes to the business classes after they have all been compiled.

For an example of an AOP framework for C#, see "What is PostSharp?" AspectJ is a widely used AOP framework for Java.

Interception

Interception is one approach to implementing the dynamic wire up that is necessary for AOP. You can use interception as a mechanism in its own right to insert code dynamically without necessarily adopting AOP, but interception is often used as the underlying mechanism in AOP approaches.


The interception approach that Unity adopts is not, strictly speaking, an AOP approach, although it does have many similarities with true AOP approaches. For example, Unity interception only supports preprocessing and post-processing behaviors around method calls and does not insert code into the methods of the target object. Also, Unity interception does not support interception for class constructors.

Interception works by dynamically inserting code (typically code that is responsible for crosscutting concerns) between the calling code and the target object. You can insert code before a method call so that it has access to the parameters being passed, or afterwards so that it has access to the return value or unhandled exceptions. This inserted code typically implements what are known as behaviors (behaviors typically implement support for cross-cutting concerns), and you can insert multiple behaviors into a pipeline between the client object and the target object in a similar way to using a chain of decorators in the decorator pattern to add support for multiple crosscutting concerns. The key difference between interception and the decorator pattern is that the interception framework dynamically creates decorator classes such as **LoggingTenantStore** and **CachingTenantStore** at run time. This makes it much easier to add behaviors that provide support for crosscutting concerns or aspects because you no longer need to manually create decorator classes for every business class that needs to support the behaviors. Now, you can use a configuration mechanism to associate the classes that implement the behaviors with the business classes that need the behaviors.

For a discussion of AOP and interception with Unity, see the article "Aspect-Oriented Programming, Interception and Unity 2.0" by Dino Esposito on MSDN.

There are many ways that you can implement interception, but the two approaches that Unity supports are known as *instance interception* and *type interception*. The next two sections describe these two different approaches.

Instance Interception

With instance interception, Unity dynamically creates a proxy object that it inserts between the client and the target object. The proxy object is responsible for passing the calls made by the client to the target object through the behaviors that are responsible for the crosscutting concerns.

Figure 2 shows how when you use instance interception, the Unity container instantiates the target **TenantStore** object, instantiates the pipeline of behaviors that implement the crosscutting concerns, and generates and instantiates a **TenantStore** proxy object.

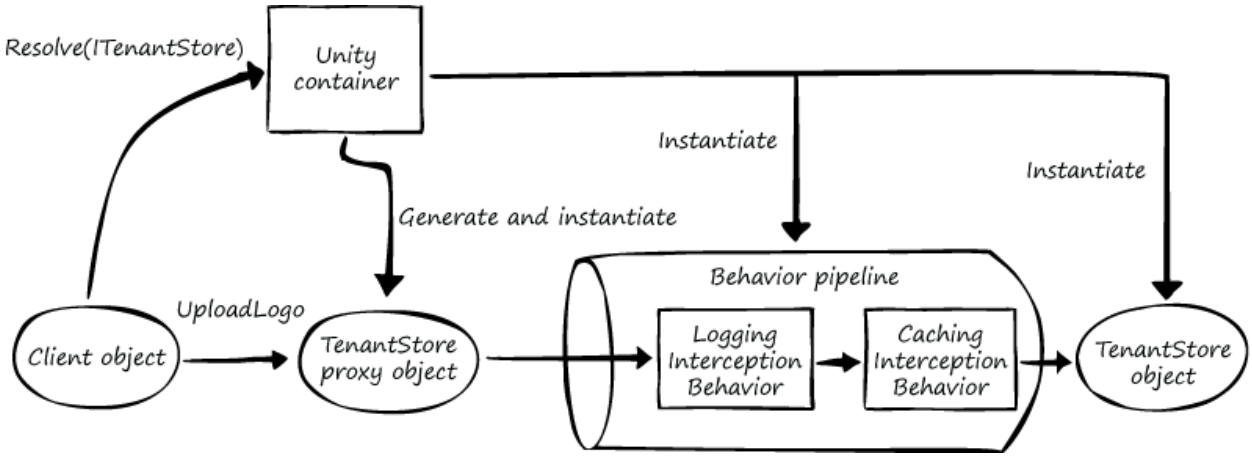

Figure 2 - An example of instance interception

Instance interception is the more commonly used of the two interception techniques supported by Unity and you can use it if the target object either implements the **MarshalByRefObject** abstract class or implements a public interface that defines the methods that you need to intercept.

You can use Unity instance interception to intercept objects created both by the Unity container and outside of the container, and you can use instance interception to intercept both virtual and non-virtual methods. However, you cannot cast the dynamically created proxy type to the type of the target object.

Type Interception

With type interception, Unity dynamically creates a new type that derives from the type of the target object and that includes the behaviors that handle the crosscutting concerns. The Unity container instantiates objects of the derived type at run time.

Figure 3 shows how with type interception, the Unity container generates and instantiates an object of a type derived from the target **TenantStore** type that encapsulates the behavior pipeline in the overridden method calls.

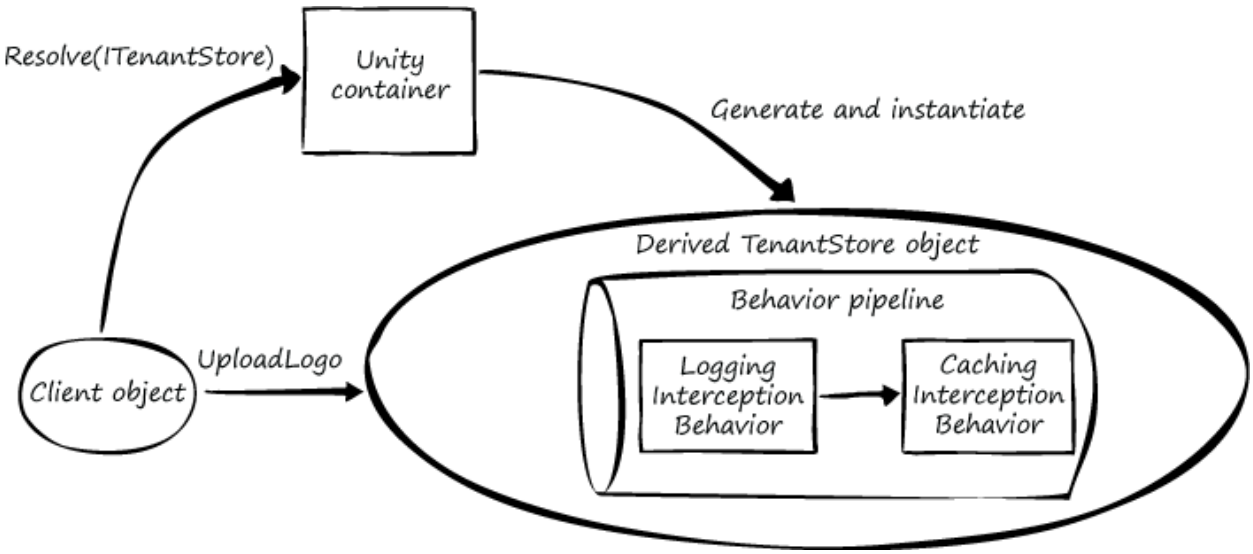

Figure 3 - An example of type interception

You can only use Unity type interception to intercept virtual methods. However, because the new type derives from the original target type, you can use it wherever you used the original target type.

For more information about the differences between these approaches, see Unity Interception Techniques.

Summary

In this chapter, you learned how interception enables you to add support for crosscutting concerns to your application by intercepting calls to your business objects at run time. Interception uses dynamically created classes to implement the decorator pattern at run time. The next chapter describes in detail how you can implement interception using the Unity container.

Chapter 5 - Interception using Unity

Introduction

Chapter 4 describes interception as a technique that you can use to dynamically insert code that provides support for crosscutting concerns into your application without explicitly using the decorator pattern in your code. In this chapter, you'll learn how you can use Unity to implement interception, and the various types of interception that Unity supports. The chapter starts by describing a common scenario for using interception and illustrates how you can implement it using Unity interception. The chapter then explores a number of alternative approaches that you could adopt, including the use of policy injection and attributes.

> You cannot use Unity Interception in Windows Store Applications. For more information, see Appendix A – Unity and Windows Store apps.

Crosscutting Concerns and Enterprise Library

This chapter begins by describing how you can use Unity interception to implement your own custom code to add support for crosscutting concerns into your application. However, the blocks in Enterprise Library provide rich support for crosscutting concerns, and therefore it's not surprising that instead of your own custom code you can use the Enterprise Library blocks with Unity interception; the chapter also describes this approach.

>  You can use policy injection and call handlers to integrate the functionality provided by the Enterprise Library blocks with Unity interception.

Interceptors in Unity

This section describes the basic steps that you'll need to complete in order to use interception in your application. To illustrate some of the advantages of the interception approach, this chapter uses the same example crosscutting concerns, logging and caching, that the discussion of the decorator pattern in Chapter 4 uses. You may find it useful to refer back to Chapter 4, "Interception," for some of the details that relate to the logging and caching functionality.

> This section uses logging as an example of a crosscutting concern and shows how you can implement it using interception. Later in this chapter, you'll see how you can use the Enterprise Library logging block in place of your own implementation.

Configuring the Unity Container to Support Interception

By default, a Unity container does not support interception; to add support for interception you must add the Unity interception extension as shown in the following code sample. You can add the Unity Interception Extension libraries to your project using NuGet.

C#

```
using Microsoft.Practices.Unity.InterceptionExtension;

...

IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
```

If you don't need to use Unity interception, you don't need to install the libraries through NuGet: interception is an extension to the core Unity installation.

For more information about how to add the interception extension to a Unity container instance and about how you can use a configuration file instead of code, see the topic Configuring a Container for Interception.

Defining an Interceptor

In Chapter 4, the two example crosscutting concerns in the decorator example were logging and caching. The example showed implementations of these behaviors in classes that implemented the **ILogger** and **ICacheManager** interfaces. To implement these behaviors as interceptors, you need classes that implement the **IInterceptionBehavior** interface. The following example shows how you can implement the logging behavior.

C#

```
using Microsoft.Practices.Unity.InterceptionExtension;

class LoggingInterceptionBehavior : IInterceptionBehavior
{
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextInterceptionBehaviorDelegate getNext)
  {
    // Before invoking the method on the original target.
    WriteLog(String.Format(
      "Invoking method {0} at {1}",
      input.MethodBase, DateTime.Now.ToLongTimeString()));

    // Invoke the next behavior in the chain.
    var result = getNext()(input, getNext);

    // After invoking the method on the original target.
```

```
    if (result.Exception != null)
    {
      WriteLog(String.Format(
        "Method {0} threw exception {1} at {2}",
        input.MethodBase, result.Exception.Message,
        DateTime.Now.ToLongTimeString()));
    }
    else
    {
      WriteLog(String.Format(
        "Method {0} returned {1} at {2}",
        input.MethodBase, result.ReturnValue,
        DateTime.Now.ToLongTimeString()));
    }

    return result;
  }

  public IEnumerable<Type> GetRequiredInterfaces()
  {
    return Type.EmptyTypes;
  }

  public bool WillExecute
  {
    get { return true; }
  }

  private void WriteLog(string message)
  {
    ...
  }
}
```

The **IInterceptionBehavior** interface defines three methods: **WillExecute**, **GetRequiredInterfaces**, and **Invoke**. In many scenarios, you can use the default implementations of the **WillExecute** and **GetRequiredInterfaces** methods shown in this example. The **WillExecute** method enables you to optimize your chain of behaviors by specifying whether this behavior should execute; in this example the method always returns **true** so the behavior always executes. The **GetRequiredInterfaces** method enables you to specify the interface types that you want to associate with the behavior. In this example, the interceptor registration will specify the interface type, and therefore the **GetRequiredInterfaces** method returns **Type.EmptyTypes**. For more information about how to use these two methods, see the topic Behaviors for Interception.

The **Invoke** method takes two parameters: **input** contains information about the call from the client that includes the method name and parameter values, **getNext** is a delegate that enables you to call the next behavior in the pipeline, or the target object if this is the last behavior in the pipeline. In this example,

you can see how the behavior captures the name of the method invoked by the client and records details of the method invocation in the log before it invokes the next behavior in the pipeline, or target object itself. This behavior then examines the result of the call to the next behavior in the pipeline and writes a log message if an exception was thrown in the call, or details of the result if the call succeeded. Finally, it returns the result of the call back to the previous behavior in the pipeline (or the client object if this behavior was first in the pipeline).

 In the **Invoke** method, you can apply pre and post processing to the call to the target object by placing your code before and after the call to the **getNext** delegate.

The following code example shows a slightly more complex behavior that provides support for caching.

C#

```
using Microsoft.Practices.Unity.InterceptionExtension;

class CachingInterceptionBehavior : IInterceptionBehavior
{
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextInterceptionBehaviorDelegate getNext)
  {

    // Before invoking the method on the original target.
    if (input.MethodBase.Name == "GetTenant")
    {
      var tenantName = input.Arguments["tenant"].ToString();
      if (IsInCache(tenantName))
      {
        return input.CreateMethodReturn(
          FetchFromCache(tenantName));
      }
    }

    IMethodReturn result = getNext()(input, getNext);

    // After invoking the method on the original target.
    if (input.MethodBase.Name == "SaveTenant")
    {
      AddToCache(input.Arguments["tenant"]);
    }

    return result;

  }

  public IEnumerable<Type> GetRequiredInterfaces()
```

```
  {
    return Type.EmptyTypes;
  }

  public bool WillExecute
  {
    get { return true; }
  }

  private bool IsInCache(string key) {...}

  private object FetchFromCache(string key) {...}

  private void AddToCache(object item) {...}
}
```

This behavior filters for calls to a method called **GetTenant** and then attempts to retrieve the named tenant from the cache. If it finds the tenant in the cache, it does not need to invoke the target object to get the tenant, and instead uses the **CreateMethodReturn** to return the tenant from the cache to the previous behavior in the pipeline.

Behaviors do not always have to pass on a call to the next behavior in the pipeline. They can generate the return value themselves or throw an exception.

The behavior also filters for a method called **SaveTenant** after it has invoked the method on the next behavior in the pipeline (or the target object) and adds to the cache a copy of the tenant object saved by the target object.

This example embeds a filter that determines when the interception behavior should be applied. Later in this chapter, you'll see how you replace this with a policy that you can define in a configuration file or with attributes in your code.

Registering an Interceptor

Although it is possible to use behaviors such as the **CachingInterceptionBehavior** and **LoggingInterceptionBehavior** without the Unity container, the following code sample shows how you can use the container to register the interception behaviors to the **TenantStore** type so that the container sets up the interception whenever it resolves the type.

C#

```
using Microsoft.Practices.Unity.InterceptionExtension;

...

container.AddNewExtension<Interception>();
```

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());
```

The first parameter to the **RegisterType** method, an **Interceptor<InterfaceInterceptor>** object, specifies the type of interception to use. In this example, the next two parameters register the two interception behaviors with the **TenantStore** type.

> The **InterfaceInterceptor** type defines interception based on a proxy object. You can also use the **TransparentProxyInterceptor** and **VirtualMethodInterceptor** types that are described later in this chapter.

The order of the interception behavior parameters determines the order of these behaviors in the pipeline. In this example, the order is important because the caching interception behavior does not pass on the request from the client to the next behavior if it finds the item in the cache. If you reversed the order of these two interception behaviors, you wouldn't get any log messages if the requested item was found in the cache.

Placing all your registration code in one place means that you can manage all the interception behaviors in your application from one location.

Using an Interceptor

The final step is to use the interceptor at run time. The following code sample shows how you can resolve and use a **TenantStore** object that has the logging and caching behaviors attached to it.

C#

```
var tenantStore = container.Resolve<ITenantStore>();
tenantStore.SaveTenant(tenant);
```

The type of the **tenantStore** variable is not **TenantStore**, it is a new dynamically created proxy type that implements the **ITenantStore** interface. This proxy type includes the methods, properties, and events defined in the **ITenantStore** interface.

Figure 1 illustrates the scenario implemented by the two behaviors and type registration you've seen in the previous code samples.

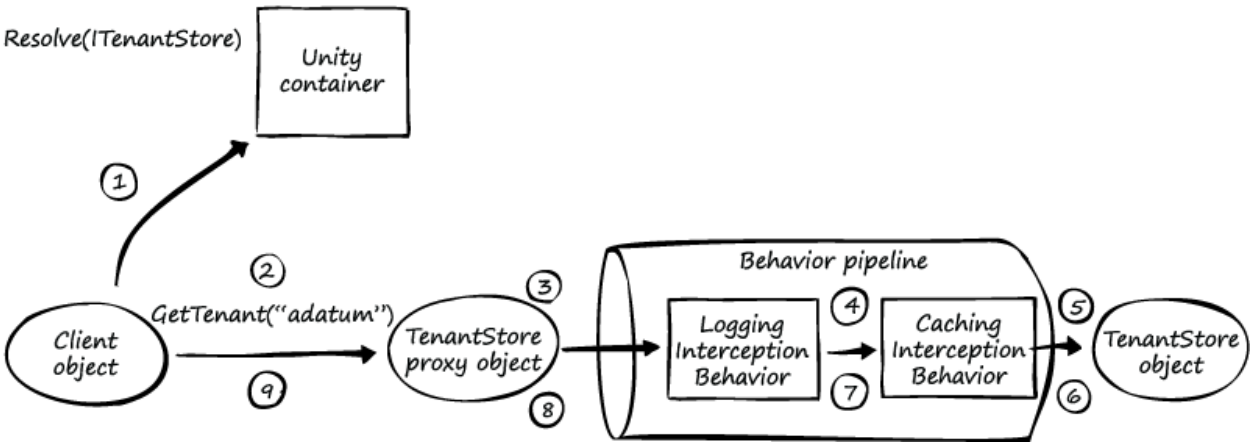

Figure 1 - The behavior pipeline

The numbers in the following list correspond to the numbers in Figure 1.

1. The client object calls the **Resolve** method on the Unity container to obtain an object that implements the **ITenantStore** interface. The container uses the **ITenantStore** registration information to instantiate a **TenantStore** object, create a pipeline with the interception behaviors, and dynamically generate a proxy **TenantStore** object.
2. The client invokes the **GetTenant** method on the **TenantStore** proxy object, passing a single string parameter identifying the tenant to fetch from the store.
3. The **TenantStore** proxy object, calls the **Invoke** method in the first interception behavior. This logging interception behavior logs details of the call.
4. The first interception behavior, calls the **Invoke** method in the next behavior in the pipeline. If the caching behavior finds the tenant in the cache, jump to step 7.
5. The last interception behavior, calls the **GetTenant** method on the **TenantStore** object.
6. The **TenantStore** object returns a tenant to the last interception behavior in the pipeline.
7. If the caching interception behavior found the tenant in the cache it returns the cached tenant, otherwise it returns the tenant from the **TenantStore** object and caches it.
8. The logging interception behavior logs details of the result of the call that it made to the next behavior in the pipeline and returns the result back to the **TenantStore** proxy object.
9. The **TenantStore** proxy object returns the tenant object to the client object.

The interceptors get access to the parameters passed to the original call, the value returned from the original call, and any exceptions thrown by the original call.

Alternative Interception Techniques

The example shown earlier in this chapter illustrated the most common way to use interception in Unity. This section discusses some variations on the basic approach and suggests some scenarios where you might wish to use them.

Instance Interception/Type Interception

The approach used in the example earlier in this chapter is an example of instance interception where Unity dynamically generates a proxy object and enables interception of methods defined in an interface, in this example the **ITenantStore** interface. As a reminder, here is the code that registers the interception.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());
```

In this example, the **Interceptor<InterfaceInterceptor>** parameter specifies you are using a type of instance interception known as interface interception, and the **InterceptionBehavior** parameters define the two behaviors to insert into the behavior pipeline.

Unity interception includes two other interceptor types: **TransparentProxyInterceptor** and **VirtualMethodInterceptor**.

The following table summarizes the available interceptor types:

Interception Type	Interceptor classes
Instance	InterfaceInterceptor
	TransparentProxyInterceptor
Type	VirtualMethodInterceptor

For more information about the different interceptor types, see the topic Unity Interception Techniques.

Using the TransparentProxyInterceptor Type

The **InterfaceInterceptor** type enables you to intercept methods on only one interface; in the example above, you can intercept the methods on the **ITenantStore** interface. If the **TenantStore** class also implements an interface such as the **ITenantLogoStore** interface, and you want to intercept methods defined on that interface in addition to methods defined on the **ITenantStore** interface then you should use the **TransparentProxyInterceptor** type as shown in the following code sample.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<TransparentProxyInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());
```

```
var tenantStore = container.Resolve<ITenantStore>();

// From the ITenantStore interface.
tenantStore.SaveTenant(tenant);

// From the ITenantLogoStore interface.
((ITenantLogoStore)tenantStore).SaveLogo("adatum", logo);
```

The drawback to using the **TransparentProxyInterceptor** type in place of the **InterfaceInterceptor** type is that it is significantly slower at run time. You can also use this approach if the **TenantStore** class doesn't implement any interfaces but does extend the **MarshalByRef** abstract base class.

Although the **TransparentProxyInterceptor** type appears to be more flexible than the **InterfaceInterceptor** type, it is not nearly as performant.

Using the VirtualMethodInterceptor Type

The third interceptor type is the **VirtualMethodInterceptor** type. If you use the **InterfaceInterceptor** or **TransparentProxyInterceptor** type, then at run time Unity dynamically creates a proxy object. However, the proxy object is not type compatible with the target object. In the previous code sample, the **tenantStore** object is not an instance of, or derived from, the **TenantStore** class.

In the example shown below, which uses the **VirtualMethodInterceptor** interception type, the type of the **tenantStore** object derives from the **TenantStore** type so you can use it wherever you can use a **TenantStore** instance. However, you cannot use the **VirtualMethodInterceptor** interceptor on existing objects, you can only configure type interception when you are creating the target object. For example, you cannot use type interception on a WCF service proxy object that is created for you by a channel factory. For the logging and caching interception behaviors to work when you invoke the **SaveTenant** method in the following example, the **SaveTenant** method in the **TenantStore** class must be virtual, and the **TenantStore** class must be public.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<VirtualMethodInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());

var tenantStore = container.Resolve<ITenantStore>();

tenantStore.SaveTenant(tenant);
```

If you use virtual method interception, the container generates a new type that directly extends the type of the target object. The container inserts the behavior pipeline by overriding the virtual methods in the base type.

Another advantage of virtual method interception is that you can intercept internal calls in the class that happen when one method in a class invokes another method in the same class. This is not possible with instance interception.

Calls made with the **VirtualMethodInterceptor** are much faster than those made with the **TransparentProxyInterceptor**.

Figure 2 shows the objects that are involved with interception behaviors in this example.

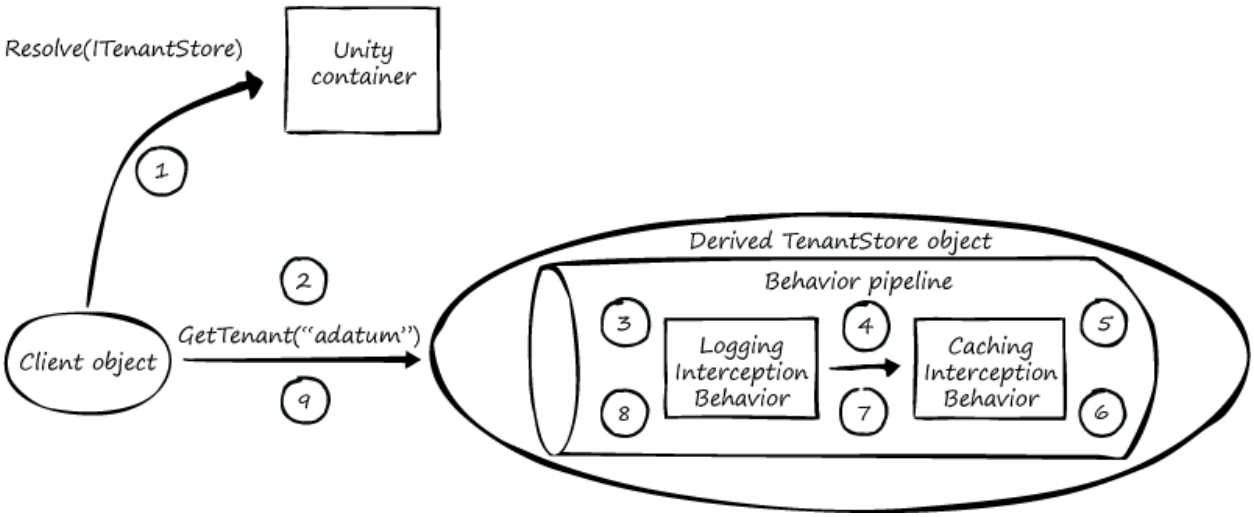

Figure 2 - Interception using the virtual method interceptor

The numbers in the following list correspond to the numbers in Figure 2.

1. The client object calls the **Resolve** method on the Unity container to obtain an object that implements the **ITenantStore** interface. The container uses the **ITenantStore** registration information to instantiate an object from a class derived from the **TenantStore** type that includes a pipeline with the interception behaviors.
2. The client invokes the **GetTenant** method on the **TenantStore** derived object, passing a single string parameter identifying the tenant to fetch from the store.
3. The **TenantStore** derived object, calls the **Invoke** method in the first interception behavior. This logging interception behavior logs details of the call.
4. The first interception behavior, calls the **Invoke** method in the next behavior in the pipeline. If the caching behavior finds the tenant in the cache, jump to step 7.
5. The last interception behavior, calls the **GetTenant** method in the **TenantStore** base class.
6. The **TenantStore** base class returns a tenant to the last interception behavior in the pipeline.

7. If the caching interception behavior found the tenant in the cache it returns the cached tenant, otherwise it returns the tenant from the **TenantStore** object.
8. The logging interception behavior logs details of the result of the call that it made to the next behavior in the pipeline and returns the result back to the **TenantStore** derived object.
9. The **TenantStore** derived object returns the tenant object to the client object.

For more information about the different interceptor types, see Unity Interception Techniques.

Using a Behavior to Add an Interface to an Existing Class

Sometimes it is useful to be able to add an interface implementation to an existing class without changing that class. For example, the following **TenantStore** class only implements the **ITenantStore** interface.

C#

```
public class TenantStore : ITenantStore
{
  ...
}
```

However, you may have a requirement for **TenantStore** instances to implement some custom logging methods such as **WriteLogMessage** defined in the **ILogger** interface. This requirement may be in addition to that of using interception to write log messages when you invoke methods, such as **GetTenant** or **SaveTenant,** on the **TenantStore** class. Using a behavior, you can make it possible to write code such as that shown in the following example without modifying the original **TenantStore** class.

C#

```
((ILogger)tenantStore).WriteLogMessage("Message: Write to the log directly...");
```

When you register the **TenantStore** type with the container, you can specify that it supports additional interfaces, such as the **ILogger** interface, as shown in the following example:

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>(),
  new AdditionalInterface<ILogger>());
```

Then, in a behavior, you can intercept any calls made to methods defined on that interface. The following code sample shows part of the **LoggingInterceptionBehavior** class that filters for calls to methods on the **ILogger** interface and handles them.

C#

```
class LoggingInterceptionBehavior : IInterceptionBehavior
{
```

```
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextInterceptionBehaviorDelegate getNext)
  {
    var methodReturn = CheckForILogger(input);

    if (methodReturn != null)
    {
      return methodReturn;
    }

    ...
  }

  ...

  private IMethodReturn CheckForILogger(IMethodInvocation input)
  {
    if (input.MethodBase.DeclaringType == typeof(ILogger)
         && input.MethodBase.Name == "WriteLogMessage")
    {
      WriteLog(input.Arguments["message"].ToString());
      return input.CreateMethodReturn(null);
    }
    return null;
  }
}
```

Note how in this example the behavior intercepts the call to the **WriteLogMessage** interface and does not forward it on to the target object. This is because the target object does not implement the **ILogger** interface and does not have a **WriteLogMessage** method.

Interception Without the Unity Container

The examples you've seen so far show how to setup interception as part of the type registration in the Unity container. If you are not using a Unity container for DI or want to use interception without using a container, you can use the **Intercept** class. The following two examples show how you can setup interception using the Unity container, and using the **Intercept** class. Both examples attach the same interception pipeline to a **TenantStore** object.

C#

```
// Example 1. Using a container.
// Configure the container for interception.
container = new UnityContainer();
container.AddNewExtension<Interception>();

// Register the TenantStore type for interception.
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
```

```
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());

// Obtain a proxy object with an interception pipeline.
var tenantStore = container.Resolve<ITenantStore>();
```

C#

```
// Example 2. Using the Intercept class.
ITenantStore tenantStore = Intercept.ThroughProxy<ITenantStore>(
  new TenantStore(tenantContainer, blobContainer),
  new InterfaceInterceptor(),
  new IInterceptionBehavior [] {
    new LoggingInterceptionBehavior(), new CachingInterceptionBehavior()
  });
```

It is sometimes convenient to use stand-alone interception and the **Intercept** class if you want to attach an intercept pipeline to an existing object as shown in the following example.

C#

```
TenantStore tenantStore =
  new TenantStore(tenantContainer, blobContainer);

...

// Attach an interception pipeline.
ITenantStore proxyTenantStore = Intercept.ThroughProxy<ITenantStore>(
  tenantStore,
  new InterfaceInterceptor(),
    new IInterceptionBehavior [] {
      new LoggingInterceptionBehavior(), new CachingInterceptionBehavior()
  });
```

You can use the **ThroughProxy** methods of the **Intercept** class to set up instance interception that uses a proxy object , and the **NewInstance** methods to set up type interception that uses a derived object. You can only attach an interception pipeline to an existing object if you use the **ThroughProxy** methods; the **NewInstance** methods always create a new instance of the target object.

You don’t need to use a Unity container if you want to use interception in your application.

For more information, see Stand-alone Unity Interception.

Design Time Configuration

All of the examples so far in this chapter have configured interception programmatically, either in a Unity container or by using the **Intercept** class. However, in the same way that you can define your type registrations for the container in a configuration file, you can also define your interception behavior pipeline.

You've seen the following registration code for the **TenantStore** class several times already in this chapter.

C#

```
// Configure the container for interception.
container = new UnityContainer();
container.AddNewExtension<Interception>();

// Register the TenantStore type for interception.
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());

// Obtain a proxy object with an interception pipeline.
var tenantStore = container.Resolve<ITenantStore>();
```

Instead of this programmatic configuration, you can add the following configuration information to your configuration file.

XML

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <sectionExtension type=
    "Microsoft.Practices.Unity.InterceptionExtension
    .Configuration.InterceptionConfigurationExtension,
    Microsoft.Practices.Unity.Interception.Configuration" />
  <namespace name="Tailspin.Web.Survey.Shared.Stores" />
  <namespace name="Tailspin.Utilities.InterceptionBehaviors" />
  <assembly name="Tailspin.Web.Survey.Shared" />
  <assembly name="Tailspin.Utilities" />
  <container>
    <register type="ITenantStore" mapTo="TenantStore">
      <interceptor type="InterfaceInterceptor"/>
      <interceptionBehavior type="LoggingInterceptionBehavior" />
      <interceptionBehavior type="CachingInterceptionBehavior" />
    </register>
  </container>
</unity>
```

Note that this snippet from the configuration file includes a **sectionExtension** element to enable you to use the interception specific elements. There are also **namespace** and **assembly** elements to enable Unity to find your interception behavior classes in addition to the **TenantStore** class and **ITenantStore** interface.

The following code sample shows how you can load this configuration information and then resolve a **TenantStore** instance with an interception pipeline that includes the logging and caching behaviors.

C#

```
IUnityContainer container = new UnityContainer();
```

```
container.LoadConfiguration();

// Obtain a proxy object with an interception pipeline.
var tenantStore = container.Resolve<ITenantStore>();

...

tenantStore.UploadLogo("tenant", logo);
```

Defining the interception behaviors in the configuration file makes it possible to change their configuration without recompiling the code.

Policy Injection

The following code sample shows how you insert two behaviors into the pipeline for a **TenantStore** instance created by the Unity container.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new Interceptor<InterfaceInterceptor>(),
  new InterceptionBehavior<LoggingInterceptionBehavior>(),
  new InterceptionBehavior<CachingInterceptionBehavior>());
```

One of the drawbacks of this approach to implementing support for crosscutting concerns is that you must configure the interceptors for every class that needs them. In practice, you may have additional store classes in your application that all require caching support, and many more that require logging behavior.

You may be able to use registration by convention to configure interceptors for multiple registered types.

The following code sample shows an alternative approach based on policies for adding interception behaviors to objects in your application. Note that definitions of the **LoggingCallHandler** and **CachingCallHandler** classes are given later in this section.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<InterfaceInterceptor>());

container.RegisterType<ISurveyStore, SurveyStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<InterfaceInterceptor>());
```

```
var first = new InjectionProperty("Order", 1);
var second = new InjectionProperty("Order", 2);

container.Configure<Interception>()
  .AddPolicy("logging")
  .AddMatchingRule<AssemblyMatchingRule>(
    new InjectionConstructor(
    new InjectionParameter("Tailspin.Web.Survey.Shared")))
  .AddCallHandler<LoggingCallHandler>(
    new ContainerControlledLifetimeManager(),
    new InjectionConstructor(),
    first);

container.Configure<Interception>()
  .AddPolicy("caching")
  .AddMatchingRule<MemberNameMatchingRule>(
    new InjectionConstructor(new [] {"Get*", "Save*"}, true))
  .AddMatchingRule<NamespaceMatchingRule>(
    new InjectionConstructor(
      "Tailspin.Web.Survey.Shared.Stores", true))
  .AddCallHandler<CachingCallHandler>(
    new ContainerControlledLifetimeManager(),
    new InjectionConstructor(),
    second);
```

This example registers two store types with the container using a **PolicyInjectionBehavior** behavior type. This behavior makes use of policy definitions to insert handlers into a pipeline when a client calls an object instantiated by the container. The example shows two policy definitions: one for the **LoggingCallHandler** interception handler and one for the **CachingCallHandler** interception handler. Each policy has a name to identify it and one or more matching rules to determine when to apply it.

You must remember to configure policy injection behavior for each type that you want to use it on as well as defining the policies.

The Policy Injection Application Block has built-in matching rules based on the following:

- Assembly name
- Namespace
- Type
- Tag attribute
- Custom attribute

- Member name
- Method signature
- Parameter type
- Property
- Return type

For more information about these matching rules and how to use them, see the topic Policy Injection Matching Rules.

You can also define your own, custom matching rule types. For more information, see the topic Creating Policy Injection Matching Rules.

The policy for logging has a single matching rule based on the name of the assembly that contains the class definition of the target object: in this example, if the **TenantStore** and **SurveyStore** classes are located in the **Tailspin.Web.Survey.Shared** assembly, then the handler will log all method calls to those objects.

The policy for caching uses two matching rules that are ANDed together: in this example, the caching handler handles all calls to methods that start either with **Get** or **Save** (the **true** parameter makes it a case sensitive test), and that are in the **Tailspin.Web.Survey.Shared.Stores** namespace.


You can use the policy injection matching rules to avoid the need embed filters in an injection behavior class.

The **first** and **second** parameters control the order that the container invokes the handlers if the matching rules match multiple handlers to a single instance. In this example, you want to be sure that the container invokes the logging handler before the caching handler in order to ensure that all calls are logged.

Remember, in the example used in this chapter, the caching handler does not pass on the call if it locates the item in the cache.

Both policies use the **ContainerControlledLifetimeManager** type to ensure that the handlers are singleton objects in the container.

The handler classes are very similar to the behavior interception classes you saw earlier in this chapter. The following code samples shows the logging handler and the cache handler used in the examples shown in this section.

C#

```
class LoggingCallHandler : ICallHandler
{
```

```
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
  {
    // Before invoking the method on the original target
    WriteLog(String.Format("Invoking method {0} at {1}",
      input.MethodBase, DateTime.Now.ToLongTimeString()));

    // Invoke the next handler in the chain
    var result = getNext().Invoke(input, getNext);

    // After invoking the method on the original target
    if (result.Exception != null)
    {
      WriteLog(String.Format("Method {0} threw exception {1} at {2}",
        input.MethodBase, result.Exception.Message,
        DateTime.Now.ToLongTimeString()));
    }
    else
    {
      WriteLog(String.Format("Method {0} returned {1} at {2}",
        input.MethodBase, result.ReturnValue,
        DateTime.Now.ToLongTimeString()));
    }

    return result;
  }

  public int Order
  {
    get;
    set;
  }

  private void WriteLog(string message)
  {
    ...
  }
}
```

C#

```
public class CachingCallHandler :ICallHandler

{
  public IMethodReturn Invoke(IMethodInvocation input,
    GetNextHandlerDelegate getNext)
  {
    //Before invoking the method on the original target
    if (input.MethodBase.Name == "GetTenant")
    {
```

```
      var tenantName = input.Arguments["tenant"].ToString();
      if (IsInCache(tenantName))
      {
        return input.CreateMethodReturn(FetchFromCache(tenantName));
      }
    }

    IMethodReturn result = getNext()(input, getNext);

    //After invoking the method on the original target
    if (input.MethodBase.Name == "SaveTenant")
    {
      AddToCache(input.Arguments["tenant"]);
    }

    return result;
  }

  public int Order
  {
    get;
    set;
  }

  private bool IsInCache(string key)
  {
    ...
  }

  private object FetchFromCache(string key)
  {
    ...
  }

  private void AddToCache(object item)
  {
    ...
  }
}
```

In many cases, policy injection handler classes may be simpler than interception behavior classes that address the same crosscutting concerns. This is because you can use the policy matching rules to control when the container invokes the handler classes whereas very often with interception behavior classes, such as the **CachingInterceptionBehavior** class shown earlier in the chapter, you need to implement a filter within **Invoke** method to determine whether the behavior should execute in a particular circumstance.

The **Order** property, which is set when you configure the policy, controls the order in which the container executes the handlers when a policy rule matches more than one handler.


You may need to consider the order of the call handlers carefully, especially if some of them don't always pass the call on to the next handler.

Policy Injection and Attributes

Using policies with matching rules such as those shown in the previous example means that you can apply and manage the policies when you configure your container. You can define the policies either in code, as shown in the example, or in a configuration file in a similar way to that described in the section "Design Time Configuration" earlier in this chapter.

However, an alternative approach is to use attributes in your business classes to indicate whether the container should invoke a particular call handler. In this case, the person writing the business class is responsible for making sure that all the crosscutting concerns are addressed.


Typically, if you define policies for your call handlers, they are all defined in a single class or configuration file where they are easy to manage. Using attributes means that information about which crosscutting concerns are associated with particular classes and methods is stored in those classes which is a less maintainable approach.

In this scenario, you can register the types that the container should inject with policies as shown in the following code sample.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<InterfaceInterceptor>());

container.RegisterType<ISurveyStore, SurveyStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<InterfaceInterceptor>());
```

There is no need to configure any policies, but you do need to define your attributes. The following code sample shows the **LoggingCallHandlerAttribute** class. The **CachingCallHandlerAttribute** class is almost identical.

C#

```
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.InterceptionExtension;

class LoggingCallHandlerAttribute : HandlerAttribute
```

```
{
  private readonly int order;

  public LoggingCallHandlerAttribute(int order)
  {
    this.order = order;
  }

  public override ICallHandler CreateHandler(IUnityContainer container)
  {
    return new LoggingCallHandler() { Order = order };
  }
}
```

The **CreateHandler** method creates an instance of the **LoggingCallHandler** class that you saw previously and sets the value of the **Order** property.

You can now use the attributes to decorate your business classes as shown in the following example.

C#

```
class MyTenantStoreWithAttributes : ITenantStore, ITenantLogoStore
{
  ...

  [LoggingCallHandler(1)]
  public void Initialize()
  {
    ...
  }

  [LoggingCallHandler(1)]
  [CachingCallHandler(2)]
  public Tenant GetTenant(string tenant)
  {
    ...
  }

  [LoggingCallHandler(1)]
  public IEnumerable<string> GetTenantNames()
  {
    ...
  }

  [LoggingCallHandler(1)]
  [CachingCallHandler(2)]
  public void SaveTenant(Tenant tenant)
  {
    ...
  }
```

```
  [LoggingCallHandler(1)]
  public virtual void UploadLogo(string tenant, byte[] logo)
  {
    ...
  }
}
```

In this example, the **LoggingCallHandler** call handler logs all the method calls, and the **GetTenant** and **SaveTenant** methods also have a caching behavior applied to them. The example uses the **Order** property of the handlers to place the logging call handler before the caching call handler in the pipeline.

Policy Injection and the Enterprise Library Blocks

In the previous examples, you implemented the caching and logging behaviors in your own call handler classes: **CachingCallHandler** and **LoggingCallHandler**. The blocks in the Enterprise Library, such as the Logging Application Block and the Validation Application Block, provide some pre-written call handlers that you can use in your own applications. This enables you to use the Enterprise Library blocks to address crosscutting concerns in your application using policy injection.


Using the Enterprise Library blocks to address your cross-cutting concerns is often easier than implementing the behaviors yourself.

For example, you could replace your **LoggingCallHandler** with **LogCallHandler** from the Enterprise Library Policy Injection Application Block as shown in the following code sample. However, Enterprise Library 6 does not include a caching handler.

C#

```
ConfigureLogger();
container.RegisterType<ITenantStore, TenantStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<InterfaceInterceptor>());

var second = new InjectionProperty("Order", 2);

container.Configure<Interception>()
  .AddPolicy("logging")
  .AddMatchingRule<AssemblyMatchingRule>(
    new InjectionConstructor(
      new InjectionParameter("Tailspin.Web.Survey.Shared")))
  .AddCallHandler<LogCallHandler>(
    new ContainerControlledLifetimeManager(),
    new InjectionConstructor(
      9001, true, true,
      "start", "finish", true, false, true, 10, 1));
```

```
container.Configure<Interception>()
  .AddPolicy("caching")
  .AddMatchingRule<MemberNameMatchingRule>(
    new InjectionConstructor(new[] { "Get*", "Save*" }, true))
  .AddMatchingRule<NamespaceMatchingRule>(
    new InjectionConstructor(
      "Tailspin.Web.Survey.Shared.Stores", true))
  .AddCallHandler<CachingCallHandler>(
    new ContainerControlledLifetimeManager(),
    new InjectionConstructor(),
    second);
```

The **LogCallHandler** class from the Logging Application Block is a call handler to use with the policy injection framework.

It's important that you configure the Logging Application Block before you configure the policy injection. The sample code in the PIABSamples solution that accompanies this guide includes a method **ConfigureLogger** to do this.

The **LogCallHandler** constructor parameters enable you to configure the logging behavior and specify the order of the handler in relation to other call handlers. In this example, the container will invoke the **LogCallHandler** call handler before the user defined **CachingCallHandler** call handler.

If you want to control the behavior of the call handlers by using attributes in your business classes instead of through policies, you can enable this approach as shown in the following code sample.

C#

```
container.RegisterType<ITenantStore, TenantStore>(
  new InterceptionBehavior<PolicyInjectionBehavior>(),
  new Interceptor<TransparentProxyInterceptor>());
```

You can then use one of the attributes, such as the **LogCallHandler** attribute, defined in the Enterprise Library blocks as shown in the following example.

C#

```
[LogCallHandler(AfterMessage = "After call",
                BeforeMessage = "Before call",
                Categories = new string[] { "General" },
                EventId = 9002,
                IncludeCallStack = false,
                IncludeCallTime = true,
                IncludeParameters = true,
                LogAfterCall = true,
                LogBeforeCall = true,
                Priority = 10,
                Severity = System.Diagnostics.TraceEventType.Information,
```

```
                Order = 1)]
[CachingCallHandler(2)]
public Tenant GetTenant(string tenant)
{
  ...
}
```

You can customize the information contained in the log message using attribute parameters, and control the order of the handlers in the pipeline using the **Order** parameter.

An alternative approach to configuring the Enterprise Library call handlers is through the **PolicyInjection** static façade. The sample code in the PIABSamples solution that accompanies this guide includes an example of this approach.

A Real World Example

This example is from the aExpense reference implementation that accompanies the Enterprise Library guidance: you can download this sample application from http://go.microsoft.com/fwlink/p/?LinkID=290917. The aExpense application is an ASP.NET application that makes extensive use of the Enterprise Library blocks, including Unity and Policy Injection. This example is taken from the aExpense implementation that use Enterprise Library 6.

The goal is to use the Semantic Logging Application Block to log calls to a specific method in the data access layer of the application. The following code sample shows part of the **ExpenseRepository** class where the **SaveExpense** method has a **Tag** attribute that identifies a policy.

C#

```
public class ExpenseRepository : IExpenseRepository
{
  ...

  [Tag("SaveExpensePolicyRule")]
  public virtual void SaveExpense(Model.Expense expense)
  {
    ...
  }

  ...
}
```

Note that the **ExpenseRepository** model class implements the **IExpenseRepository** interface: this means that you can use the **VirtualMethodInterceptor** interceptor type. The following code sample shows the registration of the **ExpenseRepository** type that includes adding the **PolicyInjectionBehavior**.

C#

```
container
  .RegisterType<IExpenseRepository, ExpenseRepository>(
    new Interceptor<VirtualMethodInterceptor>(),
    new InterceptionBehavior<PolicyInjectionBehavior>());
```

The following code sample shows the **SemanticLogCallHandler** call handler class that performs the logging.

C#

```
public class SemanticLogCallHandler : ICallHandler
{
    public SemanticLogCallHandler(NameValueCollection attributes)
    {
    }

    public IMethodReturn Invoke(IMethodInvocation input,
                                GetNextHandlerDelegate getNext)
    {
        if (getNext == null) throw new ArgumentNullException("getNext");

        AExpenseEvents.Log.LogCallHandlerPreInvoke(
          input.MethodBase.DeclaringType.FullName, input.MethodBase.Name);

        var sw = Stopwatch.StartNew();

        IMethodReturn result = getNext()(input, getNext);

        AExpenseEvents.Log.LogCallHandlerPostInvoke(
          input.MethodBase.DeclaringType.FullName, input.MethodBase.Name,
          sw.ElapsedMilliseconds);

        return result;
    }

    public int Order { get; set; }
}
```

This call handler uses the Semantic Logging Application Block to log events before and after the call, and to record how long the call took to complete.

The application configures the Policy Injection Application Block declaratively. The following snippet from the Web.config file shows how the **Tag** attribute is associated with the **SemanticLogCallHandler** type.

XML

```
<policyInjection>
  <policies>
    <add name="ExpenseTracingPolicy">
      <matchingRules>
        <add name="TagRule" match="SaveExpensePolicyRule" ignoreCase="false"
           type="Microsoft.Practices.EnterpriseLibrary.PolicyInjection
                 .MatchingRules.TagAttributeMatchingRule, ..."/>
      </matchingRules>
      <handlers>
```

```
        <add type="AExpense.Instrumentation.SemanticLogCallHandler, AExpense,
                  Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
                  name="SemanticLogging Call Handler" />
      </handlers>
    </add>
  </policies>
</policyInjection>
```

Summary

In this chapter, you learned how to use Unity and the Unity container to add support for crosscutting concerns in your application by using interception. Interception enables you to implement support for crosscutting concerns while continuing to write code that follows the single responsibility principle and the open/closed principle. The chapter also described how you can use the policy injection technique that allows you define policies to manage how and where in your application you address the crosscutting concerns. As an alternative to defining policies that are typically managed in a single class in your project or in a configuration file, you can also use attributes that enable developers to decorate their code to specify how they want to address the crosscutting concerns.

Chapter 6 - Extending Unity

Introduction

Although Unity is very flexible, there are scenarios where you might need to extend Unity in order to meet some specific requirements. Unity is extensible in many different ways, some of which you have already seen in the previous chapters of this guide, other extensibility options are covered in this chapter.

In Chapter 5, "Interception with Unity," you saw how you can customize and extend the ways that Unity supports interception by:

- Creating custom behaviors. Interception in Unity uses interception behaviors in a pipeline to implement your crosscutting concerns. You implement your custom behaviors by implementing the **IInterceptionBehavior** interface.
- Creating custom policy injection matching rules. If you use policy injection to insert behaviors into the interception pipeline, you can add your own custom matching rules by implementing the **IMatchingRule** interface.
- Creating custom policy injection call handlers. Again, if you use policy injection, you can create custom handler classes that enable you to define custom behaviors in the policy injection pipeline. You do this by implementing the **ICallHandler** interface.
- Creating custom handler attributes. Associated with custom call handlers, you can also define custom attributes to decorate methods in your business classes to control the way that policy injection works in your application.

In this chapter, you'll see how you can create custom lifetime managers and extend the Unity container. Chapter 3, "Dependency Injection with Unity," describes the built-in lifetime managers such as the **TransientLifetimeManager**, the **ContainerControlledLifetimeManager**, and the **HierarchicalLifetimeManager** and how they manage the lifetime of the objects instantiated by the Unity container. In this chapter, you'll see an example of how Unity implements a simple caching lifetime manager as a guide to creating your own custom lifetime managers.

By creating a Unity container extension, you can customize what Unity does when you register and resolve types. In this chapter, you'll see an example of how you can add functionality to the container that automatically wires up event handlers when you resolve objects from the container.

Creating Custom Lifetime Managers

Chapter 3, "Dependency Injection with Unity," describes how you can use Unity's built-in lifetime managers to manage the lifetime of the objects that Unity creates when you resolve types registered with the container. For example, if you use the default **TransientLifetimeManager** lifetime manager, the container does not hold a reference to the resolved object and such objects should not, in general, hold

on to any resources that must be disposed deterministically. If, on the other hand, you use the **ContainerControlledLifetimeManager** lifetime manager, the container itself tracks the object and is responsible for determining when the object it creates becomes eligible for garbage collection.

In addition to the seven built-in lifetime managers (**ContainerControlledLifetimeManager**, **TransientLifetimeManager**, **PerResolveLifetimeManager**, **PerThreadLifetimeManager**, **ExternallyControlledLifetimeManager**, **PerRequestLifetimeManager** and **HierarchicalLifetimeManager**), you can also create your own custom lifetime managers by extending either the abstract **LifetimeManager** class or the abstract **SynchronizedLifetimeManager** class in the **Microsoft.Practices.Unity** namespace.

You should extend the **LifetimeManager** class when your custom lifetime manager does not need to concern itself with concurrency issues: the built-in **PerThreadLifetimeManager** class extends the **LifetimeManager** class because it will not encounter any concurrency issues when it stores a resolved object in thread local storage. However, the built-in **ContainerControlledLifetimeManager** class could encounter concurrency issues when it stores a reference to a newly created object: therefore, it extends the **SynchronizedLifetimeManager** class.

You may find it useful to open the sample application, "CustomLifetimeManagers," that accompanies this guide in Visual Studio while you read this section.

Lifetime Managers and Resolved Objects

Before examining an example of a custom lifetime manager, you should understand the relationship between lifetime managers and the objects resolved by the container. Given the following code sample, the container creates an instance of the **ContainerControlledLifetimeManager** type during the call to the **RegisterType** method.

C#

```
IUnityContainer container = new UnityContainer();
container.RegisterType<Tenant>(new ContainerControlledLifetimeManager());
```

When you execute the following line of code, the container creates a new **Tenant** instance that is referenced by the **ContainerControlledLifetimeManager** instance.

C#

```
var tenant = container.Resolve<Tenant>();
```

If you call the **Resolve** method a second time, the container returns a reference to the existing **Tenant** instance that is referenced by the **ContainerControlledLifetimeManager** instance.

Both the **Tenant** instance and the **ContainerControlledLifetimeManager** instance will remain in memory until the container itself is garbage collected by the runtime.

Extending the SynchronizedLifetimeManager Type

The example in this section shows how to create a custom lifetime manager that uses a cache to store the objects resolved by the container. It extends the **SynchronizedLifetimeManager** class because potentially two (or more) clients could try to store an instance of a type in the cache at the same time.

> The sample shows the basics of creating a custom lifetime manager. However, it does not show how to create a fully featured lifetime manager. For example, the code shown here does not support registering generic types.

This example defines a very simple interface that abstracts the logic of dealing with the cache from the lifetime manager. The following code sample shows this interface.

C#

```
public interface IStorage
{
  object GetObject(string key);
  void StoreObject(string key, object value);
  void RemoveObject(string key);
}
```

The following code sample shows the implementation of the **CachedLifetimeManager** class that provides overrides of three methods from the **SynchronizedLifetimeManager** class.

C#

```
public class CachedLifetimeManager
  : SynchronizedLifetimeManager, IDisposable
{
  private IStorage storage;
  private string key;

  public CachedLifetimeManager(IStorage storage)
  {
    this.storage = storage;
    this.key = Guid.NewGuid().ToString();
  }

  public string Key
  {
    get { return key; }
  }


  protected override object SynchronizedGetValue()
  {
    return storage.GetObject(key);
  }

  protected override void SynchronizedSetValue(object newValue)
```

```
  {
    storage.StoreObject(key, newValue);
  }

  public override void RemoveValue()
  {
    Dispose();
  }

  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected void Dispose(bool disposing)
  {
    if (disposing && storage != null)
    {
      storage.RemoveObject(key);
      storage = null;
    }
  }
}
```

The **SynchronizedGetValue** method could return a null if the object does not exist in the cache; in this scenario, the container instantiates a new object and then calls **SynchronizedSetValue** to add it to the cache.

> Nothing in Unity calls the **RemoveValue** method: it is shown here for completeness.

When Unity creates an instance of the **SynchronizedLifetimeManager** class, the constructor generates a GUID to use as the cache key for whatever object this instance **SynchronizedLifetimeManager** class is responsible for managing.

This example also provides a simple implementation of the **IDisposable** interface so that when the Unity container is disposed, the custom lifetime manager has the opportunity to perform any clean up; in this case, removing the resolved object from the cache.

The following code sample shows how you can use this custom lifetime manager. The **SimpleMemoryCache** class implements the **IStorage** interface.

C#

```
class Program
{
  static void Main(string[] args)
  {
    IUnityContainer container = new UnityContainer();
    var cache = new SimpleMemoryCache();
```

```
    container.RegisterType<Tenant>(new CachedLifetimeManager(cache));
    var tenant = container.Resolve<Tenant>();
    var tenant2 = container.Resolve<Tenant>();
  }
}
```

The sequence diagram shown in Figure 1 summarizes how the container uses the custom lifetime manager.

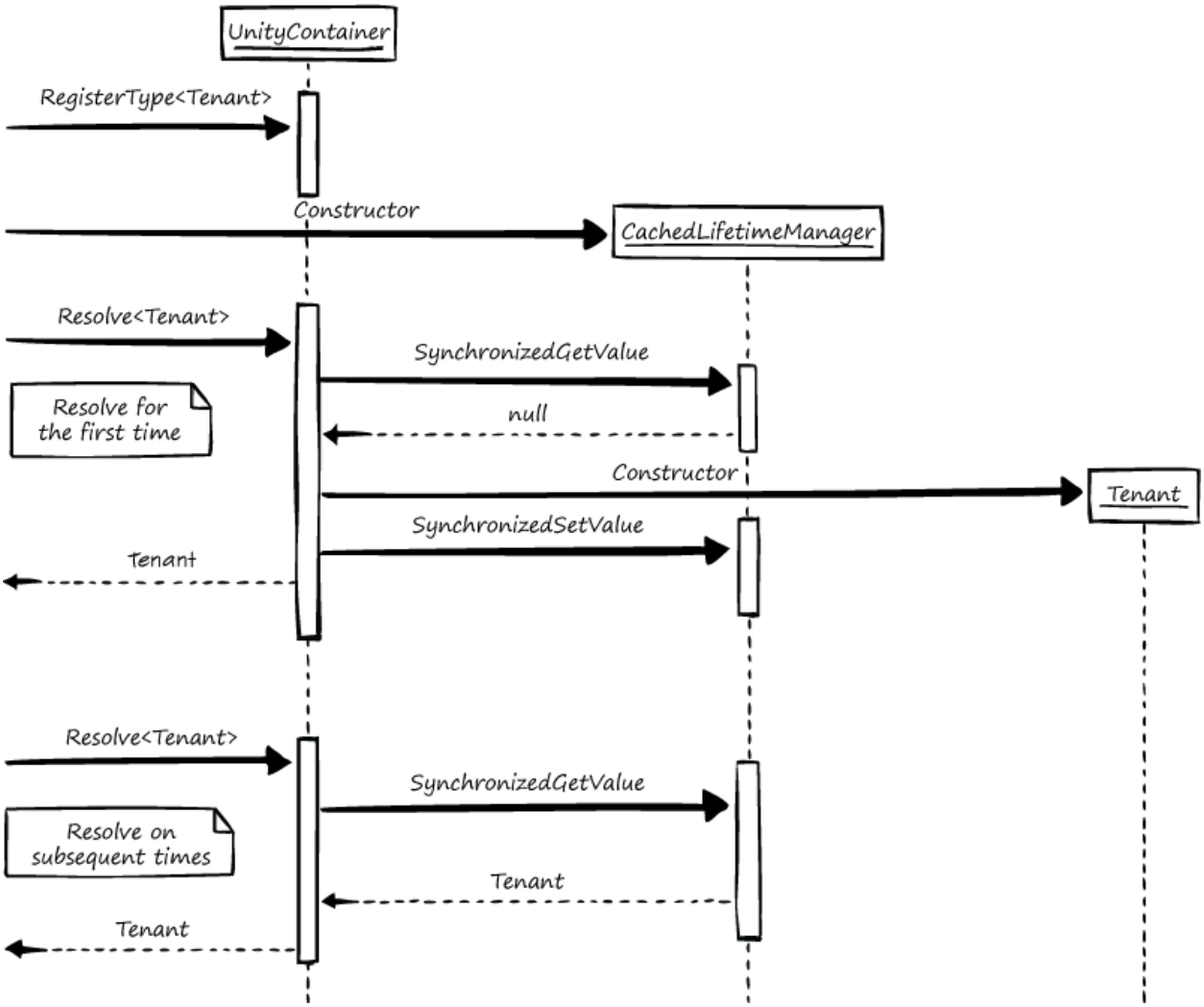

Figure 1 - Container Interactions with a Custom Lifetime Manager

The sequence diagram shows how the call to the **RegisterType** method results in a new **CachedLifetimeManager** instance. The first time you call the **Resolve** method, the **CachedLifetimeManager** instance does not find an instance of the requested type in the cache, so the Unity container creates an instance of the requested type and then calls **SynchronizedSetValue** to store the instance in the cache. Subsequent calls to the **Resolve** method result in the object being returned from the cache.

Extending the LifetimeManager Type

Extending the **LifetimeManager** class is almost the same as extending the **SynchronizedLifetimeManager** class. Instead of overriding the **SynchronizedGetValue** and **SynchronizedSetValue** methods, you override the **GetValue** and **SetValue** methods.

Extending the Unity Container

Unity enables you to create container extensions that add functionality to a container. Unity uses this extension mechanism to implement some of its own functionality such as interception: to start using interception, the first thing you must do is to add the container extension as shown in the following code sample.

C#

```
using Microsoft.Practices.Unity.InterceptionExtension;

...

IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>();
```

This section will show you an example of a container extension that can automatically wire up event handlers in objects that you resolve from the container. However, before seeing how to implement the container extension, you should understand what this extension does. This will make it easier for you to follow the details of the extension implementation later in this chapter.

Without the Event Broker Container Extension

The example uses a pair of very simple classes to illustrate the functionality of the extension. The two classes are the **Publisher** and **Subscriber** classes shown in the following code sample. These two classes use some basic event declarations.

C#

```
class Publisher
{
  public event EventHandler RaiseCustomEvent;

  public void DoSomething()
  {
    OnRaiseCustomEvent();

  }

  protected virtual void OnRaiseCustomEvent()
  {
    EventHandler handler = RaiseCustomEvent;

    if (handler != null)
```

```
    {
      handler(this, EventArgs.Empty);
    }
  }
}

class Subscriber
{
  private string id;
  public Subscriber(string ID, Publisher pub)
  {
    id = ID;
    pub.RaiseCustomEvent += HandleCustomEvent;
  }

  public void HandleCustomEvent(object sender, EventArgs e)
  {
    Console.WriteLine(
      "Subscriber {0} received this message at: {1}", id, DateTime.Now);
  }
}
```

If you don't have the event broker extension and you want to use Unity to instantiate a single **Publisher** instance wired to two **Subscriber** instances, then you have several options. For example, you could register and resolve the types as shown in the following code sample:

C#

```
IUnityContainer container = new UnityContainer();
container
  .RegisterType<Publisher>(
    new ContainerControlledLifetimeManager())
  .RegisterType<Subscriber>(
    new InjectionConstructor("default", typeof(Publisher)));

  var pub = container.Resolve<Publisher>();
  var sub1 = container.Resolve<Subscriber>(
    new ParameterOverride("ID", "sub1"));
  var sub2 = container.Resolve<Subscriber>(
    new ParameterOverride("ID", "sub2"));

// Call the method that raises the event.
pub.DoSomething();
```

In this example, it's important to use the **ContainerControlledLifetimeManager** lifetime manager when you register the **Publisher** class: this ensures that the container creates a single **Publisher** instance that the **Resolve** method returns and that the container passes to the **Subscriber** class constructor when resolving the **Subscriber** type. Resolving the two **Subscriber** instances uses a parameter override to pass different IDs to the two instances so that you can identify the output from each instance.

With the Event Broker Extension

With the event broker extension, it's possible to simplify the implementation of the **Subscriber** class as shown in the following code sample.

You may find it useful to open the sample application, "EventBroker," that accompanies this guide in Visual Studio while you read this section.

C#

```
class Publisher
{
  [Publishes("CustomEvent")]
  public event EventHandler RaiseCustomEvent;

  public void DoSomething()
  {
    OnRaiseCustomEvent();

  }

  protected virtual void OnRaiseCustomEvent()
  {
    EventHandler handler = RaiseCustomEvent;

    if (handler != null)
    {
      handler(this, EventArgs.Empty);
    }
  }
}

class Subscriber
{
  private string id;

  public Subscriber(string ID)
  {
    id = ID;
  }

  [SubscribesTo("CustomEvent")]
  public void HandleCustomEvent(object sender, EventArgs e)
  {
    Console.WriteLine(
      "Subscriber {0} received this message at: {1}", id, DateTime.Now);
  }
}
```

In these new versions of the two classes, the event in the **Publisher** class is decorated with the **Publishes** attribute, and the handler method in the **Subscriber** class is decorated with the **SubscribesTo** attribute. In addition, the constructor in the **Subscriber** class is much simpler in that it no longer receives a reference to the **Publisher** class and no longer hooks up the event handler.


You have effectively decoupled the **Subscriber** class from the **Publisher** class. There is no longer a reference to the **Publisher** type anywhere in the **Subscriber** class.

The registration code is now also simpler.

C#

```
IUnityContainer container = new UnityContainer();
container
  .AddNewExtension<SimpleEventBrokerExtension>()
  .RegisterType<Publisher>()
  .RegisterType<Subscriber>(new InjectionConstructor("default"));

  var sub1 = container.Resolve<Subscriber>(
    new ParameterOverride("ID", "sub1"));
  var sub2 = container.Resolve<Subscriber>(
    new ParameterOverride("ID", "sub2"));

// Call the method
pub.DoSomething();
```

The **AddNewExtension** method registers the container extension, which is custom class that extends the abstract **UnityContainerExtension** class. Registering the **Publisher** type now uses the default **TransientLifetimeManager** lifetime manager, and the **Subscriber** registration only needs to define how the **ID** value is passed to the constructor.

Implementing the Simple Event Broker

Now that you've seen what the event broker extension does, using attributes to define how the container should wire up event handlers and simplifying the registration of the two classes, it's time to see how to implement the extension.

An **EventBroker** class tracks the event subscriber classes that subscribe to events in event publisher classes. The following code sample shows an outline of this class.

C#

```
public class EventBroker
{
  ...

  public IEnumerable<string> RegisteredEvents
  {
```

```
    get
    {
      ...
    }
  }

  public void RegisterPublisher(string publishedEventName,
    object publisher, string eventName)
  {
    ...
  }

  public void UnregisterPublisher(string publishedEventName,
    object publisher, string eventName)
  {
    ...
  }

  public void RegisterSubscriber(string publishedEventName,
    EventHandler subscriber)
  {
    ...
  }

  public void UnregisterSubscriber(string publishedEventName,
    EventHandler subscriber)
  {
    ...
  }

  public IEnumerable<object> GetPublishersFor(string publishedEvent)
  {
    ...
  }

  public IEnumerable<EventHandler> GetSubscribersFor(string publishedEvent)
  {
    ...
  }

  private PublishedEvent GetEvent(string eventName)
  {
    ...
  }
}
```

The **EventBroker** class uses the **PublishedEvent** class shown in the following code sample.

C#

```
public class PublishedEvent
{
  private List<object> publishers;
  private List<EventHandler> subscribers;

  ...

  public IEnumerable<object> Publishers
  {
    get { ... }
  }

  public IEnumerable<EventHandler> Subscribers
  {
    get { ... }
  }

  public void AddPublisher(object publisher, string eventName)
  {
    ...
  }

  public void RemovePublisher(object publisher, string eventName)
  {
    ...
  }

  public void AddSubscriber(EventHandler subscriber)
  {
    ...
  }

  public void RemoveSubscriber(EventHandler subscriber)
  {
    ...
  }

  private void OnPublisherFiring(object sender, EventArgs e)
  {
    foreach(EventHandler subscriber in subscribers)
    {
      subscriber(sender, e);
    }
  }
}
```

The **SimpleEventBroker** extension must create and populate an **EventBroker** instance when you register and resolve types from the container that use the **Publishes** and **SubscribesTo** attributes.

Implementing the Container Extension

You may find it useful to open the sample application, "EventBroker," that accompanies this guide in Visual Studio while you read this section.

The first step to implement a container extension is to extend the abstract **UnityContainerExtension** class and to override the **Initialize** method as shown in the following code sample.

C#

```
public class SimpleEventBrokerExtension :
  UnityContainerExtension, ISimpleEventBrokerConfiguration
{
  private readonly EventBroker broker = new EventBroker();

  protected override void Initialize()
  {
    Context.Container.RegisterInstance(broker);

    Context.Strategies.AddNew<EventBrokerReflectionStrategy>(
      UnityBuildStage.PreCreation);
    Context.Strategies.AddNew<EventBrokerWireupStrategy>(
      UnityBuildStage.Initialization);
  }

  public EventBroker Broker
  {
    get { return broker; }
  }
}
```

The **Intialize** method first registers an instance of the **EventBroker** class shown in the previous section with the container. The **Initialize** method then adds two new strategies to the container: an **EventBrokerReflectionStrategy** and an **EventBrokerWireUpStrategy**.

You can add strategies to the container that take effect at different stages in the container's activities as it builds up an object instance. The **EventBrokerReflectionStrategy** strategy takes effect at the **PreCreation** stage: during this stage, the container is using reflection to discover the constructors, properties, and methods of the type currently being resolved. The **EventBrokerWireUpStrategy** strategy takes effect at the **Initialization** stage: during this stage, the container performs property and method injection on the object currently being instantiated by the container.

The following table summarizes the different stages where you can add your custom strategies to the container.

Stage	Description
Setup	The first stage. By default, nothing happens here.
TypeMapping	The second stage. Type mapping takes place here.

Lifetime	The third stage. The container checks for a lifetime manager.
PreCreation	The fourth stage. The container uses reflection to discover the constructors, properties, and methods of the type being resolved.
Creation	The fifth stage. The container creates the instance of the type being resolved.
Initialization	The sixth stage. The container performs any property and method injection on the instance it has just created.
PostInitialization	The last stage. By default, nothing happens here.

If the container discovers a lifetime manager during the **Lifetime** phase that indicates that an instance already exists, then the container skips the remaining phases. You can see this happening in Figure 1, the sequence diagram that shows how the custom lifetime manager works.

The strategy classes that you add to the list of strategies at each stage all extend the **BuilderStrategy** class. In these strategy classes, you typically override the **PreBuildUp** method to modify the way that the container builds the object it is resolving, although you can also override the **PostBuildUp** method to modify the object after the container has completed its work and the **PreTearDown** and **PostTearDown** methods if you need to modify the way the container removes the object.

Discovering the Publishers and Subscribers

In the event broker example, during the **PreCreation** stage, **EventBrokerReflectionStrategy** strategy scans the type the container will create for events decorated with the **Publishes** attribute and methods decorated with the **SubscribesTo** attribute and stores this information in an **EventBrokerInfoPolicy** object as shown in the following code sample.

C#

```
public class EventBrokerReflectionStrategy : BuilderStrategy
{
  public override void PreBuildUp(IBuilderContext context)
  {
    if (context.Policies.Get<IEventBrokerInfoPolicy>
      (context.BuildKey) == null)
    {
      EventBrokerInfoPolicy policy = new EventBrokerInfoPolicy();
      context.Policies.Set<IEventBrokerInfoPolicy>(policy, context.BuildKey);

      AddPublicationsToPolicy(context.BuildKey, policy);
      AddSubscriptionsToPolicy(context.BuildKey, policy);
    }
  }

  private void AddPublicationsToPolicy(NamedTypeBuildKey buildKey,
    EventBrokerInfoPolicy policy)
  {
```

```
    Type t = buildKey.Type;
    foreach(EventInfo eventInfo in t.GetEvents())
    {
      PublishesAttribute[] attrs =
      (PublishesAttribute[])eventInfo.GetCustomAttributes(
        typeof(PublishesAttribute), true);
      if(attrs.Length > 0)
      {
        policy.AddPublication(attrs[0].EventName, eventInfo.Name);
      }
    }
  }

  private void AddSubscriptionsToPolicy(NamedTypeBuildKey buildKey,
    EventBrokerInfoPolicy policy)
  {
    foreach(MethodInfo method in buildKey.Type.GetMethods())
    {
      SubscribesToAttribute[] attrs =
      (SubscribesToAttribute[])
      method.GetCustomAttributes(typeof(SubscribesToAttribute), true);
      if(attrs.Length > 0)
      {
        policy.AddSubscription(attrs[0].EventName, method);
      }
    }
  }
}
```

Wiring Up the Publishers and Subscribers

During the **Initialization** stage, the **EventBrokerWireUpStrategy** strategy retrieves the policy information stored by the **EventBrokerReflectionStrategy** strategy during the **PreCreation** stage and populates the **EventBroker** object with information about the events and publishers that the subscribers subscribe to. The following code sample shows how the **EventBrokerWireUpStrategy** strategy implements this final step in the event broker extension.

C#

```
public class EventBrokerWireupStrategy : BuilderStrategy
{
  public override void PreBuildUp(IBuilderContext context)
  {
    if (context.Existing != null)
    {
      IEventBrokerInfoPolicy policy =
      context.Policies.Get<IEventBrokerInfoPolicy>(context.BuildKey);
      if(policy != null)
      {
```

```
        EventBroker broker = GetBroker(context);
        foreach(PublicationInfo pub in policy.Publications)
        {
          broker.RegisterPublisher(pub.PublishedEventName,
            context.Existing, pub.EventName);
        }
        foreach(SubscriptionInfo sub in policy.Subscriptions)
        {
          broker.RegisterSubscriber(sub.PublishedEventName,
          (EventHandler)Delegate.CreateDelegate(
          typeof(EventHandler),
          context.Existing,
          sub.Subscriber));
        }
      }
    }
  }

  private EventBroker GetBroker(IBuilderContext context)
  {
    var broker = context.NewBuildUp<EventBroker>();
    if(broker == null)
    {
      throw new InvalidOperationException("No event broker available");
    }
    return broker;
  }
}
```

Summary

This chapter described two ways that you can extend Unity. First, it described how to create custom lifetime managers that enable you to modify the way that Unity stores and tracks the instances of registered types created by the container. Second, it described how you can create container extensions that enable you to add custom behavior into the build-up process that occurs when the container resolves a type and creates a new instance.

The Unity source code is a great reference for implementations of both lifetime managers and container extensions because Unity uses both of these mechanisms to implement some of its standard functionality.

Chapter 7 - Summary

The goal of this guide was to introduce you to dependency injection, interception, and the Unity application block from Enterprise Library. Dependency injection and interception are widely used techniques that help developers write maintainable, testable, flexible, and extensible code, especially for large, enterprise applications.

Unity is not the only tool you can use to add dependency injection and interception to your applications, but if you have read this guide it should be clear that Unity is easy to use and easy to extend. You can use Unity in a wide range of application types such as desktop, web, WCF, and cloud.

Chapters 2 and 3 of this guide focused on the topic of dependency injection, while Chapters 4 and 5 focused on interception. Chapter 6 described a number of ways that you can extend Unity itself if you find that out-of-the-box it doesn't support a specific scenario.

The remaining parts of this guide, "Tales from the Trenches," offer a different perspective: these chapters offer brief case studies that describe real-world use of Unity. They make clear the range of scenarios in which you can use Unity, and also highlight its ease of use and flexibility.

We hope you find Unity adds significant benefits to your applications and helps you to achieve those goals of maintainability, testability, flexibility, and extensibility in your own projects.

Tales from the Trenches: Using Unity in a Windows Store app

Case study provided by David Britch

AdventureWorks Shopper

The AdventureWorks Shopper reference implementation is a Windows Store business app, which uses Prism for the Windows Runtime to accelerate app development. It provides guidance to developers who want to create a Windows Store business app using C#, Extensible Application Markup Language (XAML), the Windows Runtime, and modern development practices.

The reference implementation shows how to:

- Implement pages, controls, touch, navigation, settings, suspend/resume, search, tiles, and tile notifications.
- Implement the Model-View-ViewModel (MVVM) pattern.
- Validate user input for correctness.
- Manage application data.
- Test your app and tune its performance.

Prism for the Windows Runtime accelerates app development by providing core services commonly required by a Windows Store app. These core services include providing support for bootstrapping MVVM apps, state management, validation of user input, navigation, event aggregation, data binding, commands, Flyouts, settings, and search.

While Prism for the Windows Runtime does not require you to use a dependency injection container, for AdventureWorks Shopper we chose to use the Unity dependency injection container. AdventureWorks Shopper uses the MVVM pattern, and in the context of a Windows Store app that uses the MVVM pattern, there are specific advantages to using Unity:

- It can be used to register and resolve view models and views.
- It can be used to register services, and inject them into view models.
- It can be used to create view models and inject the views.

In AdventureWorks Shopper we used Unity to manage the instantiation of view model and infrastructure service classes. Only one class holds a reference to the Unity dependency injection container. This class instantiates the **UnityContainter** object and registers instances and types with the container.

The main reason for using Unity is that it gave us the ability to decouple our concrete types from the code that depends on those types. During an object's creation, the container injects any dependencies that the object requires into it. If those dependencies have not yet been created, the container creates and resolves them first.

Using Unity has resulted in several advantages for AdventureWorks Shopper:

- It removed the need for a component to locate its dependencies and manage their lifetime.
- It allowed mapping of implemented dependencies without affecting the components.
- It facilitated testability by allowing dependencies to be mocked.
- It increased maintainability by allowing new components to be easily added to the system.

Overall, using Unity in a Windows Store business app has resulted in a much cleaner architecture and has helped to further our separation of concerns.

References

For further information about the AdventureWorks Shopper reference implementation, and Prism for the Windows Runtime, see:

- Developing a business app for the Windows Store using C#: AdventureWorks Shopper
- Prism for the Windows Runtime reference
- Prism StoreApps library
- Prism PubSubEvents library

Tales from the Trenches: One User's Story - Customizing Unity

Case study provided by Dan Piessens

My use of Unity started with attending the first Unity workshop hosted by the p&p group post 1.0 release. Admittedly, those were the early days of Dependency Injection and Inversion of Control (IoC) and I was completely off base as to my understanding of the concepts. However, as members of the Unity team began asking people in the room for their use cases, I began to understand how Unity was not only a well-designed IoC container, but extremely flexible and lightweight in its implementation.

My first stab at customizing Unity was to have an **ILogger** interface in our application that automatically set its logger name to the class it was being injected into. I looked at the extension interface; it looked simple enough, just have a class implement the extension base class and you're good to go right? Well it's a little more complicated than that.

First, let's take a look at what we wanted the logger to do. Most loggers require something to identify where a log message is coming from such as a category, a source, or a name. More often than not, a fully qualified class name is used to enable a user to then filter by namespace. Since the value of the category is static, after you have injected the logger, it becomes redundant to include it on every log line or to resolve it on every call; therefore, factoring it into dependency injection makes life easier for the developer. An example of this would be as follows:

C#

```
class MyClass
{
  private readonly ILogger _logger;

  public MyClass(ILogger logger)
  {
    _logger = logger;
  }

  public string SayHello(string name)
  {
    // Logger name here will be 'MyApplication.MyClass'
    _logger.LogDebug("Saying Hello to {0}", name);

    return string.Format("Hello {0}", name);
  }
}
```

The first critical thing to understand is that Unity includes a sequence of *build stages* that are similar to a road map: each stage in this pipeline facilitates Unity in building the object that needs to be resolved. Some pipeline stages run each time an object is constructed, others only run the first time the container

needs to build an object. Because of this, you need to plan how your customization fits in to this build pipeline, and which stages you need to modify to create the instance you need. There was some trial and error but in the end, it was fairly easy to modify the correct stages for the custom logger.

The next step is to think about how your extension will store or discover the additional data it needs; most of this revolves around the *build context*. The build context, which is similar to a SQL query plan, provides information to the pipeline on subsequent builds of the object and indicates what information the pipeline can be cached and what it needs to determine on each build. The Unity team made this straightforward with the build context so it's not nearly as error prone as with some other IoC containers.

For this extension, I created the *LoggerNameResolverPolicy*. This policy is ultimately what constructs and injects the logger into the target class. It also stores the logger name, which in this case is the parent class name. Now, you might ask why we called it the "parent?" One if the tricky points to grasp is that the build plan is actually for the logger, not the class that you are injecting the logger into. The policy class looks like this:

C#

```
public sealed class LoggerNameResolverPolicy : IDependencyResolverPolicy
{
  private readonly Type _type;
  private readonly string _name;
  private readonly string _parentClassName;

  public LoggerNameResolverPolicy(Type type, string name,
    string parentClassName)
  {
    _type = type;
    _name = name;
    _parentClassName = parentClassName;
  }

  public object Resolve(IBuilderContext context)
  {
    var lifetimeManager = new ContainerControlledLifetimeManager();
    lifetimeManager.SetValue(_parentClassName);
    var loggerNameKey =
      new NamedTypeBuildKey(typeof(string), "loggerName");


    //Create the build context for the logger
    var newKey = new NamedTypeBuildKey(_type, _name);

    //Register the item as a transient policy
    context.Policies.Set<IBuildKeyMappingPolicy>(
      new BuildKeyMappingPolicy(loggerNameKey), loggerNameKey);
    context.Policies.Set<ILifetimePolicy>(
```

```
      lifetimeManager, loggerNameKey);
    context.Lifetime.Add(lifetimeManager);

    try
    {
      return context.NewBuildUp(newKey);
    }
    finally
    {
      context.Lifetime.Remove(lifetimeManager);
      context.Policies.Clear<IBuildKeyMappingPolicy>(loggerNameKey);
      context.Policies.Clear<ILifetimePolicy>(loggerNameKey);
    }
  }
}
```

I could refactor this implementation of the resolver policy class to use the parameter override support in Unity 2.0 and 3.0, but this code worked with Unity 1.0 and still works today. I wanted to show how even though Unity has evolved, older plugins such as this continue to work. If you're wondering what the code is doing, it is using the concept of a "transient policy" in Unity. The policy only exists while the container is building the object and is then it is disposed. The interesting part is how the parent policy is cached with the object so whether you need one instance or a thousand, the container only creates the policy once for the class.

The next step was to capture the logger name whenever a property or constructor parameter asks for it. This was fairly simple as Unity has specific policies that run whenever it resolves a property or constructor argument (methods too but we don't use method resolution). These are represented by interfaces **IPropertySelectorPolicy** and **IConstructorSelectorPolicy**. There are also abstract implementations that are generally very useful. In our case however, we had the need to create resolver implementations that used type metadata in several places and wanted to abstract out that portions of the property or constructor selection policy with our own that passed in the *buildContext* variable and the constructing type. The following code example shows how we overrode the default constructor selector policy to suit our needs. The primary modification is in the **CreateResolver** method, passing in the build context to look for our own policies and using the build key to capture the type being created.

C#

```
public class ContainerConstructorSelectorPolicy : IConstructorSelectorPolicy
{
  public virtual SelectedConstructor SelectConstructor(
    IBuilderContext context, IPolicyList resolverPolicyDestination)
  {
    // Same as default implementation...
  }

  private static SelectedConstructor CreateSelectedConstructor(
    IBuilderContext context, ConstructorInfo ctor,
```

```
    IPolicyList resolverPolicyDestination)
  {
    var result = new SelectedConstructor(ctor);


    foreach (var param in ctor.GetParameters())
    {
      var key = Guid.NewGuid().ToString();
      var policy = CreateResolver(context, param);
      resolverPolicyDestination.Set(policy, key);
      DependencyResolverTrackerPolicy.TrackKey(context.PersistentPolicies,
        context.BuildKey, key);
      result.AddParameterKey(key);
    }
    return result;
  }

  private static IDependencyResolverPolicy CreateResolver(
    IBuilderContext context, ParameterInfo parameterInfo)
  {
    var policy = context.Policies
     .Get<IParameterDependencyResolver>(
     new NamedTypeBuildKey(parameterInfo.ParameterType));

    IDependencyResolverPolicy resolverPolicy = null;

    if (policy != null)
    {
      resolverPolicy = policy.CreateResolver(context.BuildKey.Type,
        parameterInfo);
    }

    return resolverPolicy ?? new FixedTypeResolverPolicy(
      parameterInfo.ParameterType);
  }

  private static ConstructorInfo FindInjectionConstructor(Type typeToConstruct)
  {
    // Same as default implementation...
  }

  private static ConstructorInfo FindLongestConstructor(Type typeToConstruct)
  {
    // Same as default implementation...
  }
}
```

Once we had these main build policies in place, we needed to implement our new interfaces **IPropertyDependencyResolver** and **IParameterDependencyResolver**. The signatures are almost

identical, except that one passes in parameter arguments and the other passes in constructor arguments. For simplicity, I'll only show the constructor resolver, but the pattern is the same for a parameter resolver or even a method resolver if you choose to support them.

C#

```
public class LoggerNameConstuctorParameterPolicy : IParameterDependencyResolver
{
  public IDependencyResolverPolicy CreateResolver(
    Type currentType, ParameterInfo param)
  {
    return new LoggerNameResolverPolicy(param.ParameterType, null,
      currentType.FullName);
  }
}
```

The final module wire up is easy; in fact, most of the items don't need additional configuration. We had a few errors on the first try, but the debugging experience made those errors very descriptive. The fact that Unity is open source also helped greatly, there were a few times when I peeked at the default implementation code to get hints about the right way to construct things. The Unity guides that ship with this release will also prove to be a big help with customization. The code to create the extension itself simply involved registering the policies into the parameter and constructor discovery stages and indicating that this should occur for any **ILogger** item. It also included a section to override the default constructor selector and property selector policies with our custom ones. In the end we moved this code to a separate extension so it could be better reused. The final registration code looked like this:

C#

```
public class LoggerNameExtension : UnityContainerExtension
{
  protected override void Initialize()
  {
    // Override base Unity policies.
    Context.Policies.ClearDefault<IConstructorSelectorPolicy>();
    Context.Policies.SetDefault<IConstructorSelectorPolicy>(
      new ContainerConstructorSelectorPolicy());
    Context.Policies.SetDefault<IPropertySelectorPolicy>(
      new ContainerPropertySelectorPolicy());

    // Set logging specific policies
    var buildKey = new NamedTypeBuildKey(typeof(ILogger));
    Context.Policies.Set<IParameterDependencyResolver>(
      new LoggerNameConstuctorParameterPolicy(), buildKey);
    Context.Policies.Set<IPropertyDependencyResolver>(
      new LoggerNamePropertyPolicy(), buildKey);
  }
}
```

Emboldened by our initial success, my team and I went on to create several more Unity extensions; some added additional dependency resolution attribute options, others effectively created specialized factory instances of objects. Over the past few years, many of the extensions were retired as Unity itself expanded to support factory methods, alternate registration attributes, and most recently **Lazy<T>** item support and registration by convention. The only real customization needed today is for scenarios such as the logger, where metadata is pulled from the code that calls it.

Appendix A - Unity and Windows Store apps

When you create applications for Windows 8, those applications are known as Windows Store apps. You can develop Windows Store apps using C#, Visual Basic, JavaScript, and C++.

You can use Unity when you develop Windows Store apps using C# and Visual Basic. However, there are some limitations in the functionality of the version of Unity that targets the .NET for Windows Store apps version of the .NET Framework; this appendix describes those limitations.

The UnityServiceLocator Class

You cannot use the **Microsoft.Practices.Unity.UnityServiceLocator** class with the .NET for Windows Store apps version of the .NET Framework. This class depends on the Common Service Locator library, which is not available for Windows Store apps.

Unity Design-time Configuration

You cannot configure your Unity container using a configuration file such as app.config. Windows Store apps must configure Unity containers programmatically.

This is because the Unity.Configuration assembly is not compatible with Windows Store apps.

Unity Interception

Unity Interception is not available to Windows Store apps.

This is because the Unity.Interception and Unity.Interception.Configuration assemblies are not compatible with Windows Store apps.