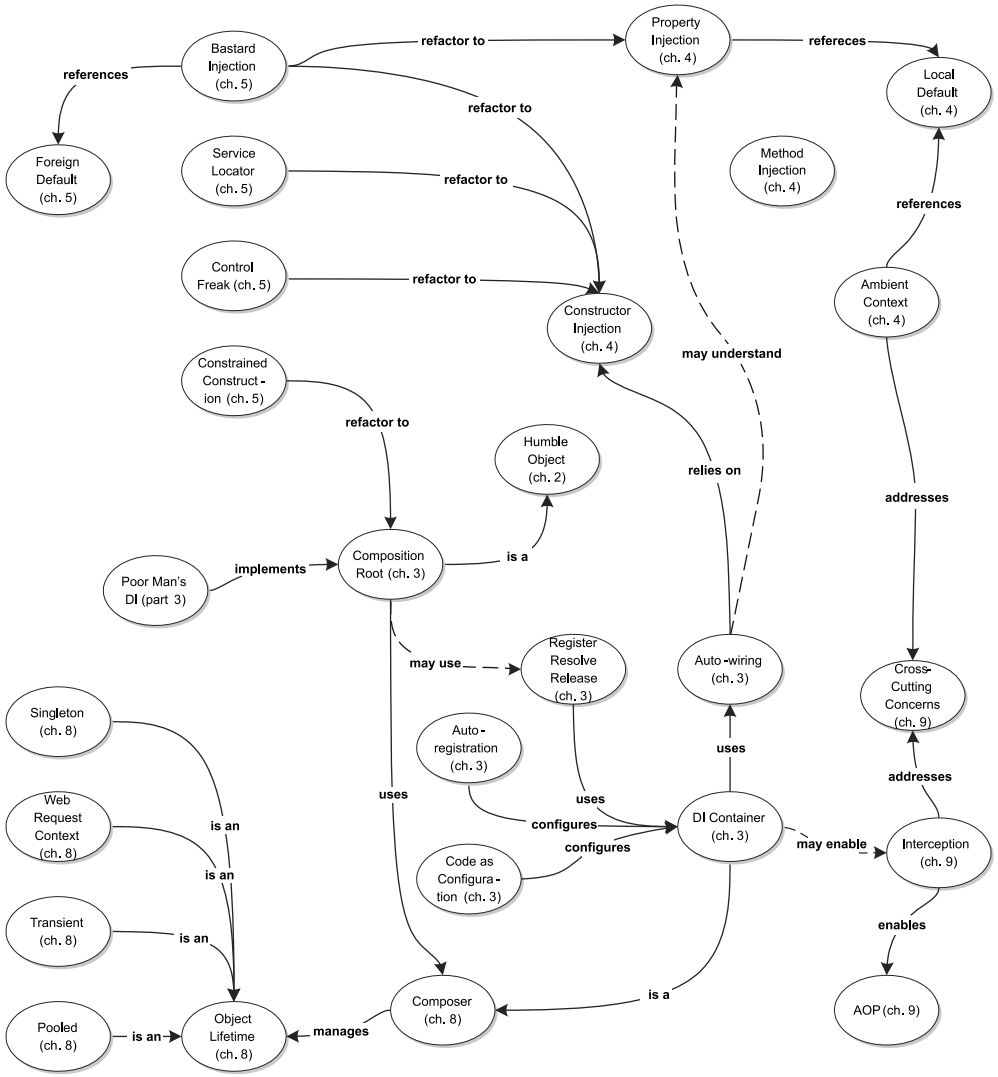
Dependency Injection in .NET

Mark Seemann

Foreword by Glenn Block

MANNING

GLOSSARY CONCEPTUAL MAP



This figure maps out how the important concepts and terms in this book relate to each other, and provides a reference to the chapters where they are covered. There's also a Glossary in the back of the book with one-sentence descriptions of each term.

Dependency Injection in .NET

Dependency Injection in .NET

MARK SEEMANN

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Cynthia Kane
Copyeditors: June Eding, Tiffany Taylor
Proofreader: Katie Tennant
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781935182504
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 17 16 15 14 13 12 11

To Cecilie
I couldn't have done it without you

brief contents

PART 1 PUTTING DEPENDENCY INJECTION ON THE MAP 1

1 ■ A Dependency Injection tasting menu 3
2 ■ A comprehensive example 29
3 ■ DI Containers 58

PART 2 DI CATALOG 93

4 ■ DI patterns 95
5 ■ DI anti-patterns 133
6 ■ DI refactorings 162

PART 3 DIY DI 197

7 ■ Object Composition 199
8 ■ Object Lifetime 236
9 ■ Interception 275

PART 4 DI CONTAINERS 311

10 ■ Castle Windsor 313
11 ■ StructureMap 347
12 ■ Spring.NET 385
13 ■ Autofac 417
14 ■ Unity 448
15 ■ MEF 492

contents

foreword xvii
preface xix
acknowledgments xxi
about this book xxiv
about the cover illustration xxix

PART 1 PUTTING DEPENDENCY INJECTION ON THE MAP ...1

1 *A Dependency Injection tasting menu 3*

1.1 Writing maintainable code 5

Unlearning DI 5 ▪ Understanding the purpose of DI 8

1.2 Hello DI 13

Hello DI code 13 ▪ Benefits of DI 15

1.3 What to inject and what not to inject 22

Seams 22 ▪ Stable Dependencies 23
Volatile Dependencies 23

1.4 DI scope 24

Object Composition 25 ▪ Object Lifetime 26
Interception 26 ▪ DI in three dimensions 27

1.5 Summary 28

2 *A comprehensive example 29*

2.1 Doing it wrong 30

Building a tightly coupled application 31
Smoke test 36 ▪ Evaluation 37 ▪ Analysis 39

2.2 Doing it right 41

Rebuilding the commerce application 43 ▪ Analyzing the loosely coupled implementation 51

2.3 Expanding the sample application 53

Architecture 53 ▪ Basket feature 54

2.4 Summary 57

3 *DI Containers 58*

3.1 Introducing DI Containers 61

Hello container 62 ▪ Auto-wiring 64

3.2 Configuring DI Containers 67

Configuring containers with XML 68 ▪ Configuring containers with code 70 ▪ Configuring containers by convention 72

3.3 DI Container patterns 75

Composition Root 75 ▪ Register Resolve Release 81

3.4 DI Container landscape 87

Selecting a DI Container 87 ▪ Microsoft and DI 89

3.5 Summary 91

PART 2 DI CATALOG.....93

4 *DI patterns 95*

4.1 Constructor Injection 98

How it works 98 ▪ When to use it 99 ▪ Known use 100
Example: Adding a currency provider to the shopping basket 101
Related patterns 103

4.2 Property Injection 104

How it works 104 ▪ When to use it 105 ▪ Known use 107
Example: Defining a currency profile service for the BasketController 108 ▪ Related patterns 110

4.3 Method Injection 111

How it works 111 ▪ When to use it 112 ▪ Known use 113
Example: Converting baskets 114 ▪ Related patterns 117

4.4 Ambient Context 118

How it works 118 ▪ When to use it 120 ▪ Known use 123
Example: Caching Currency 123 ▪ Related patterns 130

4.5 Summary 131

5 *DI anti-patterns 133*

5.1 Control Freak 136

Example: newing up Dependencies 136 ▪ Example: Factory 137
Analysis 143

5.2 Bastard Injection 144

Example: ProductService with Foreign Default 144
Analysis 146

5.3 Constrained Construction 149

Example: late-binding ProductRepository 149
Analysis 151

5.4 Service Locator 154

Example: ProductService using a Service Locator 156
Analysis 157

5.5 Summary 160

6 *DI refactorings 162*

6.1 Mapping runtime values to Abstractions 163

Abstractions with runtime Dependencies 164
Example: selecting a routing algorithm 166
Example: using a CurrencyProvider 168

6.2 Working with short-lived Dependencies 170

Closing connections through Abstractions 170
Example: invoking a product-management service 173

6.3 Resolving cyclic Dependencies 175

Addressing Dependency cycles 176
Example: composing a window 178

6.4 Dealing with Constructor Over-injection 182

Recognizing and addressing Constructor Over-injection 182
Example: refactoring order reception 185

6.5 Monitoring coupling 188

Unit-testing coupling 189 ▪ Integration-testing coupling 191
Using NDepend to monitor coupling 193

6.6 Summary 195

PART 3 DIY DI.....197

7 *Object Composition 199*

7.1 Composing console applications 202

Example: updating currencies 202

7.2 Composing ASP.NET MVC applications 206

ASP.NET MVC extensibility 206 ▪ Example: implementing CommerceControllerFactory 208

7.3 Composing WCF applications 210

WCF extensibility 211 ▪ Example: wiring up a product-management service 212

7.4 Composing WPF applications 219

WPF Composition 219 ▪ Example: wiring up a product-management rich client 220

7.5 Composing ASP.NET applications 224

ASP.NET composition 224 ▪ Example: wiring up a CampaignPresenter 225

7.6 Composing PowerShell cmdlets 230

Example: composing basket-management cmdlets 231

7.7 Summary 235

8 *Object Lifetime 236*

8.1 Managing Dependency Lifetime 239

Introducing Lifetime Management 239
Managing lifetime with a container 242

8.2 Working with disposable Dependencies 247

Consuming disposable Dependencies 248
Managing disposable Dependencies 251

8.3 Lifestyle catalog 255

Singleton 255 ▪ Transient 258 ▪ Per Graph 259
Web Request Context 261 ▪ Pooled 266 ▪ Other lifestyles 271

8.4 Summary 273

9 *Interception 275*

9.1 Introducing Interception 277

Example: implementing auditing 277 ▪ Patterns and principles for Interception 281

9.2 Implementing Cross-Cutting Concerns 285

Intercepting with a Circuit Breaker 286
Handling exceptions 292 ▪ Adding security 293

9.3 Declaring aspects 295

Using attributes to declare aspects 296
Applying dynamic Interception 300
Example: intercepting with Windsor 303

9.4 Summary 308

PART 4 DI CONTAINERS 311

10 Castle Windsor 313

10.1 Introducing Castle Windsor 314

Resolving objects 315 ▪ Configuring the container 317
Packaging configuration 322

10.2 Managing lifetime 323

Configuring lifestyle 324 ▪ Using advanced lifestyles 325
Developing a custom lifestyle 327

10.3 Working with multiple components 333

Selecting among multiple candidates 333
Wiring sequences 336 ▪ Wiring Decorators 339

10.4 Configuring difficult APIs 341

Configuring primitive Dependencies 341
Registering components with code blocks 343
Wiring with Property Injection 344

10.5 Summary 345

11 StructureMap 347

11.1 Introducing StructureMap 348

Resolving objects 350 ▪ Configuring the container 352
Packaging configuration 358

11.2 Managing lifetime 361

Configuring lifestyles 362 ▪ Developing a custom lifestyle 364

11.3 Working with multiple components 370

Selecting among multiple candidates 371
Wiring sequences 374 ▪ Wiring Decorators 377

11.4 Configuring difficult APIs 380

Configuring primitive Dependencies 380 ▪ Creating objects with code blocks 381 ▪ Wiring with Property Injection 382

11.5 Summary 383

12 *Spring.NET 385*

12.1 Introducing Spring.NET 386

Resolving objects 387 ▪ Configuring the container 389 Loading XML 393

12.2 Managing lifetime 397

Configuring object scopes 398

12.3 Working with multiple components 399

Selecting among multiple candidates 400 Wiring sequences 402 ▪ Wiring Decorators 405 Creating Interceptors 407

12.4 Configuring difficult APIs 412

Configuring primitive Dependencies 412 ▪ Configuring static factories 413 ▪ Wiring with Property Injection 414

12.5 Summary 416

13 *Autofac 417*

13.1 Introducing Autofac 418

Resolving objects 420 ▪ Configuring the ContainerBuilder 422 Packaging configuration 427

13.2 Managing lifetime 429

Configuring instance scope 430

13.3 Working with multiple components 433

Selecting among multiple candidates 434 Wiring sequences 438 ▪ Wiring Decorators 440

13.4 Registering difficult APIs 442

Configuring primitive Dependencies 443 ▪ Registering objects with code blocks 444 ▪ Wiring with Property Injection 445

13.5 Summary 447

14 *Unity 448*

14.1 Introducing Unity 450

Resolving objects 451 ▪ Configuring the container 453 Packaging configuration 458

14.2 Managing lifetime 459

Configuring lifetime 460 ▪ Developing a custom lifetime 464

14.3 Working with multiple components 473

Selecting among multiple candidates 473
Wiring sequences 476 ▪ Wiring Decorators 479
Creating Interceptors 481

14.4 Configuring difficult APIs 486

Configuring primitive Dependencies 486
Registering components with code blocks 487
Wiring with Property Injection 489

14.5 Summary 490

15 ***MEF 492***

15.1 Introducing MEF 495

Resolving objects 496 ▪ Defining imports and exports 499
Working with catalogs 504

15.2 Managing lifetime 509

Declaring creation policy 509 ▪ Releasing objects 511

15.3 Working with multiple components 513

Selecting among multiple candidates 513
Wiring sequences 516 ▪ Wiring Decorators 519

15.4 Composing difficult APIs 521

Compositing primitive parts 521 ▪ Composing parts with non-public constructors 522 ▪ Wiring with Property Injection 523

15.5 Summary 524

resources 526
glossary 531
index 535

foreword

My first experience with Dependency Injection was almost 10 years ago. I was working at an ISV (independent software vendor) as an architect on an enterprise framework building LOB (line-of-business) applications. In those days, it seemed like all my friends in the industry were building similar frameworks. The framework supported various layers across *n*-tier applications addressing data access, business, and UI concerns. The product-supported business objects could be persisted across multiple databases and represented in multiple UIs; the challenge was finding a way to build the system to make it extensible and maintainable. We found our answer by wading into the waters of Dependency Injection. Using a DI approach, we clearly defined contracts for the layers, allowing us to more easily test the layers as well as to swap their implementations without breaking the code.

Mark talks quite a bit in this book about "poor man's DI" and this is exactly what we were doing. In those days, we didn't have DI containers at our disposal. We also didn't have the type of guidance you'll find in this book. As a result, we made a lot of mistakes—mistakes you won't have to make.

In the past four years, I've personally worked with hundreds of customers and I'm aware of thousands that have found success using the techniques described in this book.

It all starts with patterns.

DI containers are just tools. The tools are only useful if you're building systems that incorporate the patterns that the tools are addressing. They aren't the solution to every problem. Ideally, you need to first learn what Dependency Injection is, what kinds of problems it solves, and what the patterns are for using it. Then you can look at the various tools as aids in applying those patterns.

This book will help you with all of the above. The early chapters present an overview of the general problems that occur when software is tightly coupled. The book then discusses ways we can apply various techniques, both simple and advanced, to address those problems. Along the way, the book classifies various patterns and identifies when they are most appropriate for specific situations. In the second half, the book presents a comprehensive overview of the most common DI containers/frameworks in .NET and explains how to use them to apply different techniques.

With this book, you will benefit from the knowledge of someone who has many years of real-world experience in applying these techniques. This is a real treat; often, those who start using DI quickly find themselves lost in a sea of confusion. This book addresses any potential misunderstanding, starting with basic questions like, "Where should I put my IoC?" or "Should I expose my container?" Mark covers these questions and many more.

Throughout the book, Mark not only describes the techniques but really goes into depth explaining when you should—and, more importantly—*shouldn't* use them. When he describes a problem, he uses realistic examples to keep the big picture in focus.

If you are new to IoC, I believe you'll find *Dependency Injection in .NET* to be a great resource for learning. Even if you have extensive experience with IoC, you'll still benefit from the painstaking work Mark has done to classify various patterns and create a taxonomy for IoC. I also think that you will find his comparisons with other IoC containers beneficial.

Regardless of your level of experience, I wish you success with this book.

GLENN BLOCK
SENIOR PROGRAM MANAGER
MICROSOFT

preface

There's a peculiar phenomenon related to Microsoft called the *Microsoft Echo Chamber*. Microsoft is a huge organization and the surrounding ecosystem of Microsoft Certified Partners multiplies that size by orders of magnitude. If you're sufficiently embedded in this ecosystem, it can be hard to see past its boundaries. Whenever you look for a solution to a problem with a Microsoft product or technology, you're likely to find an answer that involves throwing even more Microsoft products at it. No matter what you yell within the echo chamber, the answer is *Microsoft!*

When Microsoft hired me in 2003, I was already firmly embedded in the echo chamber, having worked for Microsoft Certified Partners for years—and I loved it! They soon shipped me off to an internal tech conference in New Orleans to learn about the latest and greatest Microsoft technology.

Today, I can't recall any of the Microsoft product sessions I attended—but I do remember the last day. On that day, having failed to experience any sessions that could satisfy my hunger for cool tech, I was mostly looking forward to flying home to Denmark. My top priority was to find a place to sit so I could attend to my email, so I chose a session that seemed marginally relevant for me and fired up my laptop.

The session was loosely structured and featured several presenters. One was a bearded guy named Martin Fowler, who talked about Test-Driven Development (TDD) and dynamic mocks. I had never heard of him and I didn't listen very closely, but, nonetheless, something must have stuck in my mind.

Soon after returning to Denmark, I was tasked with rewriting a big ETL (extract, transform, load) system from scratch, and I decided to give TDD a try (it turned out to

be a *very* good decision). The use of dynamic mocks followed naturally, but also introduced a need to manage dependencies. I found that to be a very difficult but very captivating problem, and I couldn't stop thinking about it.

What started as a side effect of my interest in TDD became a passion in itself. I did a lot of research, read lots of blog posts about the matter, wrote quite a few blogs myself, experimented with code, and discussed the topic with anyone who cared to listen. Increasingly, I had to look outside the Microsoft Echo Chamber for inspiration and guidance. Along the way, people associated me with the ALT.NET movement even though I was never very active in it.

I made all the mistakes it was possible to make, but I was gradually able to develop a coherent understanding of Dependency Injection (DI).

When Manning approached me with the idea for a book about Dependency Injection in .NET my first reaction was, *Is this even necessary?* I felt that all the concepts you need to understand DI were already described in numerous blog posts. Was there anything to add? Honestly, I thought DI in .NET was a topic that had been done to death already.

Upon reflection, however, it dawned on me that while the *knowledge* is definitely out there, it's *very* scattered and uses a lot of conflicting terminology. Before this book, there were no titles about DI that attempted to present a coherent description of it. After thinking about it further, I realized that Manning was offering me a tremendous challenge and a great opportunity to collect and systematize all that I knew about DI.

The result is this book. It uses .NET and C# to introduce and describe a comprehensive terminology and guidance for DI, but I hope that the value of the book will reach well beyond the platform. I think that the pattern language articulated here is universal. Whether you are a .NET developer or use another object-oriented platform, I hope that this book will help you be a better software engineer.

acknowledgments

Gratitude may seem like a cliché, but this is only because it's such a fundamental part of human nature. While I was writing the book, many people gave me good reasons to be grateful, and I would like to thank them all.

First of all, writing a book in my spare time has given me a new understanding of just how taxing such a project is on marriage and family life. My wife Cecilie stayed with me and actively supported me during the whole process. Most importantly, she understood just how important this project was to me. We are still together and I look forward to being able to spend more time with her and our kids Linea and Jarl (who miss me, although I've been right here all the time).

Both my parents and in-laws have also been a huge help in keeping the family running during those times when I needed to direct my efforts towards the book. I couldn't have done it without them.

On a more professional level, I wish to thank Manning for giving me this opportunity. Karen Tegtmeyer originally "discovered" me and helped me establish a relationship with Manning. Michael Stephens initiated the project and believed in me when things looked bleak. There were times when it looked like I'd never be able to finish the book by myself, but Michael took a chance with me, and I'm immensely grateful that I was allowed to complete the book as the consistent work of a single person. Cynthia Kane served as my development editor and kept a keen eye on the quality of the text. She helped me identify weak spots in the manuscript and provided extensive constructive criticism. Despite all the frustration along the way, I'm particularly grateful that she convinced me to rework chapters 1 through 3. *Kill your darlings*,

as the saying goes. I'm much happier with the final result, and I have Cynthia to thank for that.

Although writing the book was an unpaid side project, I never had any doubt that it would impact my work performance. When I started the project, my manager at the time, Peter Haastrup, was very supportive. I want to thank both him and our CEO, Niels Flensted-Jensen, for providing an inspiring and supportive work environment. Unfortunately, the company went out of business, but my new employer, Jørn Floor Andersen, has been exceptionally patient with me.

Karsten Strøbæk and Brian Rasmussen read through numerous early drafts and provided much helpful feedback. Karsten also served as the technical proofreader during production.

The following reviewers read the manuscript at various stages of development and I am grateful for their comments and insight: Christian Siegers, Amos Bannister, Rama Krishna Vavilala, Doug Ferguson, Darren Neimke, Chuck Durfee, Paul Grebenc, Lester Lobo, Jonas Bandi, Braj Panda, Alan Ruth, Timothy Binkley-Jones, Andrew Siemer, Javier Lozano, David Barkol, and Patrick Steger.

Many of the participants in the Manning Early Access Program (MEAP) also provided feedback and asked difficult questions that exposed the weak parts of the text.

I was so fortunate that the existing .NET DI CONTAINER community received the book project with a very positive attitude. Several of the specific DI CONTAINERS' creators offered to review the chapters on "their" container. Krzysztof Koźmic reviewed the Castle Windsor chapter, Stephen Bohlen the Spring.NET chapter, Nicholas Blumhardt the Autofac chapter, Chris Tavares the Unity chapter, and Glenn Block looked over the MEF chapter while Jeremy Miller answered my stupid questions via Twitter and the StructureMap forum. I'm grateful for their participation, for it provided confirmation that my way of presenting their work could be aligned with their own. I would also like to thank Glenn Block for contributing the foreword.

Mogens Heller Grabe courteously allowed me to use his picture of a hairdryer wired directly into a wall outlet, and Patrick Smacchia provided me with a copy of *NDepend* and reviewed the related section.

In many ways, Martin Gildenpfennig sowed more seeds for this book than he may realize. Even before I was (lightly) exposed to Martin Fowler's presentation of TDD back in 2003, Martin Gildenpfennig had already introduced me to the concept of unit testing, although we never got around using it at that time. Much later, I was stuck with the false conviction that SERVICE LOCATOR was a blessing, and, with a few simple sentences, he made me realize that there's a better alternative.

My former colleague, Mikkel Christensen, was a pleasure to work with while I wrote great portions of the book. We had many good discussions about API design and patterns, and I could bounce even my craziest ideas off of him and always get an open and qualified discussion out of it.

Finally, I wish to thank Thomas Jaskula for all the support and inspiration along the way. We've never had the pleasure of meeting each other, but Thomas has time and again exhibited an almost overwhelming delight with my work. He may not realize it, but there were times when this was the only thing that kept me going.

about this book

This is a book about Dependency Injection first and foremost. It's also a book about .NET, but that's much less important. C# is used for code examples, but much of the discussion in this book can be easily applied to other languages and platforms. In fact, I learned a lot of the underlying principles and patterns from reading books where Java or C++ was used in examples.

Dependency Injection (DI) is a set of related patterns and principles. It's a way to think about and design code more than it's a specific technology. The ultimate purpose of using DI is to create maintainable software within the object-oriented paradigm.

The concepts used throughout this book all relate to object-oriented programming. The problem that DI addresses (code maintainability) is universal, but the proposed solution is given within the scope of object-oriented programming in statically typed languages: C#, Java, Visual Basic .NET, C++, and so on. You can't apply DI to procedural programming, and it may not be the best solution in functional or dynamic languages.

DI in isolation is just a small thing, but it's closely interconnected with a large complex of principles and patterns for object-oriented software design. Whereas the book focuses consistently on DI from start to finish, it also discusses many of these other topics in the light of the specific perspective that DI can give. The goal of the book is more than just teaching you about DI specifics: it's to make you a better object-oriented programmer.

Who should read this book?

It would be tempting to state that this is a book for all .NET developers. However, the .NET community today is vast and spans developers working with web applications, desktop applications, smartphones, RIA, integration, office automation, content management systems, and even games. Although .NET is object-oriented, not all of those developers write object-oriented code.

This is a book about object-oriented programming, so at minimum readers should be interested in object orientation and understand what an interface is. A few years of professional experience and knowledge of design patterns or SOLID will certainly be a benefit as well. In fact, I don't expect beginners to get much out of the book; it's mostly targeted towards experienced developers and software architects.

The examples are all written in C#, so readers working with other .NET languages must be able to read and understand C#. Readers familiar with non-.NET object-oriented languages such as Java and C++ may also find the book valuable, because the .NET platform-specific content is relatively light. Personally, I read a lot of pattern books with examples in Java and still get a lot out of them, so I hope the converse is true as well.

Roadmap

The contents of this book are divided into four parts. Ideally, I'd like you to first read it from cover to cover and then subsequently use it as a reference, but I understand if you have other priorities. For that reason, a majority of the chapters are written so that you can dive right in and start reading from that point.

The first part is the major exception. It contains a general introduction to DI and is probably best read sequentially. The second part is a catalog of patterns and the like, whereas the third part is an examination of DI from three different angles. The fourth and largest part of the book is a big catalog of six DI CONTAINER libraries.

There are a lot of interconnected concepts and because I introduce them the first time it feels natural, this means that I often mention concepts before I've formally introduced them. To distinguish these universal concepts from more local terms, I consistently use SMALL CAPS to make them stand out. All these terms are briefly defined in the glossary, which also contains references to a more extensive description.

- Part 1 is a general introduction to DI. If you don't know what DI is, this is the place to start; but even if you do, you may want to familiarize yourself with the contents of part 1, as it establishes a lot of the context and terminology used in the rest of the book. Chapter 1 discusses the purpose and benefits of DI and provides a general outline. Chapter 2 contains a big and rather comprehensive example, and chapter 3 explains how DI CONTAINER libraries fit into the overall picture. Compared to the other parts, part 1 has a much more linear progression of its content. You'll need to read each chapter from the beginning to gain the most from it.
- Part 2 is a catalog of patterns, anti-patterns, and refactorings. This is where you'll find prescriptive guidance on how to implement DI, and the dangers to

look out for. Chapter 4 is a catalog of DI design patterns, and, conversely, chapter 5 is a catalog of anti-patterns. Chapter 6 contains generalized solutions to commonly occurring issues. As a catalog, each chapter contains a set of loosely related sections that are designed to be read in isolation as well as in sequence.

- Part 3 examines DI from three different angles: OBJECT COMPOSITION, LIFETIME MANAGEMENT, and INTERCEPTION. In chapter 7, I discuss how to implement DI on top of existing application frameworks such as WCF, ASP.NET MVC, WPF, and others. In many ways, you can use chapter 7 as a catalog of how to implement DI on a set of frameworks. Chapter 8 describes how to manage dependency lifetimes to avoid resources leaks. Whereas the structure is a little less stringent than previous chapters, a large part of the chapter can be used as a catalog of well-known lifetime styles. Chapter 9 finally describes how to compose applications with CROSS-CUTTING CONCERNS. This is where we harvest the benefits of all the work that came before, so, in many ways, I consider this to be the climax of the book.
- Part 4 is a catalog of DI CONTAINER libraries. Six chapters each cover a specific container in a fair amount of detail: Castle Windsor, StructureMap, Spring.NET, Autofac, Unity, and MEF. Each chapter covers its container in a rather condensed form to save space, so you may want to read about only the two or three containers that interest you the most. In many ways, I regard part 4 as a very big set of appendixes.

To keep the discussion of the DI principles and patterns free of any specific container APIs, most of the book, with the exception of part 4, is written without referencing a particular container. This is also why the containers appear with such force in part 4. It's my hope that by keeping the discussion general, the book will be useful for a longer period of time.

You can also take the concepts from parts 1 through 3 and apply them to container libraries not covered in part 4. There are good containers available that, unfortunately, I couldn't cover, but even for users of these libraries, I hope that this book has a lot to offer.

Code conventions and downloads

There are many code examples in this book. Most of it is C#, but there's also a bit of XML here and there. Source code in listings and text is in a `fixed-width font` to separate it from ordinary text.

All the source code for the book is written in C# and Visual Studio 2010. The ASP.NET MVC applications are written against ASP.NET MVC 3.

Only a few of the techniques described in this book hinge on modern language features. I started writing loosely coupled code in .NET 1.1 and I could have written most of the book's code examples on that platform without having to change my conclusions. As it were, I wanted to strike a reasonable balance between conservative and modern coding styles. When I write code professionally I use the modern

language features to a much greater degree, but here the most advanced features are generics and LINQ. The last thing I want is for you to get the idea that DI can only be applied with ultra-modern languages.

Writing code examples for a book presents its own set of challenges. Compared to a modern computer monitor, a book only allows for very short lines of code. It was very tempting to write code in a terse style with short but cryptic names for methods and variables. Such code is already difficult to understand as real code when you still have an IDE and a debugger nearby, but it becomes really difficult to follow in a book. I found it very important to keep names as readable as possible. To make it all fit, I've often had to resort to some unorthodox line breaks. All the code compiles, but sometimes the formatting looks a bit funny.

The code also makes extensive use of the `var` keyword. In my professional code I use this almost exclusively, but for written text I often find it helpful when paired with explicit declarations because the IDE isn't around to help. Still, to save space, I use `var` wherever I judge that an explicit declaration is unnecessary.

The word *class* is often used as a synonym for a *type*. In .NET, classes, structs, interfaces, enums, and so on are all *types*, but because the word *type* is also a word with a lot of overloaded meaning in ordinary language, it would often make the text less clear if used.

Most of the code in this book relates to an overarching example running through the book: an online store complete with supporting internal management applications. This is about the least exciting example you can expect to see in any software text, but I chose it for a few reasons:

- It's a well-known problem domain for most readers. Although it may seem boring, I think this is an advantage because it doesn't steal focus from DI.
- I also have to admit that I couldn't really think of any other domain that was rich enough to support all the different scenarios I had in mind.

I wrote a lot of code to support the code examples, and most of that code is not even in the book. In fact, I wrote almost all of it using Test-Driven Development (TDD), but as this isn't a TDD book, I generally don't show the unit tests in the book.

The source code for all examples in this book is available from Manning's website: http://manning.com/DependencyInjectionin.NET. The ReadMe.txt in the root of the download contains instructions for compiling and running the code.

Author Online

The purchase of *Dependency Injection in .NET* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to http://manning.com/DependencyInjectionin.NET. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It isn't a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Mark Seemann is a programmer, software architect, and speaker living in Copenhagen, Denmark. He has been working with software since 1995 and TDD since 2003, including six years with Microsoft as a consultant, developer, and architect. Mark is currently professionally engaged with software development, and is working out of Copenhagen. He enjoys reading, painting, playing the guitar, good wine, and gourmet food.

about the cover illustration

On the cover of *Dependency Injection in .NET* is "A woman from Vodnjan," a small town in the interior of the peninsula of Istria in the Adriatic Sea, off Croatia. The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Vodnjan is a culturally and historically significant town, situated on a hilltop with a beautiful view of the Adriatic and known for its many churches and treasures of sacral art. The woman on the cover wears a long black linen skirt and a short black jacket over a white linen shirt. The jacket is trimmed with blue embroidery and a blue linen apron completes the costume. The woman is also wearing a large-brimmed black hat, a flowered scarf, and big hoop earrings. Her elegant costume indicates that she is an inhabitant of the town, rather than a village. Folk costumes in the surrounding countryside are more colorful, made of wool, and decorated with rich embroidery.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few

miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Putting Dependency Injection on the map

Dependency Injection (DI) is one of the most misunderstood concepts of object-oriented programming. The confusion is abundant and spans terminology, purpose, and mechanics. Should it be called *Dependency Injection*, *Inversion of Control*, or even *Third-Party Connect?* Is the purpose of DI only to support unit testing or is there a broader purpose? Is DI the same as Service Location? Is a DI CONTAINER required?

There are plenty of blog posts, magazine articles, conference presentations, and so on that discuss DI, but, unfortunately, many of them use conflicting terminology or give bad advice. This is true across the board, and even big and influential actors like Microsoft add to the confusion.

It doesn't have to be this way. In this book I present and use a consistent terminology that I hope others will adopt. For the most part, I've adopted and clarified existing terminology defined by others, but occasionally I add a bit of terminology where none existed previously. This has helped me tremendously in evolving a specification of the scope or boundaries of DI.

One of the underlying reasons behind all the inconsistency and bad advice is that the boundaries of DI are quite blurry. Where does DI end and other object-oriented concepts begin? I think that it's impossible to draw a distinct line between DI and other aspects of writing good object-oriented code. To talk about DI we have to draw in other concepts such as SOLID and Clean Code. I don't feel that I can credibly write about DI without also touching on some of these other topics.

The first part of the book helps you understand the place of DI in relation to other facets of software engineering—putting it on the map, so to speak.

The first chapter gives you a quick tour of DI, covering its purpose, principles, and benefits, as well as providing an outline of the scope for the rest of the book. If you want to learn what DI is, and why you should be interested in it, this is the place to start. The chapter assumes you have no prior knowledge of DI, but even if you already know about it you may still want to read it—it may turn out to be something other than what you expected.

Chapter 1 is focused on the big picture and doesn't go into a lot of details. Chapter 2, on the other hand, is completely reserved for a big example. This example is intended to give you a much more concrete feel for DI. It's divided into two parts and almost shaped like a narrative. To contrast DI with a more "traditional" style of programming, the chapter first showcases a typical, tightly coupled implementation of a sample application, and then subsequently re-implements it with DI.

The third and final chapter of part 1 introduces the concept of a DI CONTAINER and explains how it fits into the overall picture of DI. I discuss DI in general terms and, although I provide code examples that demonstrate how a typical DI CONTAINER works, the purpose of the chapter isn't to explain specific API details. The main point of chapter 3 is to show that a DI CONTAINER is a (very helpful) optional tool. It's entirely possible to utilize DI without using a DI CONTAINER, so parts 2 and 3 more or less ignore DI CONTAINERS and instead discuss DI in a container-agnostic way. Then, in part 4, we return to DI CONTAINERS to dissect six specific containers.

Part 1 establishes the context for the rest of the book. It's aimed at readers who don't have any prior knowledge of DI, but experienced DI practitioners may also benefit from skimming the chapters to get a feeling for the terminology used throughout the book. By the end of part 1, you should have a firm grasp of the vocabulary and overall concepts, even if some of the concrete details are still a little fuzzy. That's okay—the book becomes more concrete as you read on, so parts 2, 3, and 4 should answer the questions you're likely to have after reading part 1.

1 A Dependency Injection tasting menu

Menu

- Misconceptions about Dependency Injection
- Purpose of Dependency Injection
- Benefits of Dependency Injection
- When to apply Dependency Injection

You may have heard that making a *sauce béarnaise* is difficult. Even many people who cook regularly have never attempted to make one. This is a shame, because the sauce is delicious (it's traditionally paired with steak, but it's also an excellent accompaniment with white asparagus, poached eggs, and other dishes). Some resort to substitutes like ready-made sauces or instant mixes, but these aren't nearly as satisfying as the real thing.

> **DEFINITION** A sauce béarnaise is an emulsified sauce made from egg yolk and butter that's flavored with tarragon, chervil, shallots, and vinegar. It contains no water.

The biggest challenge to making a sauce béarnaise is that preparation can fail—the sauce may curdle or separate, and if that happens, you can't resurrect it. It takes about 45 minutes to prepare, so a failed attempt means that you'll have no time for a second try.

On the other hand, any chef can prepare a sauce béarnaise. It's part of their training and, as they will tell you, it's not difficult. You don't have to be a professional cook to make it. Anyone learning to make it will fail at least once, but once you get the hang of it, you'll succeed every time.

I think *Dependency Injection (DI)* is like sauce béarnaise. It's assumed to be difficult and so few employ it. If you try to use it and fail, it's likely there won't be time for a second attempt.

DEFINITION *Dependency Injection* is a set of software design principles and patterns that enable us to develop loosely coupled code.

Despite the Fear, Uncertainty, and Doubt (FUD) surrounding DI, it's as easy to learn as making a sauce béarnaise. You may make mistakes while you learn, but once you've mastered the technique, you'll never again fail to apply it successfully.

The software development Q&A website Stack Overflow features an answer to the question *How to explain Dependency Injection to a 5-year old*. The most highly rated answer, provided by John Munsch,[1] provides a surprisingly accurate analogy targeted at the (imaginary) five-year-old inquisitor:

> *When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.*
>
> *What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.*

What this means in terms of object-oriented software development is this: collaborating classes (the five-year-olds) should rely on the infrastructure (the parents) to provide the necessary services.

As figure 1.1 shows, this chapter is fairly linear in structure. First, I introduce DI, including its purpose and benefits. Although I include examples, overall, this chapter has less code than any other chapter in the book.

Before I introduce DI, I'll discuss the basic purpose of DI: maintainability. This is important because it's easy to misunderstand DI if you aren't properly prepared. Next, after an example (Hello DI), I'll discuss benefits and scope,

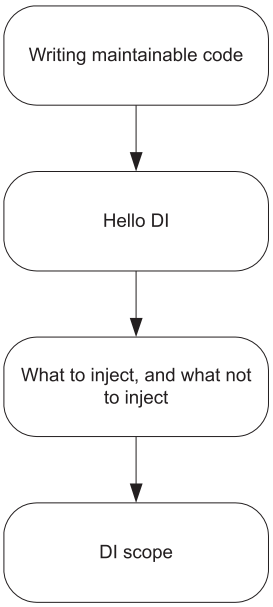


Figure 1.1 The structure of the chapter is fairly linear. You should read the first section before the next, and so on. This may seem obvious, but some of the later chapters in the book are less linear in nature.

[1] John Munsch et al., "How to explain Dependency Injection to a 5-year old," 2009, http://stackoverflow.com/questions/ 1638919/how-to-explain-dependency-injection-to-a-5-year-old

essentially laying out a road map for the book. When you're done with this chapter, you should be prepared for the more advanced concepts in the rest of the book.

To most developers, DI may seem like a rather backward way of creating source code, and, like sauce béarnaise, there's much FUD involved. To learn about DI, you must first understand its purpose.

1.1 Writing maintainable code

What purpose does DI serve? DI isn't a goal in itself; rather, it's a means to an end. Ultimately, the purpose of most programming techniques is to deliver working software as efficiently as possible. One aspect of that is to write maintainable code.

Unless you write prototypes or applications that never make it past release 1, you'll soon find yourself maintaining and extending existing code bases. To be able to work effectively with such a code base, it must be as maintainable as possible.

One of many ways to make code maintainable is through *loose coupling*. As far back as 1995, when the Gang of Four wrote *Design Patterns,*[2] this was already common knowledge:

> *Program to an interface, not an implementation.*

This important piece of advice isn't the conclusion, but, rather, the premise, of *Design Patterns*; to wit: it appears on page 18. Loose coupling makes code extensible, and extensibility makes it maintainable.

DI is nothing more than a technique that enables loose coupling. However, there are many misconceptions about DI, and sometimes they get in the way of proper understanding. Before you can learn, you must unlearn what (you think) you already know.

1.1.1 Unlearning DI

Like a Hollywood martial arts cliché, you must unlearn before you can learn. There are many misconceptions about DI, and if you carry those around, you'll misinterpret what you read in this book. You must clear your mind to understand DI.

There are at least four common myths about DI:

- DI is only relevant for late binding.
- DI is only relevant for unit testing.
- DI is a sort of Abstract Factory on steroids.
- DI requires a DI CONTAINER.

Although none of these myths are true, they're prevalent nonetheless. We need to dispel them before we can start to learn about DI.

---

[2] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software (New York,* Addison-Wesley, 1994), 18.

Figure 1.2 Late binding is enabled by DI, but to assume it's only applicable in late binding scenarios is to adopt a narrow view of a much broader vista.

LATE BINDING

In this context, *late binding* refers to the ability to replace parts of an application without recompiling the code. An application that enables third-party add-ins (such as Visual Studio) is one example.

Another example is standard software that supports different runtime environments. You may have an application that can run on more than one database engine: for example, one that supports both Oracle and SQL Server. To support this feature, the rest of the application can talk to the database through an interface. The code base can provide different implementations of this interface to provide access to Oracle and SQL Server, respectively. A configuration option can be used to control which implementation should be used for a given installation.

It's a common misconception that DI is only relevant for this sort of scenario. That's understandable, because DI *does* enable this scenario, but the fallacy is to think that the relationship is symmetric. Just because DI enables late binding doesn't mean it's only relevant in late binding scenarios. As figure 1.2 illustrates, late binding is only one of the many aspects of DI.

If you thought DI was only relevant for late binding scenarios, this is something you need to unlearn. DI does much more than enable late binding.

UNIT TESTING

Some people think that DI is only relevant to support unit testing. This isn't true, either—although DI is certainly an important part of supporting unit testing.

To tell you the truth, my original introduction to DI came from struggling with certain aspects of Test-Driven Development (TDD). During that time I discovered DI and learned that other people had used it to support some of the same scenarios I was addressing.

Even if you don't write unit tests (if you don't, you should start now), DI is still relevant because of all the other benefits it offers. Claiming that DI is only relevant to support unit testing is like claiming that it's only relevant for supporting late binding. Figure 1.3 shows that although this is a different view, it's a view as narrow as figure 1.2. In this book, I'll do my best to show you the whole picture.

If you thought DI was only relevant for unit testing, unlearn this assumption. DI does much more than enable unit testing.

Figure 1.3 Although the assumption that unit testing is the sole purpose of DI is a different view than late binding, it's also a narrow view of a much broader vista.

An Abstract Factory on steroids

Perhaps the most dangerous fallacy is that DI involves some sort of general-purpose Abstract Factory[3] that we can use to create instances of the Dependencies that we need.

In the introduction to this chapter, I wrote that "collaborating classes…should rely on the infrastructure…to provide the necessary services."

What were your initial thoughts about this sentence? Did you think about the *infrastructure* as some sort of service you could query to get the Dependencies you need? If so, you aren't alone. Many developers and architects think about DI as a service that can be used to locate other services; this is called a Service Locator, but it's the exact opposite of DI.

If you thought of DI as a Service Locator—that is, a general-purpose Factory—this is something you need to unlearn. DI is the opposite of a Service Locator; it's a way to structure code so that we never have to imperatively ask for Dependencies. Rather, we *force* consumers to supply them.

DI Containers

Closely associated with the previous misconception is the notion that DI *requires* a DI Container. If you held the previous, mistaken belief that DI involves a Service Locator, then it's easy to conclude that a DI Container can take on the responsibility of the Service Locator. This might be the case, but it's not at all how we should use a DI Container.

A DI Container is an optional library that can make it easier for us to compose components when we wire up an application, but it's in no way required. When we compose applications without a DI Container we call it Poor Man's DI; it takes a little more work, but other than that we don't have to compromise on any DI principles.

If you thought that DI *requires* a DI Container, this is another notion you need to unlearn. DI is a set of principles and patterns, and a DI Container is a useful, but *optional* tool.

You may think that, although I've exposed four myths about DI, I have yet to make a compelling case against any of them. That's true. In a sense, this whole book is one big argument against these common misconceptions.

In my experience, unlearning is vital because people tend to try to retrofit what I tell them about DI and align it with what they think they already know. When this happens, it takes a lot of time before it finally dawns on them that some of their most basic

[3] Ibid., 87.

premises are wrong. I want to spare you that experience. So, if you can, try to read this book as though you know nothing about DI.

Let's assume that you don't know anything about DI or its purpose and begin by reviewing what DI does.

1.1.2 Understanding the purpose of DI

DI isn't an end-goal—it's a means to an end. DI enables loose coupling, and loose coupling makes code more maintainable. That's quite a claim, and although I could refer you to well-established authorities like the Gang of Four for details, I find it only fair to explain why this is true.

Software development is still a rather new profession, so in many ways we're still figuring out how to implement good architecture. However, individuals with expertise in more traditional professions (such as construction) figured it out a long time ago.

CHECKING INTO A CHEAP HOTEL

If you're staying at a cheap hotel, you might encounter a sight like the one in figure 1.4. Here, the hotel has kindly provided a hair dryer for your convenience, but apparently they don't trust you to leave the hair dryer for the next guest: the appliance is directly attached into the wall outlet. Although the cord's long enough to give you a certain degree of movement, you can't take the dryer with you. Apparently, the hotel management has decided that the cost of replacing stolen hair dryers is high enough to justify what's otherwise an obviously inferior implementation.

What happens when the hair dryer stops working? The hotel has to call in a skilled professional who can deal with the issue. To fix the hardwired hair dryer, they will have to cut the power to the room, rendering it temporarily useless. Then, the technician will use special tools to painstakingly disconnect the hair dryer and replace it with a new one. If you're lucky, the technician will remember to turn the power to the room back on and go back to test whether the new hair dryer works…if you're lucky.

Figure 1.4 In a cheap hotel room, you might find the hair dryer wired directly into the wall outlet. This is equivalent to using the common practice of writing *tightly coupled* code.

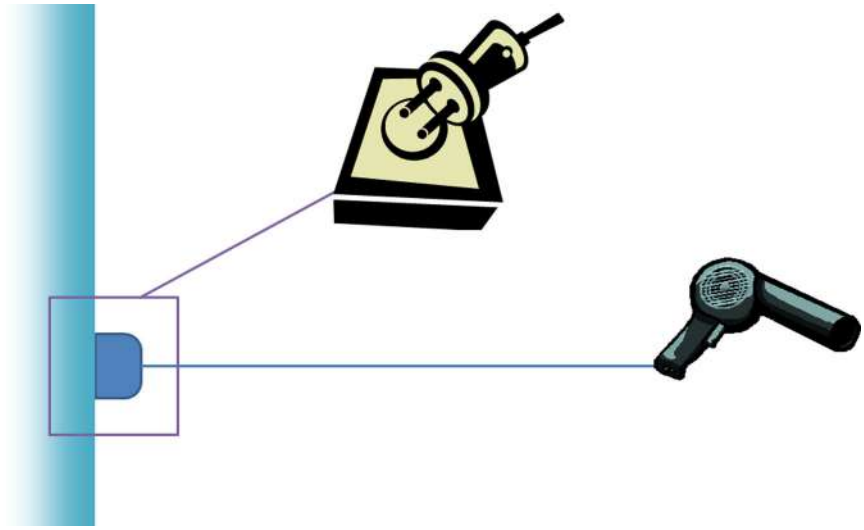


Figure 1.5 Through the use of sockets and plugs, a hair dryer can be *loosely coupled* to the wall outlet.

Does this procedure sound at all familiar?

This is how you would approach working with *tightly coupled* code. In this scenario, the hair dryer is tightly coupled to the wall and you can't easily modify one without impacting the other.

COMPARING ELECTRICAL WIRING TO DESIGN PATTERNS

Usually, we don't wire electrical appliances together by attaching the cable directly to the wall. Instead, as in figure 1.5, we use plugs and sockets. A socket defines a shape that the plug must match. In an analogy to software design, the socket is an *interface*.

In contrast to the hardwired hair dryer, plugs and sockets define a *loosely coupled* model for connecting electrical appliances. As long as the plug fits into the socket, we can combine appliances in a variety of ways. What's particularly interesting is that many of these common combinations can be compared to well-known software design principles and patterns.

First, we're no longer constrained to hair dryers. If you're an average reader, I would guess that you need power for a computer much more than you do for a hair dryer. That's not a problem: we unplug the hair dryer and plug a computer into the same socket, as shown in figure 1.6.

It's amazing that the concept of a socket predates computers by decades, and yet it provides an essential service to computers, too. The original designers of sockets couldn't possibly have foreseen personal computers, but because the design is so versatile, needs that were originally unanticipated can be met. The ability to replace one end without changing the other is similar to a central software design principle called the LISKOV SUBSTITUTION PRINCIPLE. This principle states that we should be able to

Figure 1.6 Using sockets and plugs, we can replace the original hair dryer from figure 1.5 with a computer. This corresponds to the LISKOV SUBSTITUTION PRINCIPLE.

replace one implementation of an interface with another without breaking either client or implementation.

When it comes to DI, the LISKOV SUBSTITUTION PRINCIPLE is one of the most important software design principles. It's this principle that enables us to address requirements that occur in the future, even if we can't foresee them today.

As figure 1.7 illustrates, we can unplug the computer if we don't need to use it at the moment. Even though nothing is plugged in, the wall doesn't explode.

If we unplug the computer from the wall, neither the wall outlet nor the computer breaks down (in fact, if it's a laptop computer, it can even run on its batteries for a period of time). With software, however, a client often expects a service to be available. If the service was removed, we get a `NullReferenceException`. To deal with this type of situation, we can create an implementation of an interface that does "nothing." This is a design pattern known as *Null Object*,[4] and it corresponds roughly to unplugging the

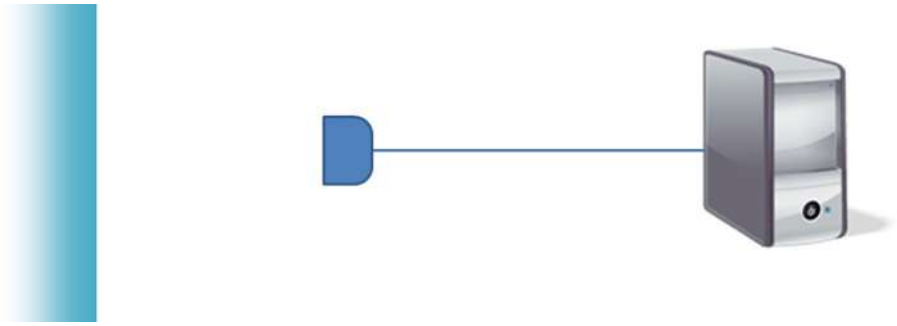


Figure 1.7 Unplugging the computer causes neither wall nor computer to explode. This can be roughly likened to the Null Object pattern.

[4] Robert C. Martin et al., *Pattern Languages of Program Design* 3 (New York, Addison-Wesley, 1998), 5.

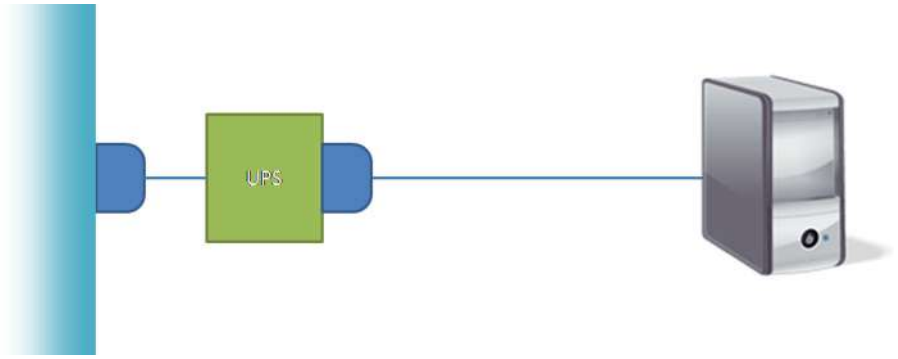


Figure 1.8 An Uninterrupted Power Supply can be introduced to keep the computer running in case of power failures. This corresponds to the Decorator design pattern.

computer from the wall. Because we're using loose coupling, we can replace a real implementation with something that does nothing without causing trouble.

There are many other things we can do. If we live in a neighborhood with intermittent power failures, we may wish to keep the computer running by plugging in into an Uninterrupted Power Supply (UPS), as shown in figure 1.8: we connect the UPS to the wall outlet and the computer to the UPS.

The computer and the UPS serve separate purposes. Each has a SINGLE RESPONSIBILITY that doesn't infringe on the other appliance. The UPS and computer are likely to be produced by two different manufacturers, bought at different times, and plugged in at different times. As figure 1.6 demonstrates, we can run the computer without a UPS, but we could also conceivably use the hair dryer during blackouts by plugging it into the UPS.

In software design, this way of INTERCEPTING one implementation with another implementation of the same interface is known as the *Decorator*[5] design pattern. It gives us the ability to incrementally introduce new features and CROSS-CUTTING CONCERNS without having to rewrite or change a lot of our existing code.

Another way to add new functionality to an existing code base is to compose an existing implementation of an interface with a new implementation. When we aggregate several implementations into one, we use the *Composite*[6] design pattern. Figure 1.9 illustrates how this corresponds to plugging diverse appliances into a power strip.

The power strip has a single plug that we can insert into a single socket, while the power strip itself provides several sockets for a variety of appliances. This enables us to add and remove the hair dryer while the computer is running. In the same way, the Composite pattern makes it easy to add or remove functionality by modifying the set of composed interface implementations.

---

[5] Gamma, *Design Patterns*, 175.
[6] Ibid., 163.

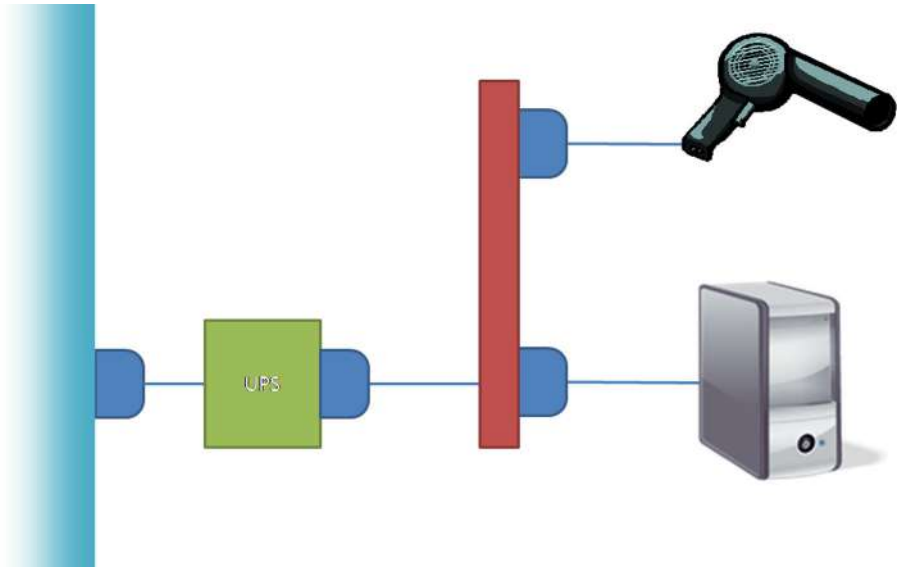


Figure 1.9 A power strip makes it possible to plug several appliances into a single wall outlet. This corresponds to the Composite design pattern.

Here's a final example. We sometimes find ourselves in situations where a plug doesn't fit into a particular socket. If you've traveled to another country, you've likely noticed that sockets differ across the world. If you bring something, like the camera in figure 1.10, along when traveling, you need an adapter to charge it. Appropriately, there's a design pattern with the same name.

The *Adapter*[7] design pattern works like its physical namesake. It can be used to match two related, yet separate, interfaces to each other. This is particularly useful

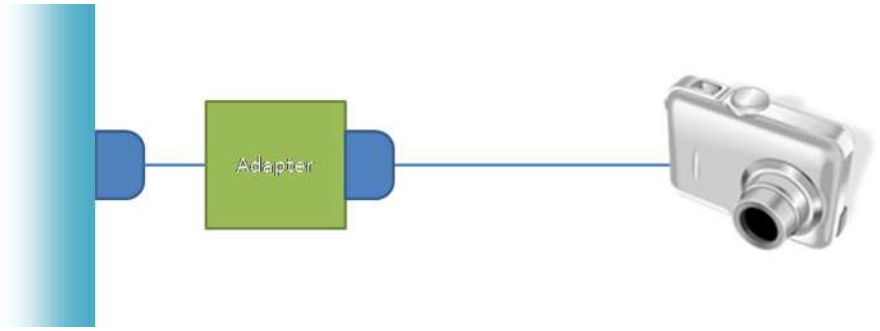


Figure 1.10 When traveling, we often need to use an adapter to plug an appliance into a foreign socket (for example, to recharge a camera). This corresponds to the Adapter design pattern.

[7] Ibid., 139.

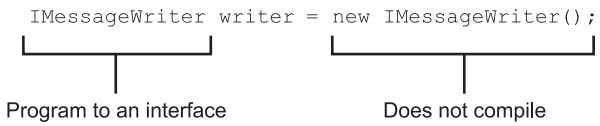
when you have an existing third-party API that you wish to expose as an instance of an interface your application consumes.

What's amazing about the socket and plug model is that, over decades, it's proven to be an easy and versatile model. Once the infrastructure is in place, it can be used by anyone and adapted to changing needs and unpredicted requirements. What's even more interesting is that, when we relate this model to software development, all the building blocks are already in place in the form of design principles and patterns.

Loose coupling can make a code base much more maintainable.

That's the easy part. Programming to an interface instead of an implementation is easy. The question is, where do the instances come from? In a sense, this is what this entire book is about.

You can't create a new instance of an interface the same way that you create a new instance of a concrete type. Code like this doesn't compile:



An interface has no constructor, so this isn't possible. The `writer` instance must be created using a different mechanism. DI solves this problem.

With this outline of the purpose of DI, I think you're ready for an example.

1.2 Hello DI

In the tradition of innumerable programming textbooks, let's take a look at a simple console application that writes "Hello DI!" to the screen. In this section, I'll show you what the code looks like and briefly outline some key benefits without going into details—in the rest of the book, I'll get more specific.

1.2.1 Hello DI code

You're probably used to seeing Hello World examples that are written in a single line of code. Here, we'll take something that's extremely simple and make it more complicated. Why? We'll get to that shortly, but let's first see what Hello World would look like with DI.

COLLABORATORS

To get a sense of the structure of the program, we'll start by looking at the `Main` method of the console application, and then I'll show you the collaborating classes:

```
private static void Main()
{
    IMessageWriter writer = new ConsoleMessageWriter();
    var salutation = new Salutation(writer);
    salutation.Exclaim();
}
```

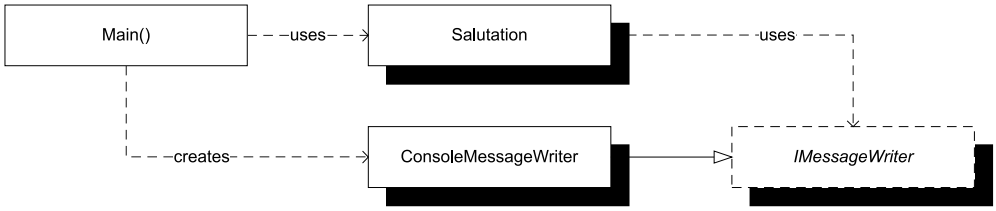


Figure 1.11 The `Main` method creates new instances of both the `ConsoleMessageWriter` and `Salutation` classes. `ConsoleMessageWriter` implements the `IMessageWriter` interface, which `Salutation` uses. In effect, Salutation uses `ConsoleMessageWriter`, although this indirect usage isn't shown.

The program needs to write to the console, so it creates a new instance of `ConsoleMessageWriter` that encapsulates exactly that functionality. It passes that message writer to the `Salutation` class so that the `salutation` instance knows where to write its messages. Because everything is now wired up properly, you can execute the logic, which results in the message being written to the screen.

Figure 1.11 shows the relationship between the collaborators.

The main logic of the application is encapsulated in the `Salutation` class, shown in the following listing.

Listing 1.1 `Salutation` class

```
public class Salutation
{
    private readonly IMessageWriter writer;

    public Salutation(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }

        this.writer = writer;
    }

    public void Exclaim()
    {
        this.writer.Write("Hello DI!");
    }
}
```

❶ **Inject Dependency**

❷ **Use Dependency**

The `Salutation` class depends on a custom interface called `IMessageWriter`, and it requests an instance of it through its constructor ❶. This is called CONSTRUCTOR INJECTION and is described in detail in chapter 4, which also contains a more detailed walk-through of a similar code example.

The `IMessageWriter` instance is later used in the implementation of the `Exclaim` method ❷, which writes the proper message to the DEPENDENCY.

`IMessageWriter` is a simple interface defined for the occasion:

```
public interface IMessageWriter
{
    void Write(string message);
}
```

It could have had other members, but in this simple example you only need the `Write` method. It's implemented by the `ConsoleMessageWriter` class that the `Main` method passes to the `Salutation` class:

```
public class ConsoleMessageWriter : IMessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

The `ConsoleMessageWriter` class implements `IMessageWriter` by wrapping the Base Class Library's `Console` class. This is a simple application of the Adapter design pattern that we talked about in section 1.1.2.

You may be wondering about the benefit of replacing a single line of code with two classes and an interface with a total line count of 11, and rightly so. There are several benefits to be harvested from doing this.

1.2.2 Benefits of DI

How is the previous example better than the usual single line of code we normally use to implement Hello World in C#? In this example, DI adds an overhead of 1,100%, but, as complexity increases from one line of code to tens of thousands, this overhead diminishes and all but disappears. Chapter 2 provides a more complex example of applied DI, and although that example is still overly simplistic compared to real-life applications, you should notice that DI is far less intrusive.

I don't blame you if you find the previous DI example to be over-engineered, but consider this: by its nature, the classic Hello World example is a *simple* problem with well-specified and constrained requirements. In the real world, software development is never like this. Requirements change and are often fuzzy. The features you must implement also tend to be much more complex. DI helps address such issues by enabling loose coupling. Specifically, we gain the benefits listed in table 1.1.

In table 1.1, I listed the *late binding* benefit first because, in my experience, this is the one that's foremost in most people's minds. When architects and developers fail to understand the benefits of loose coupling, this is most likely because they never consider the other benefits.

LATE BINDING

When I explain the benefits of *programming to interfaces* and DI, the ability to swap out one service with another is the most prevalent benefit for most people, so they tend to weigh the advantages against the disadvantages with only this benefit in mind.

Table 1.1 Benefits gained from loose coupling. Each benefit is always available but will be valued differently depending on circumstances.

Benefit	Description	When is it valuable?
Late binding	Services can be swapped with other services.	Valuable in standard software, but perhaps less so in enterprise applications where the runtime environment tends to be well-defined
Extensibility	Code can be extended and reused in ways not explicitly planned for.	Always valuable
Parallel development	Code can be developed in parallel.	Valuable in large, complex applications; not so much in small, simple applications
Maintainability	Classes with clearly defined responsibilities are easier to maintain.	Always valuable
TESTABILITY	Classes can be unit tested.	Only valuable if you unit test (which you really, really should)

Remember when I asked you to unlearn before you can learn? You may say that you know your requirements so well that you *know* you'll never have to replace, say, your SQL Server database with anything else. However, requirements change.

NoSQL, Windows Azure, and the argument for composability

Years ago, I was often met with a blank expression when I tried to convince developers and architects of the benefits of DI.

"Okay, so you can swap out your relational data access component for something else. For what? Is there any alternative to relational databases?"

XML files never seemed like a convincing alternative in highly scalable enterprise scenarios. This has changed significantly in the last couple of years.

Windows Azure was announced at PDC 2008 and has done much to convince even die-hard Microsoft-only organizations to reevaluate their position when it comes to data storage. There's now a real alternative to relational databases, and I only have to ask if people want their application to be "cloud-ready." The replacement argument has now become much stronger.

A related movement can be found in the whole NoSQL concept that models applications around denormalized data—often document databases, but concepts such as Event Sourcing[8] are also becoming increasingly important.

In section 1.2.1, you didn't use late binding because you explicitly created a new instance of `IMessageWriter` by hard-coding creation of a new `ConsoleMessageWriter`

[8] Martin Fowler, "Event Sourcing," 2005, www.martinfowler.com/eaaDev/EventSourcing.htmls

instance. However, you can introduce late binding by changing only a single piece of the code. You only need to change this line of code:

```
IMessageWriter writer = new ConsoleMessageWriter();
```

To enable late binding, you might replace that line of code with something like this:

```
var typeName =
    ConfigurationManager.AppSettings["messageWriter"];
var type = Type.GetType(typeName, true);
IMessageWriter writer =
    (IMessageWriter)Activator.CreateInstance(type);
```

By pulling the type name from the application configuration file and creating a `Type` instance from it, you can use Reflection to create an instance of `IMessageWriter` without knowing the concrete type at compile time.

To make this work, you specify the type name in the `messageWriter` application setting in the application configuration file:

```
<appSettings>
  <add key="messageWriter"
       value="Ploeh.Samples.HelloDI.CommandLine.ConsoleMessageWriter,
       ➥HelloDI" />
</appSettings>
```

> **WARNING** This example takes some shortcuts to make a point. In fact, it suffers from the CONSTRAINED CONSTRUCTION anti-pattern, covered in detail in chapter 5.

Loose coupling enables late binding because there's only a single place where you create the instance of the `IMessageWriter`. Because the `Salutation` class works exclusively against the `IMessageWriter` interface, it never notices the difference.

In the Hello DI example, late binding would enable you to write the message to a different destination than the console—for example, a database or a file. It's possible to add such features even though you didn't explicitly plan ahead for them.

EXTENSIBILITY

Successful software must be able to change. You'll need to add new features and extend existing features. Loose coupling enables us to efficiently recompose the application, similar to the way that we can rewire electrical appliances using plugs and sockets.

Let's say that you want to make the Hello DI example more secure by only allowing authenticated users to write the message. The next listing shows how you can add that feature without changing any of the existing features: you add a new implementation of the `IMessageWriter` interface.

Listing 1.2 Extending the Hello DI application with a security feature

```
public class SecureMessageWriter : IMessageWriter
{
    private readonly IMessageWriter writer;
```

```
    public SecureMessageWriter(IMessageWriter writer)
    {
        if (writer == null)
        {
            throw new ArgumentNullException("writer");
        }

        this.writer = writer;
    }

    public void Write(string message)
    {
        if (Thread.CurrentPrincipal.Identity
            .IsAuthenticated)                          ❶ Check authentication
        {
            this.writer.Write(message);                ❷ Write message
        }
    }
}
```

The `SecureMessageWriter` class implements the `IMessageWriter` interface while also consuming it: it uses CONSTRUCTOR INJECTION to request an instance of `IMessageWriter`. This is a standard application of the Decorator design pattern that I mentioned in section 1.1.2. We'll talk much more about Decorators in chapter 9.

The `Write` method is implemented by first checking whether the current user is authenticated ❶. Only if this is the case does it allow the decorated `writer` field to `Write` ❷ the message.

> **NOTE** The `Write` method in listing 1.2 accesses the current user via an AMBIENT CONTEXT. A more flexible, but slightly more complex, option would've been to also supply the user via CONSTRUCTOR INJECTION.

The only place where you need to change existing code is in the `Main` method, because you need to compose the available classes differently than before:

```
IMessageWriter writer =
    new SecureMessageWriter(
        new ConsoleMessageWriter());
```

Notice that you decorate the old `ConsoleMessageWriter` instance with the new `SecureMessageWriter` class. Once more, the `Salutation` class is unmodified because it only consumes the `IMessageWriter` interface.

Loose coupling enables you to write code which is *open for extensibility, but closed for modification*. This is called the OPEN/CLOSED PRINCIPLE. The only place where you need to modify the code is at the application's entry point; we call this the COMPOSITION ROOT.

The `SecureMessageWriter` implements the security features of the application, whereas the `ConsoleMessageWriter` addresses the user interface. This enables us to vary these aspects independently of each other and compose them as needed.