



Object-Oriented
Principles, SOLID,
Abstraction,
Encapsulation,
SOLID, Design
Patterns, Factory
Builder, Singleton,
Clean Code, Scalable
Clean Architecture

```
class Animal {  
    public void run() {  
        System.out.println("Animal is running");  
    }  
}  
class Dog extends Animal {  
    public void bark() {  
        System.out.println("Dog barks");  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        Dog h = new Dog();  
        h.run();  
        h.bark();  
    }  
}
```



Software Architecture and Design

The Practical Guide to Design Patterns

Kristian Köhler



Rheinwerk
Computing

Contents

1	Introduction	15
1.1	Programming Paradigms	17
1.1.1	Structured Programming	18
1.1.2	Object-Oriented Programming	20
1.1.3	Functional Programming	22
1.1.4	Reactive Programming	25
1.2	What Are Design Patterns and How Did They Come About?	27
1.3	What Are Software Architecture and Software Design?	31
1.3.1	Tasks of a Software Architecture	33
1.3.2	Architectural Styles and Architectural Patterns	35
1.4	The Evolution of Software Development and Architecture	38
1.4.1	Class-Responsibility-Collaboration Cards	38
1.4.2	The “Gang of Four” Patterns and Their Structure	39
1.4.3	Clean Code	43
1.4.4	Component-Based Development	45
1.4.5	Core J2EE Patterns	50
1.4.6	Enterprise Integration Patterns	54
1.4.7	Well-Architected Cloud	60
2	Principles of Good Software Design	63
2.1	Basic Concepts of Object-Oriented Programming	64
2.1.1	Objects and Classes	65
2.1.2	Encapsulation	66
2.1.3	Abstraction	68
2.1.4	Inheritance	72
2.1.5	Polymorphism	73
2.2	Clean Code Principles	75
2.2.1	Identifier Names and Conventions in the Code	77
2.2.2	Defining Functions and Methods	89
2.2.3	Don’t Repeat Yourself	104
2.2.4	Establishing Clear Boundaries Between Components	105
2.2.5	The Broken Windows Theory and the Boy Scout Rule in Software	107

2.3	SOLID Principles	108
2.3.1	The Single Responsibility Principle	109
2.3.2	The Open-Closed Principle	113
2.3.3	The Liskov Substitution Principle	119
2.3.4	The Interface Segregation Principle	123
2.3.5	The Dependency Inversion Principle and the Inversion of Control	128
2.4	Information Hiding	131
2.5	Inversion of Control and Dependency Injection	132
2.6	Separation of Concerns and Aspect Orientation	134
2.7	Quality Assurance with Unit Tests	137

3 Source Code and Documenting the Software Development 143

3.1	Comments in the Source Code	143
3.1.1	Documenting Interfaces Using Comments	145
3.1.2	Creating Expressive Code	149
3.1.3	Necessary and Meaningful Comments	150
3.1.4	Comments Not Needed	154
3.2	Documenting the Software Architecture	157
3.2.1	Documenting Quality Features	158
3.2.2	Rules for Good Software Architecture Documentation	159
3.2.3	arc42 for the Complete Documentation	162
3.3	Representing Software in Unified Modeling Language	169
3.3.1	Use Case Diagram	170
3.3.2	Class Diagram	171
3.3.3	Sequence Diagram	174
3.3.4	State Diagram	175
3.3.5	Component Diagram	177
3.4	C4 Model for Representing Software Architecture	180
3.4.1	System Context (Level 1 or C1)	184
3.4.2	Container (Level 2 or C2)	186
3.4.3	Component (Level 3 or C3)	187
3.4.4	Code (Level 4 or C4)	188
3.5	Doc-as-Code	188
3.5.1	AsciiDoc	189
3.5.2	PlantUML	191
3.5.3	Structurizr	194

4	Software Patterns	197
4.1	Factory Method	198
4.1.1	Problem and Motivation	198
4.1.2	Solution	200
4.1.3	Sample Solution	201
4.1.4	When To Use the Pattern	203
4.1.5	Consequences	204
4.1.6	Real-World Example in Open-Source Software	205
4.2	Builder	206
4.2.1	Problem and Motivation	206
4.2.2	Solution	208
4.2.3	Sample Solution	209
4.2.4	When To Use the Pattern	213
4.2.5	Consequence	214
4.2.6	Real-World Example in Open-Source Software	214
4.3	Strategy	216
4.3.1	Problem and Motivation	216
4.3.2	Solution	216
4.3.3	Sample Solution	217
4.3.4	When To Use the Pattern	220
4.3.5	Consequences	221
4.3.6	Real-World Example	222
4.4	Chain of Responsibility	223
4.4.1	Problem and Motivation	224
4.4.2	Solution	224
4.4.3	Sample Solution	226
4.4.4	When To Use the Pattern	228
4.4.5	Consequences	228
4.4.6	Real-World Example	229
4.5	Command	232
4.5.1	Problem and Motivation	232
4.5.2	Solution	233
4.5.3	Sample Solution	235
4.5.4	When To Use the Pattern	238
4.5.5	Consequences	239
4.5.6	Real-World Example	239
4.6	Observer	243
4.6.1	Problem and Motivation	243
4.6.2	Solution	244

4.6.3	Sample Solution	245
4.6.4	When To Use the Pattern	248
4.6.5	Consequences	249
4.6.6	Real-World Example	249
4.7	Singleton	251
4.7.1	Problem and Motivation	251
4.7.2	Solution	252
4.7.3	Sample Solution	253
4.7.4	When To Use the Pattern	255
4.7.5	Consequences	256
4.7.6	Real-World Example	258
4.8	Adapter/Wrapper	259
4.8.1	Problem and Motivation	259
4.8.2	Solution	260
4.8.3	Sample Solution	261
4.8.4	When To Use the Pattern	264
4.8.5	Consequences	265
4.8.6	Real-World Example	265
4.9	Iterator	268
4.9.1	Problem and Motivation	269
4.9.2	Solution	269
4.9.3	Sample Solution	271
4.9.4	When To Use the Pattern	273
4.9.5	Consequences	274
4.9.6	Real-World Example	274
4.10	Composite	276
4.10.1	Problem and Motivation	276
4.10.2	Solution	277
4.10.3	Sample Solution	278
4.10.4	When To Use the Pattern	281
4.10.5	Consequences	281
4.10.6	Real-World Example	281
4.11	The Concept of Anti-Patterns	283
4.11.1	Big Ball of Mud	283
4.11.2	God Object	284
4.11.3	Spaghetti Code	285
4.11.4	Reinventing the Wheel	286
4.11.5	Cargo Cult Programming	287

5 Software Architecture, Styles, and Patterns 289

5.1	The Role of the Software Architect	290
5.2	Software Architecture Styles	292
5.2.1	Client-Server Architecture	293
5.2.2	Layered Architecture and Service Layers	294
5.2.3	Event-Driven Architecture	299
5.2.4	Microkernel Architecture or Plugin Architecture	304
5.2.5	Microservices	307
5.3	Styles for Application Organization and Code Structure	310
5.3.1	Domain-Driven Design	311
5.3.2	Strategic and Tactical Designs	315
5.3.3	Hexagonal Architecture/Ports and Adapters	315
5.3.4	Clean Architecture	320
5.4	Patterns for the Support of Architectural Styles	324
5.4.1	Model View Controller Pattern	324
5.4.2	Model View ViewModel Pattern	329
5.4.3	Data Transfer Objects	335
5.4.4	Remote Facade Pattern	339

6 Communication Between Services 347

6.1	Styles of Application Communication	349
6.1.1	Synchronous Communication	350
6.1.2	Asynchronous Communication and Messaging	351
6.1.3	Streaming	353
6.2	Resilience Patterns	356
6.2.1	Error Propagation	357
6.2.2	Subdivision of the Resilience Patterns	358
6.2.3	Timeout Pattern	361
6.2.4	Retry Pattern	366
6.2.5	Circuit Breaker Pattern	372
6.2.6	Bulkhead Pattern	378
6.2.7	Steady State Pattern	383
6.3	Messaging Patterns	388
6.3.1	Messaging Concepts	388
6.3.2	Messaging Channel Patterns	389
6.3.3	Message Construction Patterns	395
6.3.4	Messaging Endpoint Pattern	402

6.4	Patterns for Interface Versioning	411
6.4.1	Endpoint for Version Pattern	415
6.4.2	Referencing Message Pattern	415
6.4.3	Self-Contained Message Pattern	417
6.4.4	Message with Referencing Metadata	418
6.4.5	Message with Self-Describing Metadata	420

7 Patterns and Concepts for Distributed Applications 421

7.1	Consistency	422
7.2	The CAP Theorem	423
7.3	The PACELC Theorem	424
7.4	Eventual Consistency	425
7.5	Stateless Architecture Pattern	428
7.5.1	Problem and Motivation	428
7.5.2	Solution	429
7.5.3	Sample Solution	431
7.5.4	When To Use the Pattern	433
7.5.5	Consequences	433
7.6	Database per Service Pattern	434
7.6.1	Problem and Motivation	434
7.6.2	Solution	434
7.6.3	Sample Solution	435
7.6.4	When To Use the Pattern	436
7.6.5	Consequences	436
7.7	Optimistic Locking Pattern	437
7.7.1	Problem and Motivation	437
7.7.2	Solution	438
7.7.3	Sample Solution	441
7.7.4	When To Use the Pattern	443
7.7.5	Consequences	443
7.7.6	Pessimistic Locking	444
7.8	Saga Pattern: The Distributed Transactions Pattern	446
7.8.1	Problem and Motivation	446
7.8.2	Solution	447
7.8.3	Sample Solution	447
7.8.4	When To Use the Pattern	449
7.8.5	Consequences	449

7.9	Transactional Outbox Pattern	450
7.9.1	Problem and Motivation	450
7.9.2	Solution	451
7.9.3	Sample Solution	452
7.9.4	When To Use the Pattern	454
7.9.5	Consequences	455
7.10	Event Sourcing Pattern	455
7.10.1	Problem and Motivation	455
7.10.2	Solution	456
7.10.3	Sample Solution	457
7.10.4	When To Use the Pattern	460
7.10.5	Consequences	461
7.11	Command Query Responsibility Segregation Pattern	461
7.11.1	Problem and Motivation	461
7.11.2	Solution	463
7.11.3	Sample Solution	464
7.11.4	When To Use the Pattern	467
7.11.5	Consequences	467
7.12	Distributed Tracing Pattern	467
7.12.1	Problem and Motivation	468
7.12.2	Solution	468
7.12.3	Sample Solution	470
7.12.4	When To Use the Pattern	476
7.12.5	Consequences	476
The Author		479
Index		481

Chapter 3

Source Code and Documenting the Software Development

Software must be documented. Good documentation not only consists of detailed information about the source code but also covers the architecture of a software system. Documentation enables developers to understand, adapt, and maintain software efficiently and thus contributes to the success, traceability, and quality management of a system. This chapter describes how you can create comprehensive, high-quality documentation for your software.

Documentation often makes a decisive contribution to the success of a project. However, each target group has different interests in software documentation. For example, users and customers need user documentation that describes how to use the finished software. Developers, project managers, even perhaps your quality management team, are more interested in technical details, information on the structure of the software, or even fundamental decisions made during development.

The goal of creating software documentation is to provide clear insights into the structure, functionality, and implementation of the software and thus create a better understanding among all team members. Detailed, meaningful information allows the code to be understood, expanded, and maintained more efficiently.

Many projects are not created, expanded, and maintained with a constant project team: Developers leave the project, while new developers join the project team. Documentation is therefore an indispensable part of the development process to ensure the long-term success of software and create a uniform understanding of the solution.

This chapter shows how software can be documented in the development process for various technically interested target groups.

3.1 Comments in the Source Code

Comments in the source code can be enormously helpful in making code that is difficult to understand more comprehensible for others or, after a certain time, for yourself. For example, they can provide further, additional details that enables the expansion or adaptation of the corresponding code. Poorly formulated or even outdated comments

in the code often have the exact opposite effect and lead to confusion or incorrect assumptions.

For this reason, one goal in software development should be to write expressive, readable code that speaks for itself and that a developer can understand even without separate, explicit comments within the source code. Robert C. Martin, the author of the *Clean Code* book, even describes certain comments as errors, because these comments reveal that the developer couldn't express themselves via the source code alone and couldn't produce comprehensible code. However, Martin also makes clear that, in some cases, comments are necessary, but you should never be proud of them.

In Martin's opinion, every comment should be checked to see whether the code itself can be written more clearly—and the comment thus becomes superfluous.

Bad code cannot be improved by a comment. You'll often find comments at particularly confusing code positions about what is done or intended in the respective section. If you have the feeling during development that a code point should be documented for clarity, you should consider rewriting this code block so that no additional comments are needed.

Writing fewer comments in the code also has a major advantage: It reduces the effort involved in maintaining the software. In any case, comments—like the source code itself—must be well maintained and carefully considered. If fewer comments are needed to understand the code, less maintenance is required. In projects that have undergone several revisions, comments are often outdated, incorrect, or in the wrong place due to code shifts. Unfortunately, comments cannot be fully relied upon.

A major challenge is to keep the comments synchronized with the current code. However, developers often fail to make this effort or overlook this task.

This problem can be mitigated with code that is self-documenting and self-explanatory. Comments then fade into the background as they are no longer needed, and revisions to the code do not directly lead to outdated comments. If code is moved to another location, for example, the corresponding comment does not have to be moved as well.

"Truth can only be found in one place: the code." —Robert C. Martin

Although you might be getting the impression that no comments are useful, you must not regard all comments or all documentation as mistaken, bad, or superfluous. In some situations, comments might be useful or even absolutely necessary due to various conditions (e.g., licensing terms). Public interfaces, such as the application programming interfaces (APIs) of libraries, must or should always be documented so that other developers can use them.

In the following sections, I will discuss how you can create comments in the source code and how you can minimize the scope of these comments or avoid them entirely by using more expressive code. I will show you examples where comments are useful or superfluous.

Formatting Comments in Your Source Code

In most programming languages, source code comments are introduced either by two slashes (//) or by the combination of a slash and an asterisk (/*). The latter is used to define comment blocks, which must then be closed using */. The two slashes, on the other hand, are used for single-line comments.

3.1.1 Documenting Interfaces Using Comments

In most languages, you can document a programming interface with the specific position and syntax of the comments. You can then use the appropriate tools to create a document that you can make available to other developers as documentation.

Documenting Java Source Code Using Javadoc

In Java, for example, you can use the `javadoc` command-line tool to generate a clear collection of HTML pages from Java source code, as shown in Figure 3.1. Such documentation is available for all versions of the standard Java library.

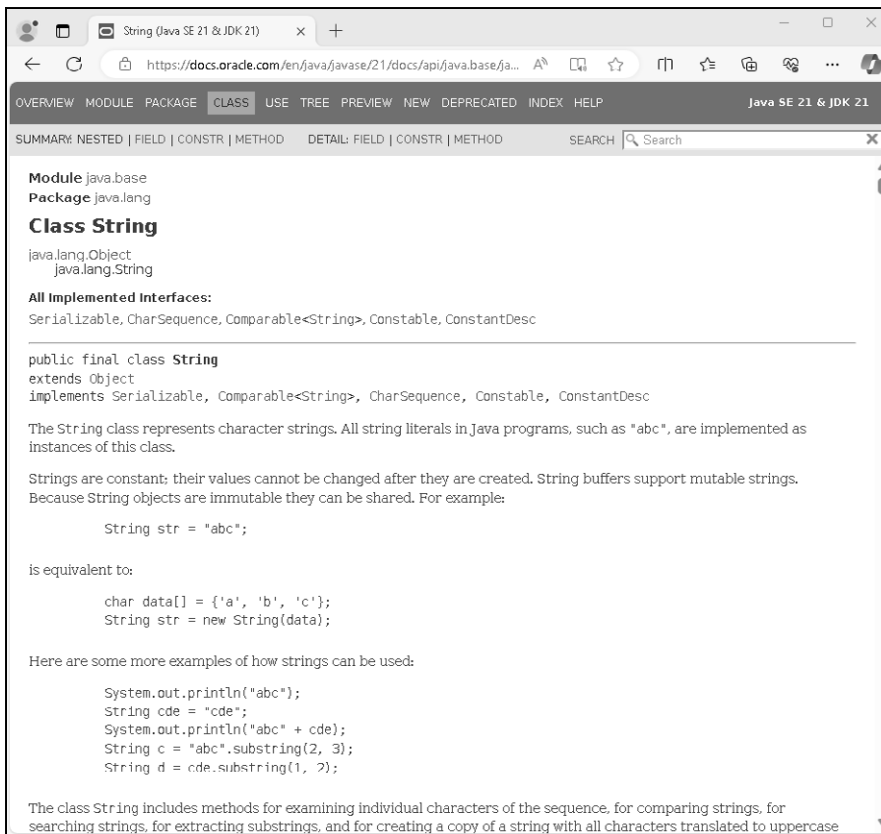


Figure 3.1 Javadoc Documentation from Source Code Comments

The documentation of the API is included via comments directly above the implementation. A few keywords or *annotations* control the appearance and meaning of the specified values. Using `@param`, for example, you document a parameter, while `@return` or `@throws` describe return values or indicate that exceptions may be thrown, as shown in Listing 3.1.

```
/**
 * Allocates a new {@code String} so that it represents the sequence of
 * characters currently contained in the character array argument. The
 * contents of the character array are copied; subsequent modification of
 * the character array does not affect the newly created string.
 *
 * @param value
 *         The initial value of the string
 */
public String(char[] value) {
    this(value, 0, value.length, null);
}
```

Listing 3.1 JavaDoc Example for the `java.lang.String` Class

Generating Documentation from Source Code

The concept that documentation can be generated from the source code can be found in many programming languages. This approach is based on the *DocBook* format in which technical documentation is stored in structured XML documents and can be converted into various output formats.

In Python, for example, you can use *Sphinx* to convert the *DocStringx* format into documentation.

Code Examples as Documentation in Go

In addition to the documentation options via text, Go also provides the option of creating *example tests*. They are implemented in the same way as unit tests and executed like a unit test during a test run using `go test`.

The code examples contained in the tests are intended to show how methods or functions can be used correctly.

Each test is assigned to a specific method or function via a test method name pattern and is automatically displayed as text under the specified method or function when the HTML representation of the documentation is created. The documentation therefore contains not only the written interface descriptions but also example code demonstrating their use.

Creating Go HTML Documentation Using Godoc

To create an HTML representation of your Go documentation, you can install and use the `godoc` tool. The installation is carried out via a `go install` command, which downloads the required files and installs them locally:

```
go install golang.org/x/tools/cmd/godoc@latest
```

If the Go environment is configured correctly and the `GOBIN` directory is contained in the operating system environment variable `PATH`, you can generate the documentation via the following command and it will be accessible at `http://localhost:6060` on your local machine:

```
godoc -http :6060
```

Because the code examples are implemented as tests and checked during each test run, the examples are always executable, and the documentation remains up to date. A user can rely on the examples given.

The checks and test conditions for the sample code are written using a special syntax. At the end, a check can be initiated using the `//Output: string`. The character string following this expression must have been output within the test via an `fmt.Print` output.

Our next example, shown in Listing 3.2, is an example test in which a `Car` object is created, and the `Color` attribute, which is pre-initialized to the value `Blue`, is changed to the value `Red` by a corresponding method call. This step is followed by an output and a check using the `//Output: string`.

```
func ExampleCar_PaintRed() {  
  
    a := &Car{Color: "Blue"}  
    a.PaintRed()  
    fmt.Println(a.Color)  
    //Output: Red  
}
```

Listing 3.2 Example Test with a Color Check (Go)

Many use cases exist in the standard Go library. Our next example, shown in Listing 3.3, illustrates a `Get` function from the `http` package, which is implemented without verification. It should “only” show the use of the function.

```
func ExampleGet() {  
    res, err := http.Get("http://www.google.com/robots.txt")  
    if err != nil {  
        log.Fatal(err)  
    }  
}
```

```

body, err := io.ReadAll(res.Body)
res.Body.Close()
if res.StatusCode > 299 {
    log.Fatalf("Response ...: %d and\nbody: %s\n", res.StatusCode, body)
}
if err != nil {
    log.Fatal(err)
}
fmt.Printf("%s", body)
}

```

Listing 3.3 Example Test from the "net/http" Package for the Get Function (Go)

Figure 3.2 shows the representation for this code shown, which is an example of the Get function from the net/http package.

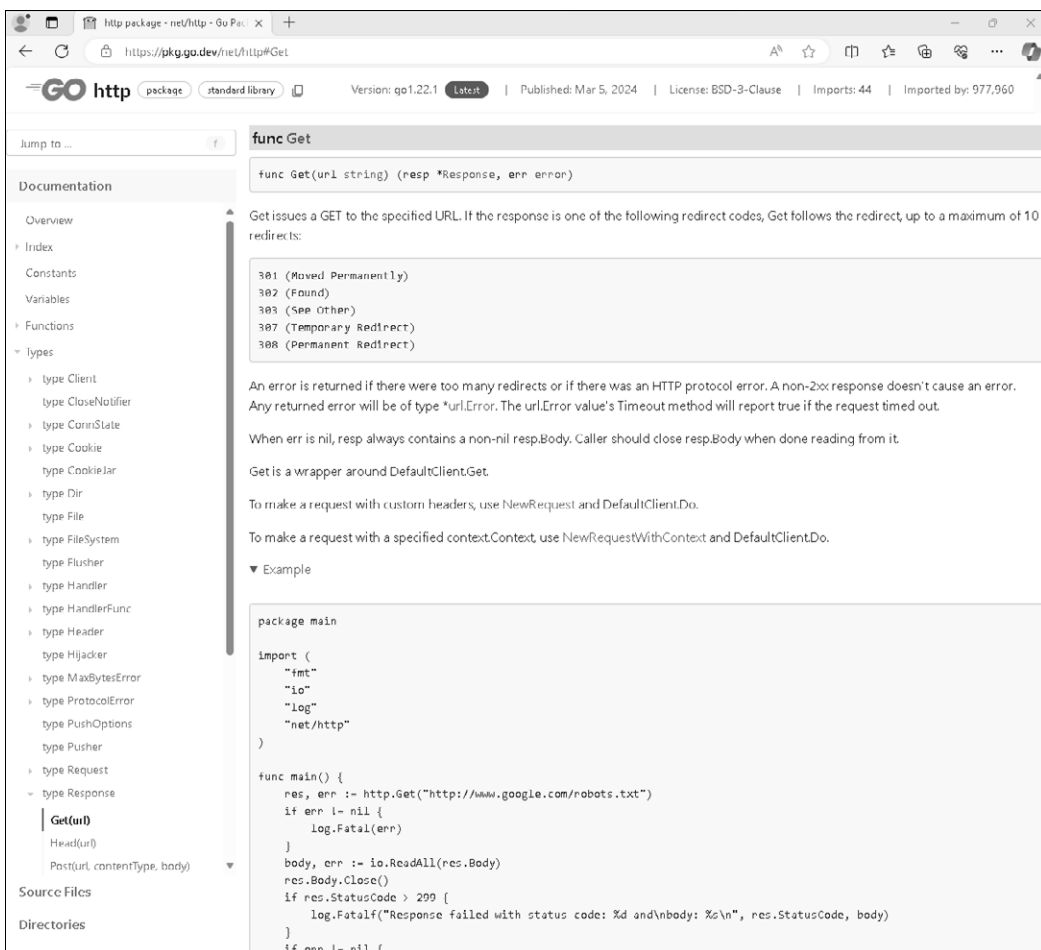


Figure 3.2 Presentation of the Test in the Documentation

The doctest Package in Python

In Python, you can use the *doctest* package to enter code within DocStrings and have the code executed at test time.

The following example shows the syntax in the documentation of the `factorial` function:

```
"""
This is the "example" module.

The example module supplies one function, factorial(). For example,

>>> factorial(5)
120
"""

def factorial(n):
    ...//Implementation was omitted
```

3.1.2 Creating Expressive Code

I already mentioned it in the introduction to this chapter: The more readable and self-explanatory a code section is, the fewer comments are needed to explain the meaning and purpose of the section to other developers. The basic principles of clean code or the use of software patterns can make an important contribution to clarity and readability.

Simple refactoring techniques, such as the *extract method* or *decompose conditional*, can be used, for example, to make details in the code that are difficult to understand clearer and thus avoid comments.

In the code example shown in Listing 3.4, a comment is helpful to understand quickly the nature of the check.

```
// Check if date is in summer
if (!aDate.isBefore(plan.summerStart) &&
    !aDate.isAfter(plan.summerEnd)){
    ...
}
```

Listing 3.4 Complex If Condition (Java)

However, extracting a method for the condition check makes the code clearer and easier to read, as shown in Listing 3.5. The comment can then be removed since the code is self-explanatory in this case.

```
if (isSummer(aDate)) {  
    ...  
}
```

Listing 3.5 Decompose Conditional Refactoring for Greater Clarity (Java)

3.1.3 Necessary and Meaningful Comments

As mentioned earlier, one of the goals in software development should be to create source code that is readable and expressive and for these reasons does not require explicit comments to explain code passages. However, in some situations, comments are important and helpful or are mandatory due to various framework conditions, such as licensing terms.

The following sections contain examples of source code comments to illustrate when the use of a comment in the code can be helpful or even necessary.

License Terms

Licensing information at the beginning of each source code file provides clarity about the conditions for the use and redistribution of the corresponding software.

Some companies and many open-source licenses, for example, require that the license text be provided together with the source code in order to provide transparent information about the applicable license terms and to ensure that the code is used accordingly.

In addition, licensing information ensures transparency with regard to company or developer copyright and helps comply with these rights.

In development environments, this kind of standard header can be configured for source code files so that the same comments can be automatically included in every file.

When commenting on license terms, you should ensure that the comment does not contain the complete license terms, only a reference to where the specific version can be found.

In the *Apache Commons* library for Java, for example, you can find the information shown in Listing 3.6 in each file.

```
/*  
 * Licensed to the Apache Software Foundation (ASF) under one or more  
 * contributor license agreements. See the NOTICE file distributed with  
 * this work for additional information regarding copyright ownership.  
 * The ASF licenses this file to You under the Apache License, Version 2.0  
 * (the "License"); you may not use this file except in compliance with  
 * the License. You may obtain a copy of the License at
```



```
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
```

Listing 3.6 Example License Terms for an Apache Commons File**Useful Information**

Short and clear information for a specific code passage can help you to understand it quickly. The example shown in Listing 3.7 demonstrates how a dense regular expression can be made easier to understand with a comment. Without the comment, someone might ask why only certain characters, and why precisely those specified, are permitted.

```
// regularExpression validates a "Vehicle Identification Number" (VIN)
// that it has a valid length and no illegal characters.
var regularExpression = regexp.MustCompile("^[A-HJ-NPR-Z0-9]{17}$")
```

Listing 3.7 A Regular Expression with a Comment (Go)

As an alternative to an explicit comment, renaming the variables in this example could further improve readability and make the comment superfluous again, as shown in Listing 3.8.

In this case, when reading the calling code, you shouldn't need to read the comment or jump to the corresponding entry in the documentation.

```
var vinLengthAndValidCharacterExpression =
    regexp.MustCompile("^[A-HJ-NPR-Z0-9]{17}$")
```

Listing 3.8 Variable Renamed for Easier Understanding (Go)

Renaming variables is certainly not possible in all situations; a variable name can rarely replace a longer, more complete, or more informative comment.

We could also combine both aspects and use a more expressive variable name and a short informative note, as shown in Listing 3.9, possibly with a reference to an external resource. In this case, the code is more readable, since what the regular expression checks is clear, and details of the expression can be looked up in the documentation.

```
// regularExpression validates a VIN that it has a valid length
// and no illegal characters.
var validVinExpression =
    regexp.MustCompile("^[A-HJ-NPR-Z0-9]{17}$")
```

Listing 3.9 Combining an Informative Comment and a Variable Name (Go)

Explanatory Comments

In some situations, code passages appear incorrect or at strange when reading. A short comment on why you have implemented this passage in this way and a brief explanation of your decision could help others to understand the block. Regardless of whether the passage could have been solved differently, the intention will become clearer.

The example shown in Listing 3.10 contains an excerpt from the standard Go library with a brief explanation of why the loop was written in the way it was.

```
// The loop condition is < instead of <= so that the last byte does not
// have a zero distance to itself. Finding this byte out of place implies
// that it is not in the last position.
for i := 0; i < last; i++ {
    f.badCharSkip[pattern[i]] = last - i
}
```

Listing 3.10 Example from the Standard Library with Explanatory Comment in Go

Clarifications

If code cannot be formulated more clearly, for example, due to confusing return values or other parameters, short, expressive comments can make your code more readable or easier to understand.

First and foremost, you should look for a solution with clearer code so that no comments are necessary. However, clearly not every code can be adapted. If you use an external library, for example, or if data is mapped into a data object, as shown in Listing 3.11, you still might need a comment for clarity.

```
type Subscription struct {
    // true/1 = switched on (no restriction)
    // false/0 = switched off (everything affected)
    PrivacyEnabledByUser    bool
}
```

Listing 3.11 Clarifying Values via Comments

Like any comment, explanatory comments can be wrong and should be replaced by meaningful code.

Warnings and Requirements

You can use comments to point out side effects that are caused by a change as shown in Listing 3.12.

```
if err != nil {  
    //Attention: this string/message is used for alerting in splunk  
    return fmt.Errorf("failed to doSth: %v", err)  
}
```

Listing 3.12 A Warning with a Comment on Possible Consequences (Go)

Even seemingly awkward design decisions or implementations can benefit from warnings or notes on their consequences so that overly diligent, refactoring-obsessed developers are slowed down and don't "make the code worse." Listing 3.13 shows an example.

```
//SimpleDateFormat is not thread safe!  
//We need to create each instance independently.  
SimpleDateFormat df =  
    new SimpleDateFormat("MMM dd yyyy HH:mm:ss");
```

Listing 3.13 A Warning about the Consequences of Multithreading (Java)

Side effects or requirements for calling code must be fully and clearly formulated in comments or documentation. The example shown in Listing 3.14 therefore clearly indicates the responsibility of the client.

```
// Body is the request's body.  
//  
// For client requests, a nil body means the request has no  
// body, such as a GET request. The HTTP Client's Transport  
// is responsible for calling the Close method.  
// ...  
Body io.ReadCloser
```

Listing 3.14 Documenting Requirements for User Code (Go)

Still to Do: TODO Comments

Many developers leave hints in the source code that indicate changes are still necessary in certain places, which they have not yet achieved for various reasons, as shown in Listing 3.15.

Such `//TODO` comments only make sense if the change cannot actually be made at the moment. The example shown in Listing 3.15 refers to a bug that must first be solved in a third-party system.

```
if testv.Builder() == "darwin-amd64-10_14" {  
    // TODO(#23011): When the 10.14 builders are gone, remove this skip.  
    t.Skip("skipping due to platform bug on macOS 10.14;  
        see https://golang.org/issue/43926")  
}
```

Listing 3.15 Notes on Dependencies (Go)

You should not use TODO comments to mark code blocks with notes indicating that no ideal solution has yet been found for this block! Some developers also consider every TODO in the code as a built-in bug because it will probably never be fixed.

For this purpose, most integrated development environments (IDEs) provide an automatic check or display of TODO comments prior to the check-in into a version control system (VCS). This check-in/check-out mechanism prevents unfinished code from being accepted by mistake.

3.1.4 Comments Not Needed

In addition to the comments that can and should be used in a reasonable manner, some comments in the source code are just superfluous. Ideally, well-structured and clearly structured code does not need any comments to explain itself.

Listing 3.16 shows an example of a superfluous comment that provides no added value.

```
public interface Validated {  
    //Empty  
}
```

Listing 3.16 Is the Interface Still Empty? (Java)

If you write a comment, it should definitely make sense and provide the reader with additional information. Every comment should be well considered and make things clear. Don't leave the reader with new questions!

In the example shown in Listing 3.17, the comment still leaves questions unanswered. Why is an attempt being made to disable the feature, or why can an error be ignored? Is the feature not important after all? What are the consequences if the feature cannot be deactivated?

For this reason, in this example, a helpful step would be to add at least one further comment with the reason why the feature should be deactivated.

```
@Override  
protected DocumentBuilderFactory createDocumentBuilderFactory(int  
    validationMode, boolean namespaceAware)  
    throws ParserConfigurationException {  
    DocumentBuilderFactory factory =
```

```
        super.createDocumentBuilderFactory(validationMode, namespaceAware);
    try {
        factory.setFeature("../features/..", false);
    } catch (Throwable e) {
        // we can get all kinds of exceptions from this
        // due to old copies of Xerces and whatnot.
    }
    return factory;
}
```

Listing 3.17 Comment in Java Apache CXF Source Code (Feature Name Shortened)

Avoiding Duplicate Statements via Comments

Each comment should provide additional information about the code. If the code is listed again in the comment in different words, the comment is simply superfluous.

If code seems to need “structuring” via comments, as shown in Listing 3.18, a better approach is to revise the code instead.

```
//do the calculation
result := thing.calculate()

//check the result if it is valid
isValid := result.IsValid()

//return error if not valid
if !isValid {
    ....return errors.New("result is not valid")
}
```

Listing 3.18 Duplicate Statement with Comments

In some projects, getter and setter methods are documented via comments due to project or company rules. What added value is the documentation shown in Listing 3.19 supposed to provide? This duplication can be omitted since it does not provide any additional information.

```
/**
 * Sets the bus
 * @param bus the bus
 */
public void setBus(Bus bus) {
    this.bus = bus;
}
/**
 * Gets the bus
```

```
* @return the bus
*/
public Bus getBus() {
    return bus;
}
```

Listing 3.19 Sample Documentation on Getter and Setter Methods in Apache CXF

Likewise, the statement shown in Listing 3.20 applies to the documentation of the fields of a class or for things that can be clearly read from the code.

```
//the license
private String license;
```

Listing 3.20 Documentation of a Field

History of a File

Back when version management was not as widespread as today, and source code was passed on via file repositories, developers often included the history of a file as a comment in the code. Fortunately, these times are over, and version management systems such as Git have become established practice. History comments have thus become unnecessary and should therefore no longer be used.

Development environments often automatically generate documentation or comment templates. You should therefore evaluate whether each automatically introduced comment is necessary and what additional information it provides.

In turn, you must also check whether the code is comprehensible and clearly formulated. You may need to modify it or add useful comments so that the code is better understood and can be used correctly.

Commented-Out Code

Even though commented-out code is not a comment in the traditional sense, I would like to discuss this topic here.

Commented-out code often remains in the code base, even if it is no longer needed. The reasons for commenting out code are usually unclear, which can confuse other developers reading the code: You cannot judge whether the commented-out code is still relevant or not. The code may have been replaced by a better alternative and simply forgotten at this point. Such ambiguities make it difficult to understand code.

If code is commented out, the reason for this should either be noted as a comment or the code should be deleted completely. Commented-out code contributes to what's called *technical debt*, which describes the additional costs and effort caused by an unstructured, outdated, or insufficiently maintained code base.

3.2 Documenting the Software Architecture

The documentation of a software and especially its architecture fulfills several purposes.

Robert C. Martin's quote that the truth lies in the code is true, but the truth usually involves more than just looking at a section. The documentation of a software architecture has a greater focus; it represents and documents more complex systems.

"The code doesn't tell the whole story." —Simon Brown

In their book *Documenting Software Architectures: Views and Beyond*, Paul Clements and his coauthors have identified three task areas for the documentation of software architectures:

- ▶ Development
- ▶ Communication
- ▶ Basis for analysis and further development

Documenting Software Architectures: Views and Beyond

In 2003, the first edition of *Documenting Software Architectures: Views and Beyond* by Paul Clements and colleagues on the subject of architectural documentation was published. The second edition, published in 2010, was revised and extended and is often described as a groundbreaking, indispensable reference work.

The book not only presents the importance of architecture documentation for communication but also considers it instrumental for supporting decisions, minimizing risks, and ensuring the long-term maintainability of your software.

If new developers or software architects join an existing team, documentation about the architecture helps onboard new members to the existing solution and its structure. With a corresponding presentation, new members and even interested external parties can obtain a structured overview and expand or deepen their knowledge of the software.

In addition to this type of knowledge transfer about the software, the documentation serves to ensure more efficient communication between the members of a development team or across its boundaries.

One of the main tasks of a software architect is to communicate solutions that have been developed or are yet to be developed. The software architecture documentation supports this communication and provides a clear and precise way of exchanging information. In addition to the specific solutions, it also documents the decisions made and the associated justifications so that they can be retraced later, for a deeper understanding.

In many cases, the architect himself is one of the main consumers of the documentation. In most cases, the decisions made and their justifications are interrelated in a complex network, and remembering every decision and its justification is impossible without informative documentation.

Decisions that have been documented can form the basis for future developments and influence them. Cumbersome design decisions can be revised; good decisions can be retained and reapplied during implementation.

In addition to successful decisions, negative experiences can also be documented. In this way, you can learn from your mistakes, and transparent documentation means you can develop more successfully in the future. Mistakes can and should also have an influence on future development and the associated decisions.

Fundamental decisions made early on about the software design within a project should be documented and communicated as soon as possible. This approach prevents similar considerations from being made more than once and possibly leading to slightly different approaches at different points in the application.

Good documentation leads to long-term maintainability of software. Knowledge is passed on and stored and can be used in future considerations.

3.2.1 Documenting Quality Features

In addition to the technical requirements (for example, that some code performs calculations correctly), most software must fulfill further quality features. Software that can generate the right result but can't deliver it correctly or at the right time has little or no added value for a user. Non-technical quality characteristics of a software solution may include the following:

- ▶ Performance
- ▶ Stability
- ▶ Security
- ▶ Maintainability
- ▶ Capability for incremental updates

High performance can be achieved, for example, by parallelizing tasks or with optimized remote calls that generate a smaller volume of data or are executed less frequently. The possible solutions are varied and depend on the area of application.

If software is to remain maintainable, it can be subdivided into individual components, each of which fulfills its specific task and can be developed independently of one another. Applying the single responsibility principle can help.

All of these quality features must be considered and addressed by the software architecture. Accordingly, these features must be included as requirements in the software architecture documentation, and the selected solution must be documented.

Three questions should be answered for each quality feature:

- ▶ What quality feature was addressed with the architecture?
- ▶ What solution was chosen to solve the challenge?
- ▶ Why does the solution ensure the quality feature?

3.2.2 Rules for Good Software Architecture Documentation

In *Documenting Software Architectures: Views and Beyond*, Paul Clements and his coauthors set out seven rules for good software architecture documentation. Let's now take a closer look at these rules.

Write from the Reader's Point of View

All documentation should be created with a certain target group in mind, and it is precisely for this target group that the documentation should be written.

The authors of the book refer to Edsger Wybe Dijkstra who claimed to spend two hours thinking about how he could make a single sentence in the documentation clearer. He was of the opinion that comments are read by many people and that these two hours quickly pay off if each person can save one to two minutes of confusion.

Dijkstra's principle can be applied to source code, which is usually written once, but read several times.

Target group-oriented documentation is generally not only read and understood but can serve as a reference later. If the reader does not receive the information they need, the documentation will play no further role for them.

Some tips to bear in mind when writing documentation include the following:

- ▶ Know the readers: Uninformed assumptions about the readership should be avoided. Know what is expected of the documentation. A brief exchange, if possible, can be helpful.
- ▶ Structured documentation: Information should be prepared in a structured manner and appear in appropriate places.
- ▶ Use clear terminology: The documentation is read by a wide range of people, sometimes even by people not familiar with the specialist terminology or jargon. When special terms are used, they should therefore be clearly defined, for example, in a glossary.
- ▶ No overloading with acronyms: Acronyms (i.e., short words made up of the first letters of their components, such as "API") sometimes make communication within a

project more efficient. However, in documentation, these abbreviations should be used with caution and should always be listed in a glossary.

Avoid Unnecessary Repetition

The same documentation should not be repeated in multiple places. You run the risk that, due to adjustments or different formulations, the information can vary one day. Which part is decisive and actually represents the correct information may not be clear. Repetition may be useful if the same information is presented in several places with a slightly different perspective or in a slightly different focus.

Avoid Ambiguity

As soon as documentation is not formulated clearly enough and can be interpreted differently, ambiguities will arise. This problem leads to differing opinions and false conclusions.

This kind of problem can be avoided or mitigated with the help of standardized formulations or presentations.

Requests for Comments: Indicate Requirement Level

The *requirement levels* defined for RFCs are an example of standardized text formulations. They are used, for example, to define the meaning of the expressions “must,” “must not,” “should,” “should not,” and “may” within the RFC documents.

For more information, see <https://datatracker.ietf.org/doc/html/rfc2119>.

Regardless of which format is used, the corresponding meaning must be clearly defined. Classic *box-and-line diagrams*, such as those often found on whiteboards, can easily cause misunderstandings.

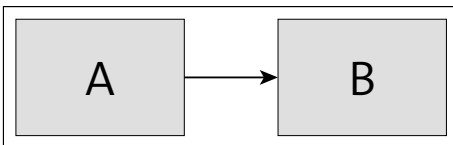


Figure 3.3 Box-and-Line Diagram

As shown in Figure 3.3, for example, the arrow can mean several different things: Does A call B synchronously? Does A instantiate type B, or is a message sent asynchronously from A to B? Is class A possibly a derivative of class B?

In many cases, the best approach is to use widespread, standardized representation formats, for example, Unified Modeling Language (UML) or the C4 model, to create clarity about the representation. Detailed descriptions of both formats will be provided later in this book.

Use a Standardized Structure

The documentation should be created using a predefined structuring template—i.e., using a template, which already specifies sections that are filled with the actual documentation.

This is an advantage for the reader, as the clear structure makes it easier to find information. The template can also be used in more than one project. This means that you can familiarize yourself with new projects and find your way around them more quickly.

In addition, the predefined structures also provide many advantages at the time of creation. Information can be added directly to the respective section as soon as it is created—regardless of whether upstream areas have already been filled or not. In this way, the template serves as a suitable repository for the snippets of information, which together make more sense.

At all times, there is transparency about how much documentation is available and whether sub-areas still need to be filled. If sections are not filled, information is missing; if all sections are complete, the documentation is accordingly complete.

For example, the *arc42* template has become widespread as a standardized structure.

Keep Track of Decisions and the Underlying Reasons

In software development, many individual decisions together lead to a software architecture. Every decision should be made consciously and ultimately result in a well thought-out, suitable software architecture.

Unconsciously made decisions often lead to unclear structures within the application or to unintended dependencies in the source code or in external systems. Changes can be more difficult to implement, and the stability or security of the software may suffer.

Overall, unconscious decisions in a software architecture can lead to a variety of problems and should therefore be avoided at all costs.

Some of the most important decisions that need to be made are, for example:

- ▶ Programming language or technology stack
- ▶ Chosen architectural style
- ▶ Identification of components and interfaces
- ▶ Data modeling
- ▶ Integration and interoperability
- ▶ Security guidelines, mechanisms, protocols, etc.

Every important decision made must be recorded in the documentation with the corresponding justification. It is also best to list the alternatives considered and the arguments in favor of the chosen solution. This means that the decision can be retraced at a later date and the reasons why an alternative is not used can be reviewed.

Keep the Documentation Up to Date, But Not Too Up to Date

For documentation to be useful, it must be up to date and correct. Outdated documentation that does not represent the actual situation causes confusion and will be avoided by the target group as soon as they discover the issue. Questions that could be answered by the documentation then require research elsewhere and generate additional work for the readership.

All documentation should be up to date. But in return, not every innovation requires immediate documentation. If alternative approaches are evaluated or new approaches are quickly discarded, the documentation can be delayed to minimize unnecessary effort. The first thing to consider in this context is how sustainable an innovation is.

Updating the documentation should be a fixed point in the development process. For *Scrum*, for example, the term *definition of done* has become popular. This definition serves as a checklist of items that must be completed before a task can be considered finished. One item on the list should be updating the documentation.

Check the Documentation for Suitability for Use

Every documentation has its own specific target group. Only this group of people should give feedback on the content and decide whether the information their members are looking for is included in an appropriate way.

For this reason, the documentation should be read and reviewed regularly by the relevant target group.

3.2.3 arc42 for the Complete Documentation

In German-speaking countries, *arc42* has spread as a structured architecture description. Initially developed by Gernot Starke and Peter Hruschka, this document structure often serves as the basis for a separate architecture documentation or is sometimes as a process model for an architecture design. These open-source templates can be used directly in your own projects in various formats. For each section, the template also provides a brief description of what should be included in the corresponding section.

Although *arc42* is open source and available in several languages, it is not yet widely used in English-speaking countries. However, the adoption of the iSAQB (International Software Architecture Qualification Board), an international association that offers a standardized training program and certifications for software architects, is certainly contributing to its further spread. The C4 model by Simon Brown, which is often seen as an alternative, will also be presented later.

arc42 provides a clear, simple, and efficient structure for the documentation and communication of software architectures and comprises twelve sections that cover the most important aspects of an architecture or application. Figure 3.4 shows the structure of the entire document template.

Not all structure points are always required, and some chapters can be omitted in documentation based on arc42.

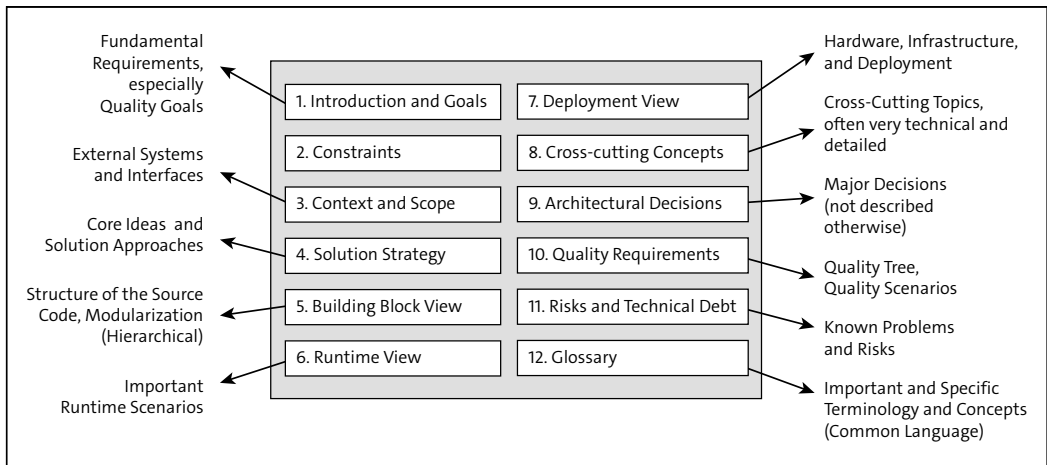


Figure 3.4 Overview of arc42 (Source: <https://arc42.org/overview>)

arc42 Documentation and Examples

The arc42 templates can be found at <https://arc42.org>. Either blank or prefilled templates are available for download; the prefilled documents themselves represent an extended documentation of arc42 itself.

In addition, you will find detailed examples of various projects at <https://arc42.org/examples>.

We'll use the HtmlSanityCheck sample application by Gernot Starke is used for the presentation. You can find it at <https://hsc.aim42.org>.

The graphics used are licensed as follows: <https://creativecommons.org/licenses/by-sa/4.0/deed.en>.

The following sections walk through the structure points of the arc42 template.

Introduction and Objectives

The introductory section describes the basic requirements and objectives of the system as well as the intention and motivation behind the development of the software solution.

In addition to the technical requirements and the basic business objectives, the most important quality requirements for the architecture are described, as well as all important stakeholders and their expectations about the software.

Constraints

Most software architectures are not created without specifications or framework conditions. In many projects, special attention must be paid to data privacy or other legal framework conditions, for example. Some companies also restrict the choice of technology or the technology stack used.

This section lists all the points that restrict the team's freedom with regard to design and implementation decisions.

If many restrictions exist, differentiating between technical, organizational, and political framework conditions may make sense.

Context and Scope

Every system or architecture must differentiate itself from all other systems and must define a clear context and the corresponding responsibilities of the system. These delimitations also help to define and display interfaces to other external systems.

A context should always be documented from a business- or domain-related perspective. In the event that infrastructure or hardware play an important role, a technical perspective can also be presented.

The contexts and their system dependencies can be presented as text or with the help of diagrams, for example with UML diagrams, as shown in Figure 3.5.

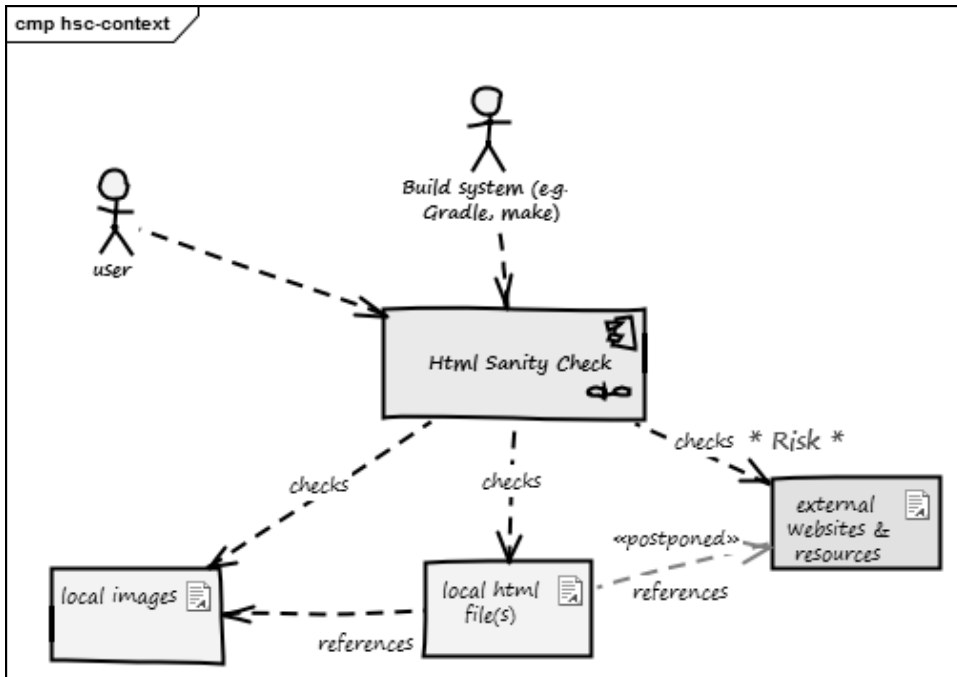


Figure 3.5 HtmlSanityCheck Sample Application: Context

Solution Strategy

This chapter provides an overview of all the fundamental decisions and approaches that determine the design and implementation of the system. They form the basic framework of the application and influence many other decisions and implementation rules.

Let's say, for example, you've selected a Go-based technology stack in the solution strategy. This decision also influences later decisions on the selection of external libraries.

The list of solutions and decisions usually contains the following items:

- ▶ Technology decisions
- ▶ Representation of the modularization at the highest level of abstraction
- ▶ Use of formative design or architectural patterns
- ▶ Decisions on the most important quality features
- ▶ Relevant organizational decisions

Building Block View

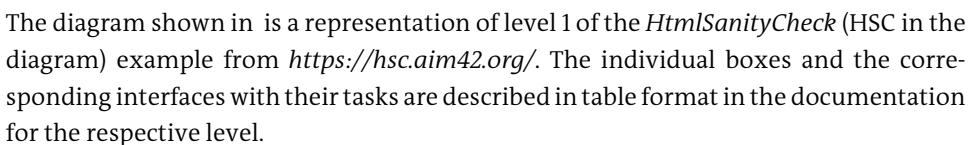
The building block view can serve as an initial overview and shows the static breakdown of a system into its components and their relationships with each other. The included modules, components, subsystems, classes, interfaces, packages, libraries, frameworks, layers, partitions, and more are best listed using diagrams. This view refines the context view in which the system is displayed as a large box.

The abstract representation of the structure of the system can help you obtain a better overview of the source code and can serve as a basis for communications at an abstract level. Implementation details do not play a role in this context.

A distinction can be made in the presentation between *white boxes* and *black boxes* as well as between multiple levels with different degrees of detail.

In the first level, the most important subsystems, components, or parts of the system are listed, and in the second level, if necessary, these elements can be described in more detail in the white boxes.

Not every component from level 1 must be shown in the second level.

The diagram shown in  is a representation of level 1 of the *HtmlSanityCheck* (HSC in the diagram) example from <https://hsc.aim42.org/>. The individual boxes and the corresponding interfaces with their tasks are described in table format in the documentation for the respective level.

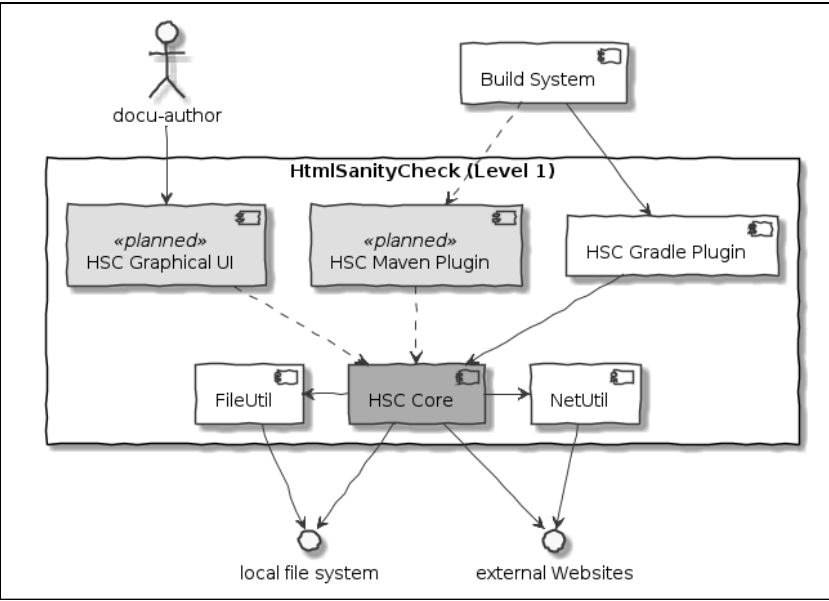


Figure 3.6 HtmlSanityCheck Sample Application: Level 1 (Source: <https://github.com/aim42/htmlSanityCheck>)

In level 2, the HSC core components are shown in detail using a white box, as shown in Figure 3.7. Here too, the individual components are explained in the documentation in a table with their corresponding tasks.

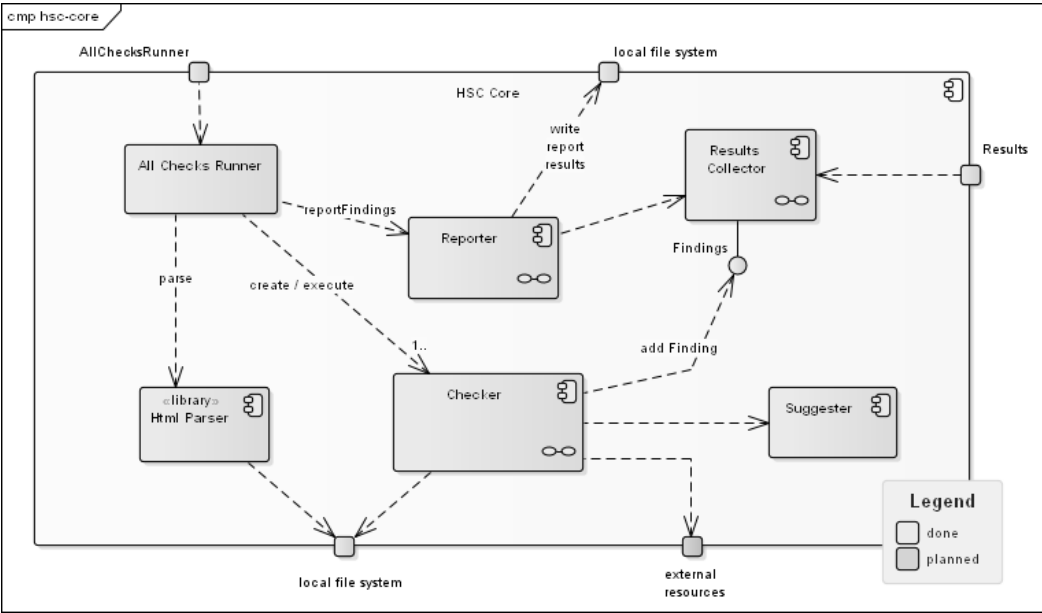


Figure 3.7 HtmlSanityCheck Sample Application: Core Display on Level 2 as a White Box

Runtime View

The runtime view shows important or critical concrete processes, tasks, and relationships between the building blocks of the architecture. These building blocks can be internal or external components.

Scenarios are used to deepen the knowledge of the individual components and their interactions with other components. Individual scenarios can be described using text or diagrams.

Deployment View

Software is executed on hardware. The distribution view shows the technical infrastructure with all environments, computers, processors, networks, and network topologies and assigns the software modules accordingly, as shown in Figure 3.8.

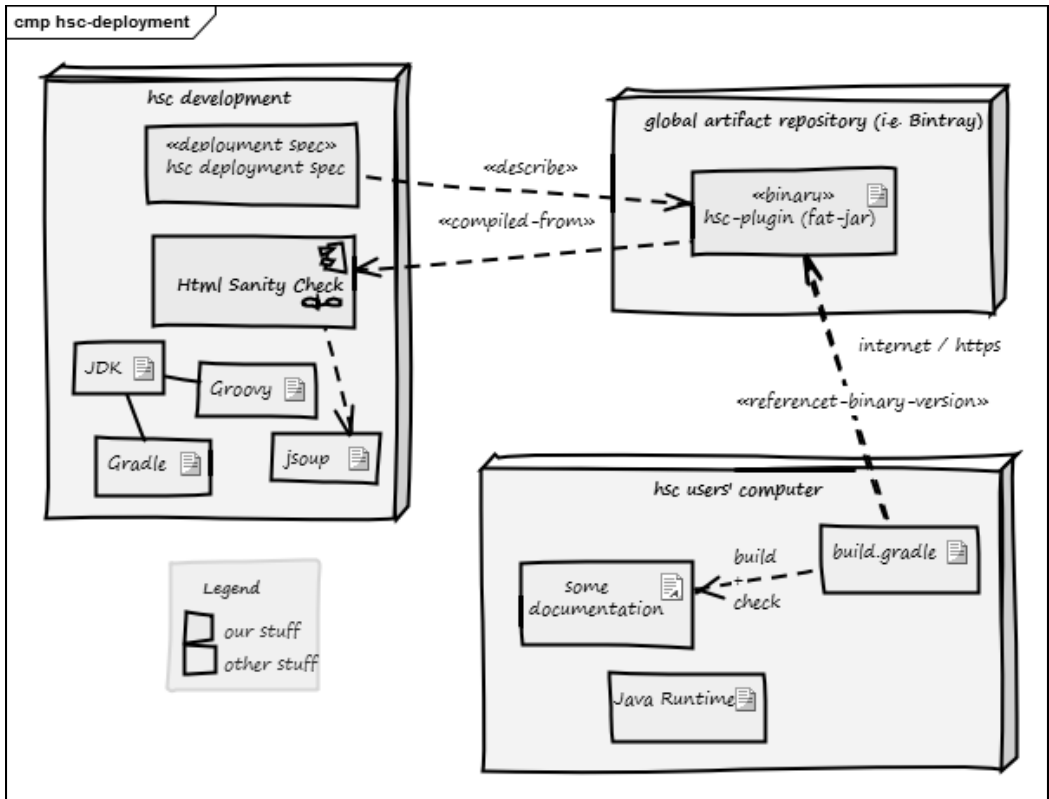


Figure 3.8 HtmlSanityCheck Sample Application: Distribution View of the Application

CrossCutting Concepts

This section documents key regulations, solutions, and concepts that affect multiple building blocks. These elements can involve widely different topics, such as the following:

- ▶ Models, especially business models
- ▶ The architecture style used
- ▶ Architecture or design patterns used
- ▶ Rules for the specific use of technologies or libraries
- ▶ Implementation rules

Architectural Decisions

Architecture decisions should always be clearly thought through, consciously made, and documented with sound reasons.

In this section, the important architecture decisions are documented with a corresponding justification, and the documentation of rejected alternatives should also be included here.

Quality Requirements

Every quality requirement can influence the software architecture. For this reason, knowing your quality requirements and presenting them in a transparent way are important.

In this section of the template, the most important quality objectives already listed in the introduction are presented as completely as possible with all other quality requirements in a what's called a *quality tree* with scenarios. This tree structure is defined in the Architecture Tradeoff Analysis Method (ATAM), a systematic evaluation method that helps identify risks and understand the compromises between competing quality attributes, such as performance, security, and modifiability, in a software architecture early in its development lifecycle.

The listed scenarios are supposed to illustrate how a system will respond if specific events occur as well as concretize any vaguely formulated requirements as well as possible.

For example, performance requirements could be defined with specific values: “The system must return a response in a maximum of 100 ms.”

Risks and Technical Debt

Every piece of software contains potential problems or immature areas with which the development team is not yet satisfied. In this section, these problems are presented in a prioritized list and are therefore transparent.

The items in the list can already be provided with suggested solutions so that they can be scheduled directly.

Glossary

The glossary explains all important domain and technical terms used for communication with individual stakeholders. Generally known acronyms such as HTTPS or REST do not need to be listed here.

A glossary can also be helpful as a reference for translations in international environments.

3.3 Representing Software in Unified Modeling Language

The *Unified Modeling Language (UML)* is a modeling language that was developed, among other things, to improve communications between software developers and make it easier for them to understand the software. The graphical language elements of UML can be used to specify, design, document, and visualize software systems.

The first version of UML was developed in the 1990s, after several modeling languages and methods for object-oriented software development had already been created. Grady Booch, Ivar Jacobson, and James Rumbaugh were instrumental in defining the language during their time at Rational Software and are therefore considered the “fathers” of UML.

In 1997, UML was handed over to the *Object Management Group (OMG)* and accepted as standard. Since then, UML has been continuously developed.

In today’s software systems, UML is a widely used tool for modeling and for standardized documentation, even if the full range of functions of the language is rarely used. The familiar graphical notation is only one aspect of the representation of the models described by UML.

The UML 2.3 specification defines 14 different diagrams, which can be roughly divided into two main categories:

► Structure diagrams

- Class diagram
- Object diagram
- Package diagram
- Component diagram
- Deployment diagram
- Profile diagram
- Composite structure diagram

► Behavior diagrams

- Activity diagram
- Use case diagram

- Interaction overview diagram
- Communication diagram
- Sequence diagram
- Timing diagram
- State diagram

UML diagrams are often used in modern software development. For this reason, the most well-known and widely used diagrams are presented in the following sections. This discussion will enable you to create your own diagrams; however, for reasons of space, I cannot offer a complete introduction to UML introduction in this book. For the diagrams in this book, I also use the UML notation for visualization. The widely used, easy-to-use, and (unlike many commercial tools) lightweight open-source modeling tool *PlantUML* allows you to create graphical UML diagrams from text files, usually directly from the development environment. I will present this tool in more detail in Section 3.5.2.

PlantUML

PlantUML is an open-source component for creating diagrams from text files. You can find PlantUML and its documentation at <https://plantuml.com/>.

3.3.1 Use Case Diagram

A use case diagram can illustrate the functions or actions of a system and their interactions with users or other systems. The objective is to show the functional requirements of the system and visualize how a user can interact with the system to achieve their goals.

Three elements are defined, as shown in Figure 3.9:

- Actor: An *actor* is a user who interacts with a system. Actors are depicted as stick figures. They can be people, but also external systems.
- Use case: This is a single action that can be performed by a user. It is displayed as an ellipse and can be subdivided by corresponding relationships or extended by means of inheritance.
- Context: The system context is drawn as a rectangle and encloses the use cases defined in it.

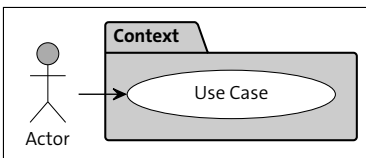


Figure 3.9 Elements of a Use Case Diagram

Use case diagrams do not contain any further details, such as information on the order in which the individual interactions must be carried out. They are primarily intended to provide an overview of the system and its context and are usually supplemented by a textual description.

Figure 3.10 shows examples of use cases for a training company in which a customer can book training courses, either online or by telephone. Once a booking has been received, the reservation is confirmed by a member of the training management team, who must first book a corresponding room. The appointment confirmation includes dispatching a confirmation message to the participant.

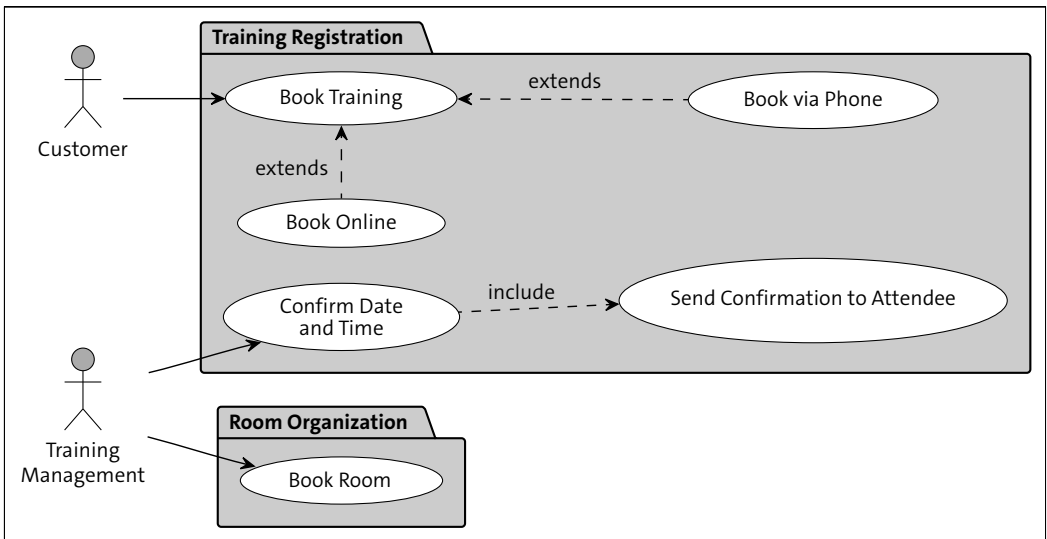


Figure 3.10 A UML Use Case Diagram

3.3.2 Class Diagram

Class diagrams are by far the most frequently used diagrams in object-oriented programming. They are used for the graphical representation of classes and interfaces and their relationships with each other.

Each class and each interface are displayed as a rectangle that contains the attributes and methods of the class or interface in addition to the name.

Figure 3.11 shows the PlantUML modeling tool for classes as it is used in this book. The following information can be displayed for a class or an interface:

- ▶ **Name:** The name of the class or interface.
- ▶ **Type:** Whether it is a class, an abstract class, or an interface. In PlantUML, classes are labeled with "C"; interfaces, with "I"; and abstract classes, with "A."
- ▶ **Stereotype:** The purpose of a class can be represented with *stereotypes*. Some types, such as the entity type used in our example, have already been defined in UML 2.1.

Classes marked with `entity`, for example, are classes for implementing business logic.

- **Generics information:** Some programming languages allow classes to be parameterized. In Java or Go, for example, these classes are called *generics*. With this addition, called *template arguments*, the corresponding information can be specified.
- **Attributes:** A list of all attributes with their corresponding visibilities.
- **Methods:** Like attributes, all methods are listed with their visibility, parameters, and return types.

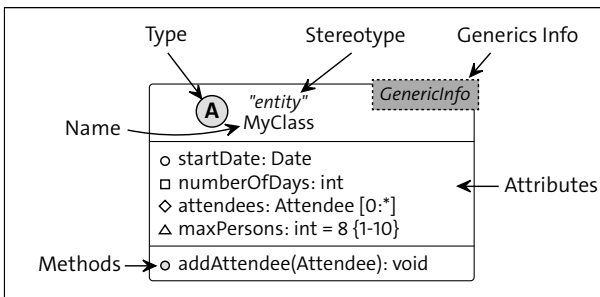


Figure 3.11 Representation of a Class Using PlantUML

The textual representation of the individual attributes of a class diagram typically adheres to the following schema in various UML tools (and in accordance with the UML standard):

```
[Visibility] [/] name [: Type] [ Multiplicity ] [= default value] [{property value*}]
```

In PlantUML, the visibility of the elements is defined accordingly within the text files via individual characters and visualized via symbols in the graphical representation. The PlantUML syntax uses the following characters and symbols:

- A circle corresponds to public or “+”
- A square corresponds to private or “-”
- A diamond corresponds to protected or “#”
- A triangle corresponds to package or “~”

Method signatures are described and visualized in the same way. An important point in this context is that the return value is included at the end of the signature, not at the beginning as in many programming languages:

```
[Visibility] name [{Parameter}]] [: Return type] [{property value*}]
```

The example shown in Figure 3.12 is a class diagram for the administration of a training session, in which relationships between the classes are also shown.

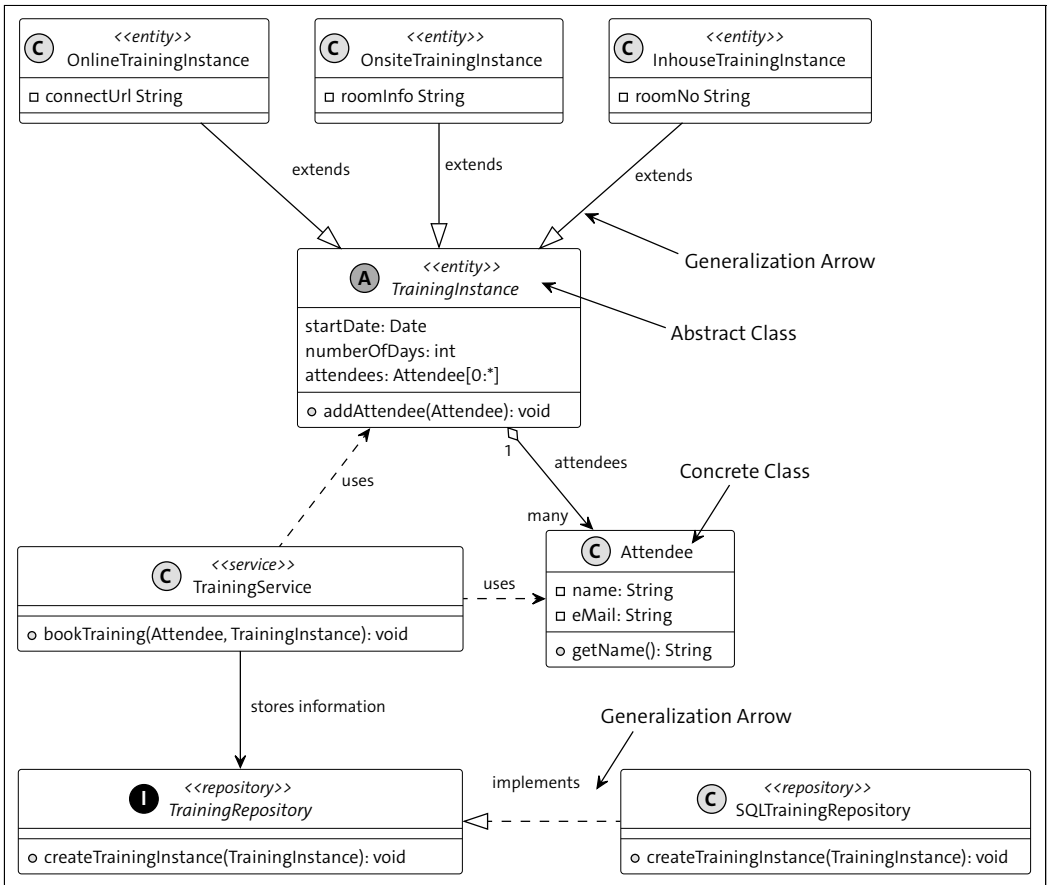


Figure 3.12 Example Class Diagram

This example contains the abstract `TrainingInstance` class, which is extended by the `OnlineTrainingInstance`, `OnsiteTrainingInstance`, and `InhouseTrainingInstance` classes. This generalization is represented by a straight, solid arrow that leads from the specific to the general implementation and has an unfilled arrowhead.

Interface implementations are also displayed using a generalization arrow, but the line of the arrow is dashed. In this example, the `SQLTrainingRepository` class implements the `TrainingRepository` interface.

Relationships between classes, *associations*, are represented by simple arrows. For example, `TrainingService` uses `TrainingRepository` to store data. Multiplicities are often used to specify how many of the referenced objects are related to the other objects. This information is then written to the arrow as additional information. In our example, `TrainingService` has exactly one `TrainingRepository` instance and therefore is not specified here.

One or more participants are required to carry out a training session, which is implemented as `TrainingInstance` in our example. For this reason, the relationship between `TrainingInstance` and `Attendee` has multiplicities, and this information is added to the relationship arrow.

The information that at least one participant must be present can be mapped with UML using special types of association, namely, composition and aggregation, as shown in Figure 3.13.

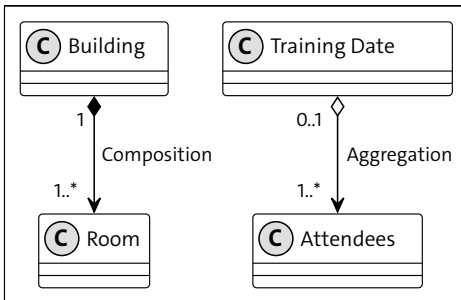


Figure 3.13 Composition and Aggregation

Let's briefly explore these two association types:

- A *composition* describes a relationship between a whole and its parts in which the whole cannot exist without its parts. A classic example is how a building cannot exist without at least one room.
- If the whole can also exist without its parts, the relationship is an *aggregation*. This relationship is possible, for example, when a training course is conducted: Even if a `TrainingInstance` without an attendee makes little sense, it must be possible to create a `TrainingInstance` without an attendee during planning.

An aggregation is shown with an unfilled diamond, which is located at the end of the relationship on the whole. In a composition, a filled diamond is used instead, as shown in Figure 3.13.

3.3.3 Sequence Diagram

Sequence diagrams represent interactions in a system and model the exchange of messages between different objects.

Each object has what's called a *lifeline* in the diagram, which serves as the starting point for sending or receiving a message and is shown as a dashed line below the object. The exchange of messages between the objects is represented by arrows.

A branching or decision syntax is not provided for in sequence diagrams, and a variant of a sequence is always shown. If multiple variants of a process are required, multiple diagrams must be created accordingly.

The example shown in Figure 3.14 is a sequence diagram for booking a training. A customer, whose lifeline is not shown for better readability, first triggers a new booking in the booking system.

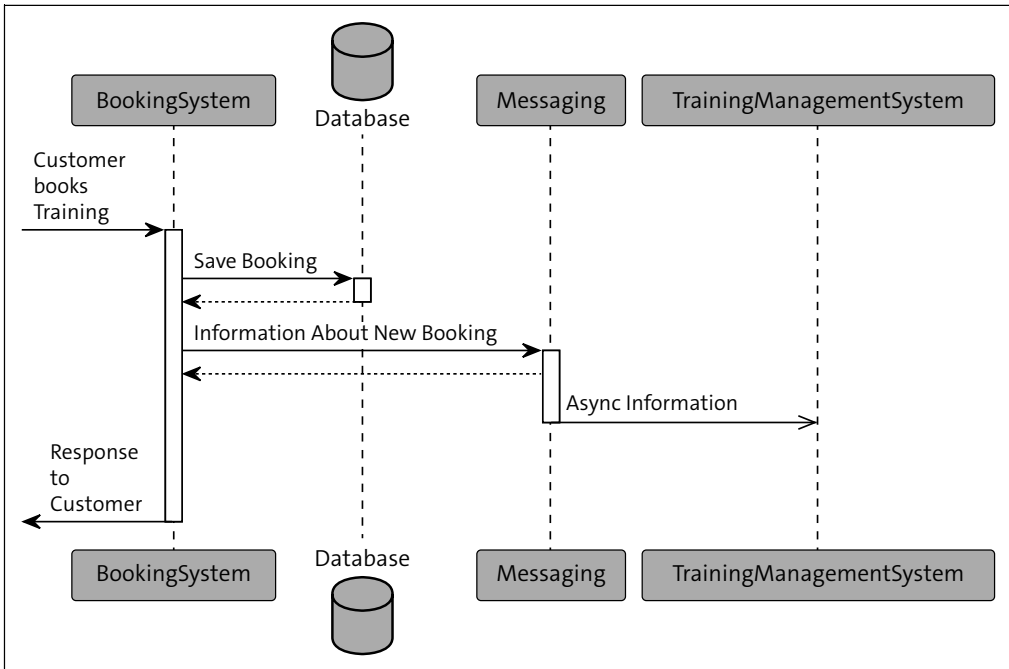


Figure 3.14 Sequence Diagram Example

The system then saves the data in a database and generates a message in the messaging system that a new booking has been received. The process is completed for the customer for the time being. However, the message with the information that a new booking has been created is transmitted asynchronously to the `TrainingManagementSystem`.

The distinction between synchronous and asynchronous messages is made via the tip of the arrow during message transmission: If the arrowhead is filled in, it is a synchronous message. If the arrowheads are not filled in, this message is an asynchronous message, as shown in our example, on the right, when a message is redirected to the `TrainingManagementSystem`.

3.3.4 State Diagram

Each object in a system can assume different combinations of internal information and thus different states during its life cycle. The possible states and their transitions can be visualized using a UML state diagram.

Each state diagram has a start point and an end point, which are also displayed as such. Unlike the start point, the end point has a border.

An object's states and their possible transitions are listed between "Start" and "End." Transitions are represented by arrows with an optional label; states are visualized as rectangles with rounded corners.

Figure 3.15 shows the possible states for a training date in a training administration system as an example. A training date can therefore have the status "On Offer," and as soon as a booking is received, the status changes to "Scheduled." If no booking is received by the deadline, the status changes to "Canceled" and so on.

In addition to the state diagram shown for a *behavioral state*, in which the behavior of an object is modeled, you can also use UML to model what are called *protocol states*.

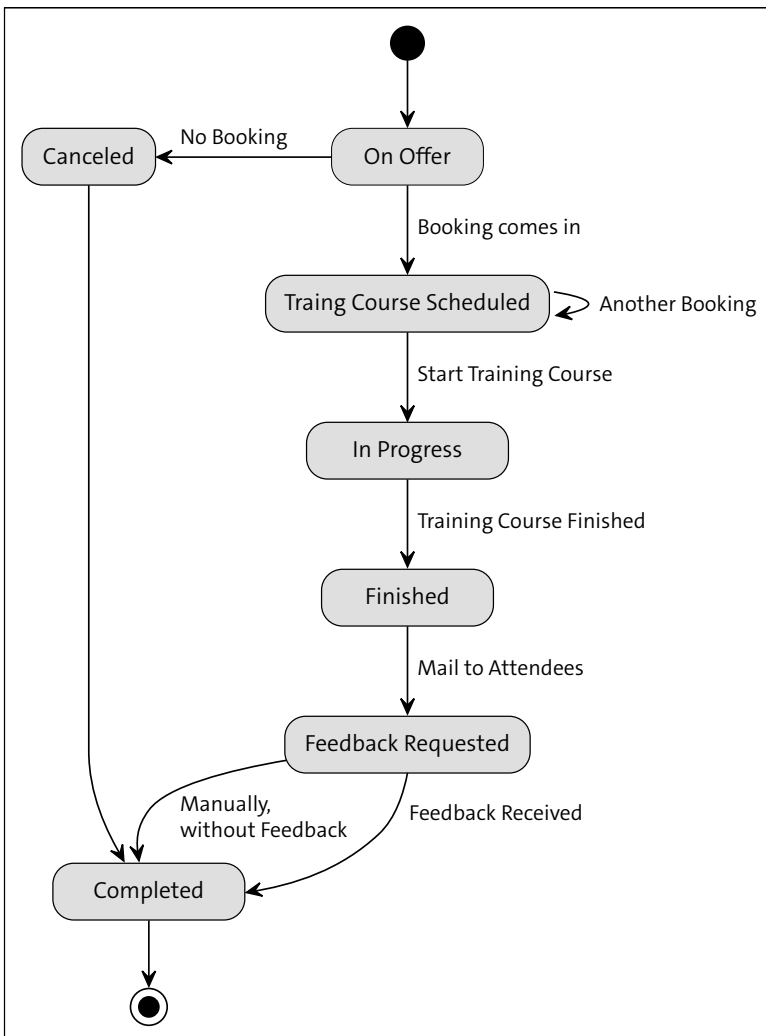


Figure 3.15 TrainingInstance State Diagram (Behavioral State)

Protocol states describe the permissible use of an object or its behavior, as in our example shown in Figure 3.16. The diagram specifies the possible methods that can be called in addition to the states.

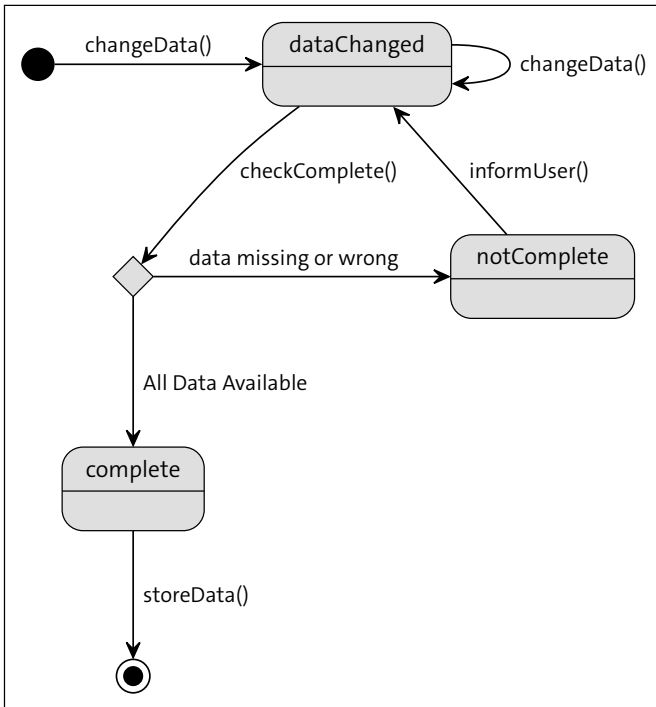


Figure 3.16 Protocol States with State Diagram

3.3.5 Component Diagram

A *component diagram*, another structural diagram in UML, represents components and their relationships to other components in the system.

In UML, the term *component* refers to a module that consists of several classes and can be regarded as an independent system or subsystem. Interfaces define the connections to other components of the rest of the system.

Component diagrams can be used to visualize software systems at a high level of abstraction in order to provide the best possible overview of all components involved. This representation is particularly useful in cloud or microservice-based applications, where multiple self-contained components interact with each other.

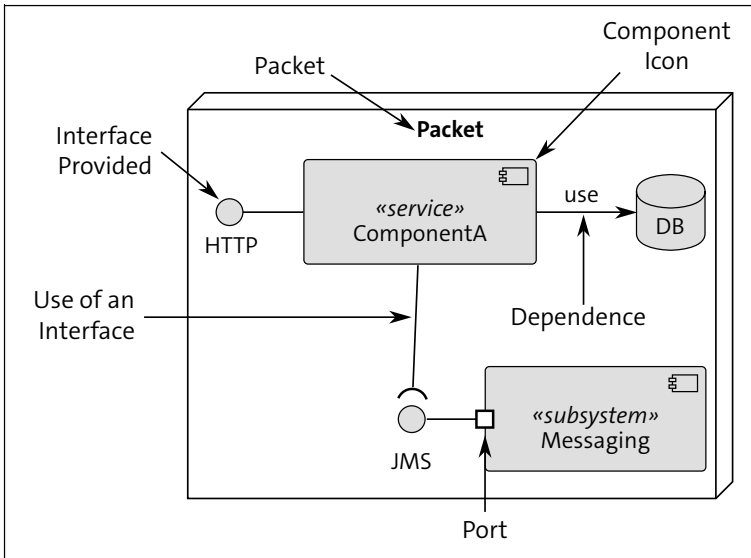


Figure 3.17 Components of a Component Diagram

The diagrams consist of several components:

- ▶ **Component:** The components are displayed as rectangles with a small component symbol. Some display variants use the `<<component>>` stereotype for marking.
- ▶ **Package:** Several components can be combined into one package to illustrate their close relationship. Packages are also displayed as rectangles or frames.
- ▶ **Interface:** Each component can provide one or more interfaces for the communication with other components. These interfaces can be used as interaction points. Available interfaces are represented by a circle with a connecting line. Used interfaces are documented with a semicircle and a corresponding line.
- ▶ **Port:** Sometimes, calls to a provided interface are delegated to internal classes, which is visualized via ports represented as squares.
- ▶ **Dependency/relationship:** As in other UML diagrams, dependencies are drawn as a line between the components. As before, the arrow indicates the direction of the dependency.

Figure 3.18 shows a component diagram for a training administration software. The diagram shows three independent application components and a cloud-based messaging solution that are connected to each other.

A booking can be triggered via the `Booking` component, which is redirected to the `Administration` component as an event using the `Messaging` solution. The `Print` component is called via an HTTP endpoint and produces documents for the training course, which are saved in a file repository.

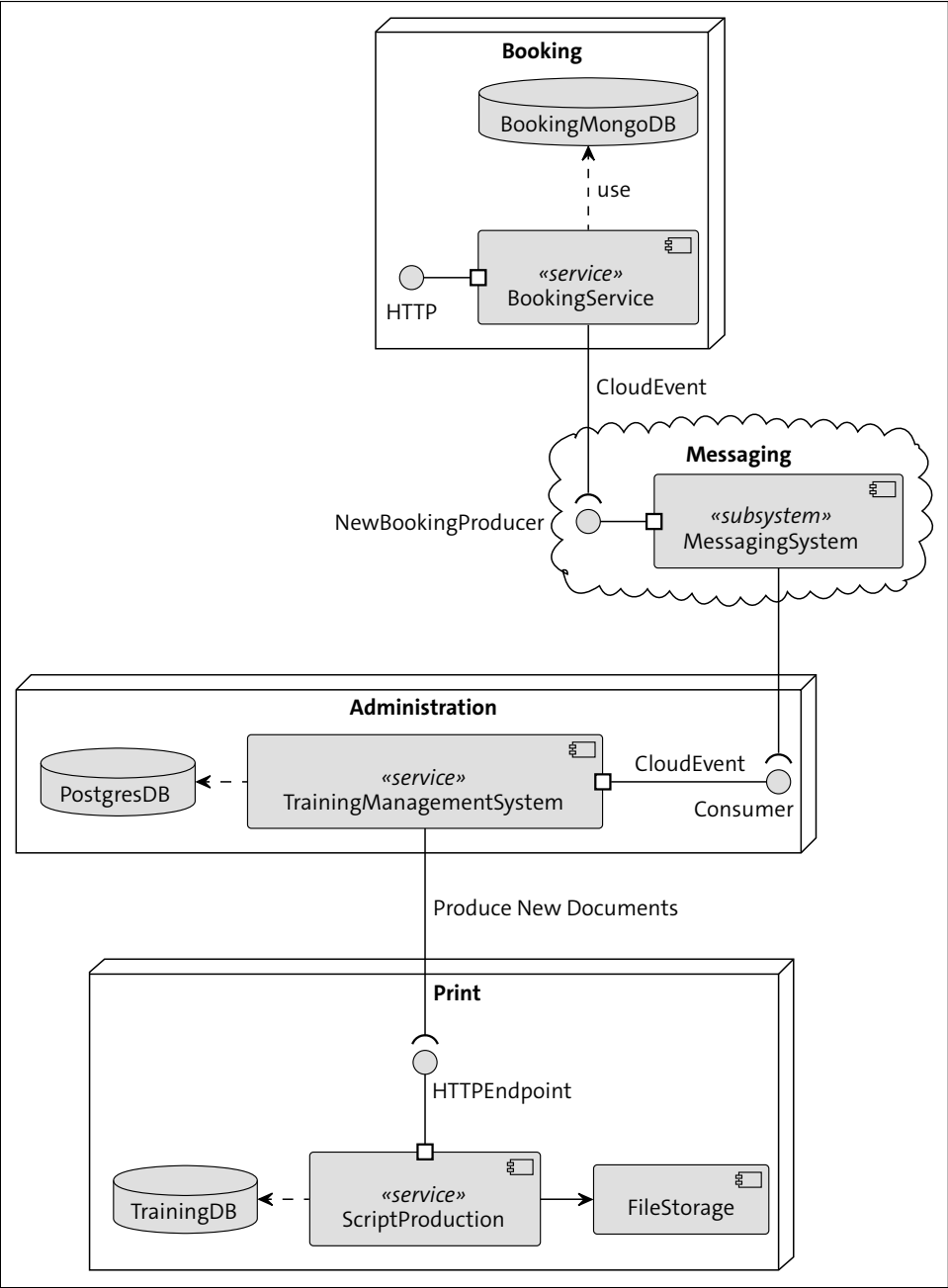


Figure 3.18 Component Diagram for the Training Application

3.4 C4 Model for Representing Software Architecture

The *C4 model* is another visualization concept designed by Simon Brown, an independent British software architect, as a lightweight approach to visualizing software architectures. In his book *Software Architecture for Developers*, published in 2012, Brown describes his experience as a software architect and introduces C4, a documentation option that has become quite popular with developers due to its pragmatic approach. It can also be used as an alternative to the arc42 template described earlier.

Simon Brown's approach critiqued the *box-and-line diagrams* that are so often used to visualize software. This representation of an architecture can easily lead to confusion because the notation and its meaning are not clearly expressed. In addition, different levels of detail are usually mixed in each diagram, and the information contained, such as the technologies or protocols used, often differs from diagram to diagram. Therefore, many diagrams cannot be understood without additional context.

In Brown's opinion, diagrams should be clear and understandable to outsiders without further explanation:

"Diagrams are the maps that help software developers navigate a complex code-base." —Simon Brown

The C4 model represents software systems using four hierarchical levels of abstraction and levels of detail associated with the levels. As the relevant key terms of the individual levels all contain the letter C, Brown speaks of the "C4" model:

- ▶ System context (level 1) provides an overview of how the software system fits into the rest of the system landscape.
- ▶ Container (level 2) is an enlarged and more detailed representation of the system in the form of coarse-grained, technical components (called *containers*) and their relationships with each other.
- ▶ Component (level 3) provides an enlarged view of a container with its internal components.
- ▶ Code (level 4) provides a further enlargement of a component and the representation of its implementation in the classic form of UML class diagrams, for example.

Several *views* can be created for each of these levels, each of which only shows a specific section of the level.

The division into predefined levels creates a standardized terminology for the individual building blocks of an architecture and describes corresponding levels of detail for a representation. Figure 3.19, Figure 3.20, Figure 3.21, Figure 3.22, and Figure 3.23 provide an overview of C4 representations and their levels of detail and scope. More precise details cannot yet be seen in this overview. I will describe them in the following sections.

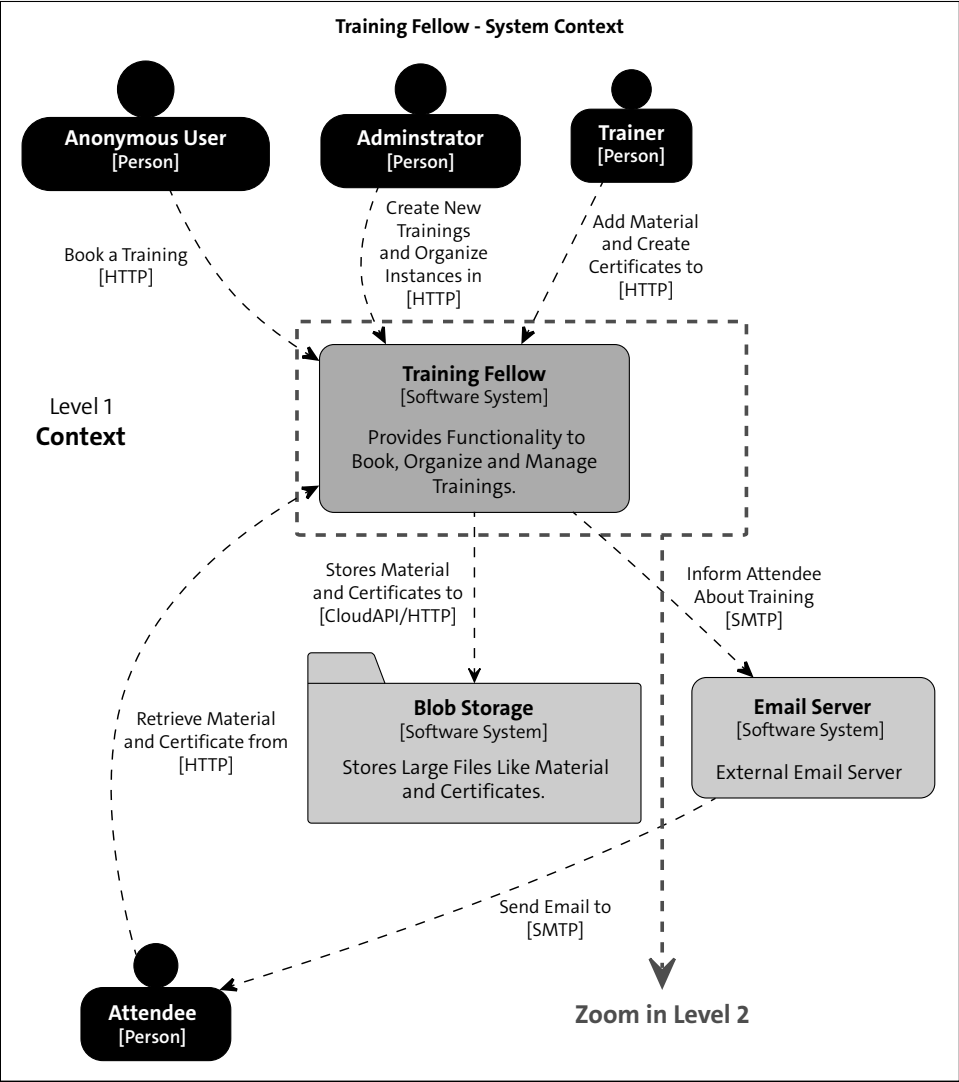


Figure 3.19 C4 Model: System Context (Level 1)

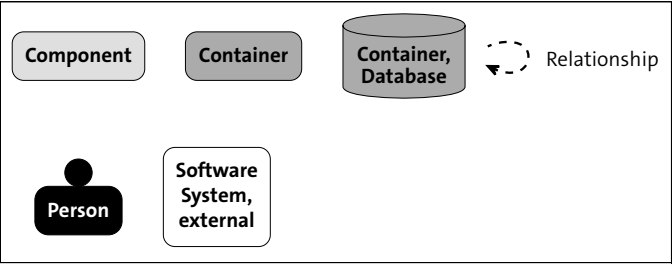


Figure 3.20 C4 Model: System Context (Level 1), Legend

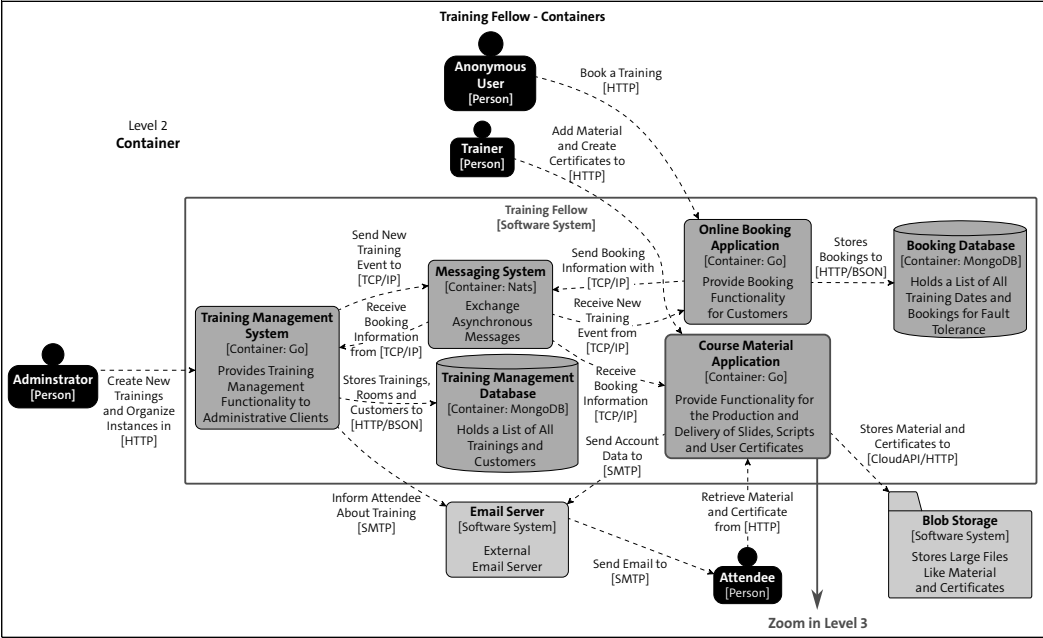


Figure 3.21 C4 Model: Container (Level 2)

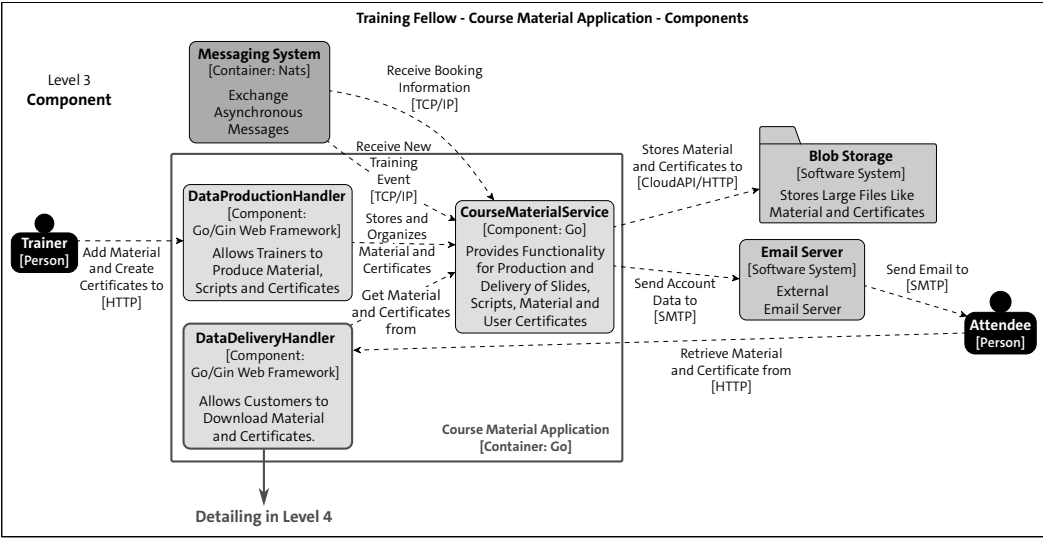


Figure 3.22 C4 Model: Component (Level 3)

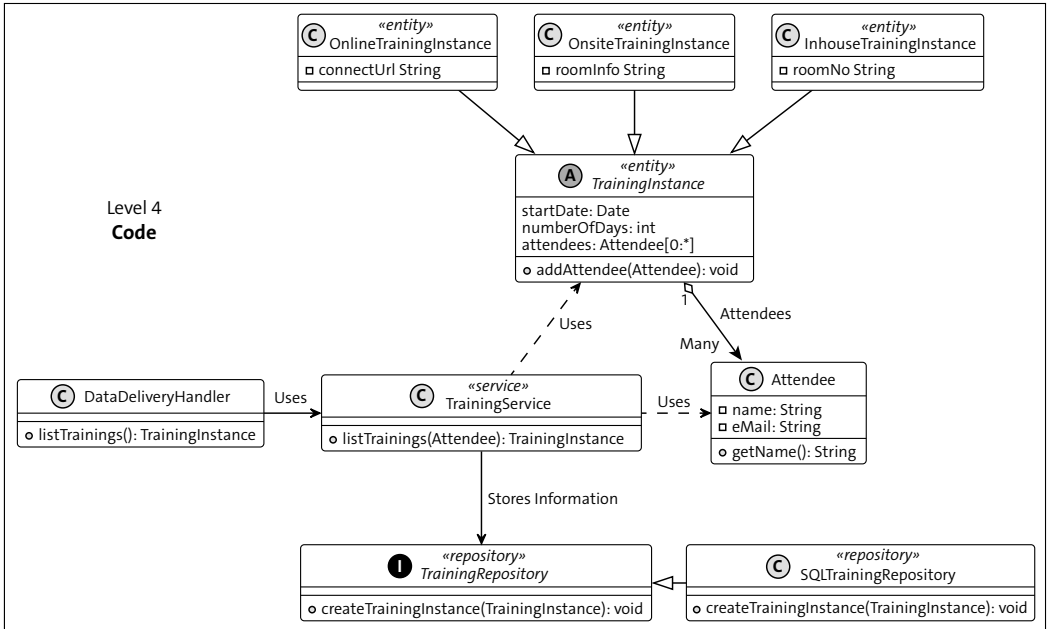


Figure 3.23 C4 Model: Code (Level 4)

With various tools, you have the option of displaying the models in such a way that you can switch to the corresponding refinement level during display, thus allowing the viewer to navigate interactively through the model.

C4 diagrams are often compared with maps, for instance, Google Maps. In these cases as well, various information is displayed at different levels of detail, and special views provide selected details. For example, a railway map does not contain any information about the highway network, which is not relevant in this context. You can zoom in for more information and zoom out again for a better overview.

In contrast to UML, since no fixed notation or design language is prescribed for C4 diagrams, you can define and use your own custom elements or designs.

However, as recommended by Brown, if you create your own elements, you should create a legend either directly in the diagram or in a separate explanation. Doing so allows you to clearly indicate the meaning of shapes, line type, color, borders, and abbreviations and thus increase comprehensibility. Figure 3.20 shows a legend for the diagram shown in Figure 3.19.

A notation of the C4 diagrams, as generated for the examples using *Structurizr*, is widely used. *Structurizr* is a modeling tool that enables you to generate different views from a common textually described model. I present the tool in more detail in Section 3.5.3.

The common forms of presentation are summarized in Figure 3.24. Each element has a name, a specified element type, and a description. Technical details can also be included, which can be displayed depending on the view's level of detail. Colors support

the presentation. External systems are highlighted in gray, for example. For the example, no distinction was made between synchronous and asynchronous calls, but such details could also be displayed by choosing different colors for the relationship arrows.

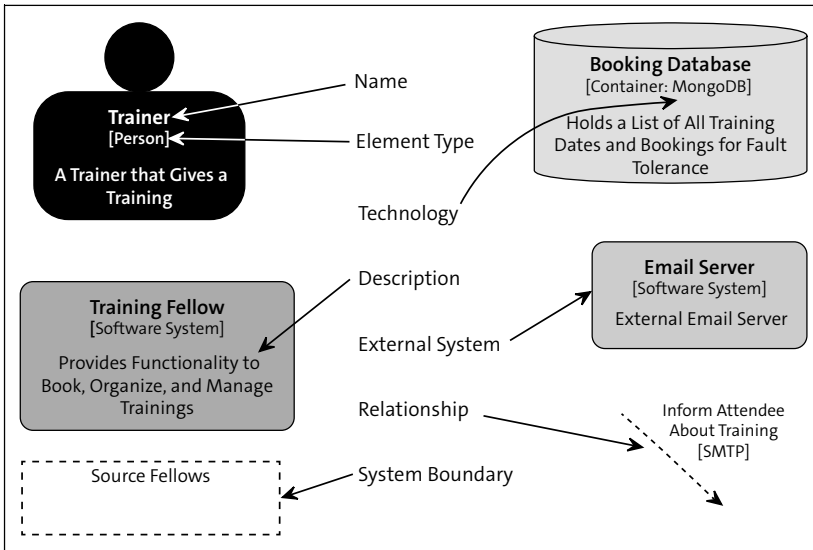


Figure 3.24 Legend for the Notation Used

3.4.1 System Context (Level 1 or C1)

The system context diagram is intended to give both technically and non-technically interested parties an overview of the system and answer the following questions:

- What is the software system that has been or will be created?
- Who uses the system?
- How does it fit into the system landscape?

At this level, only a few details are displayed, as shown in Figure 3.25 for the Training Fellow sample application. The goal is to provide a rough overview of the system and its dependencies, the *big picture*.

The system is represented as a central box around which the other participants are arranged. All dependencies to external systems and users are drawn in the diagram, and ideally, each relationship is provided with brief information on why and how communication with the external system or user takes place.

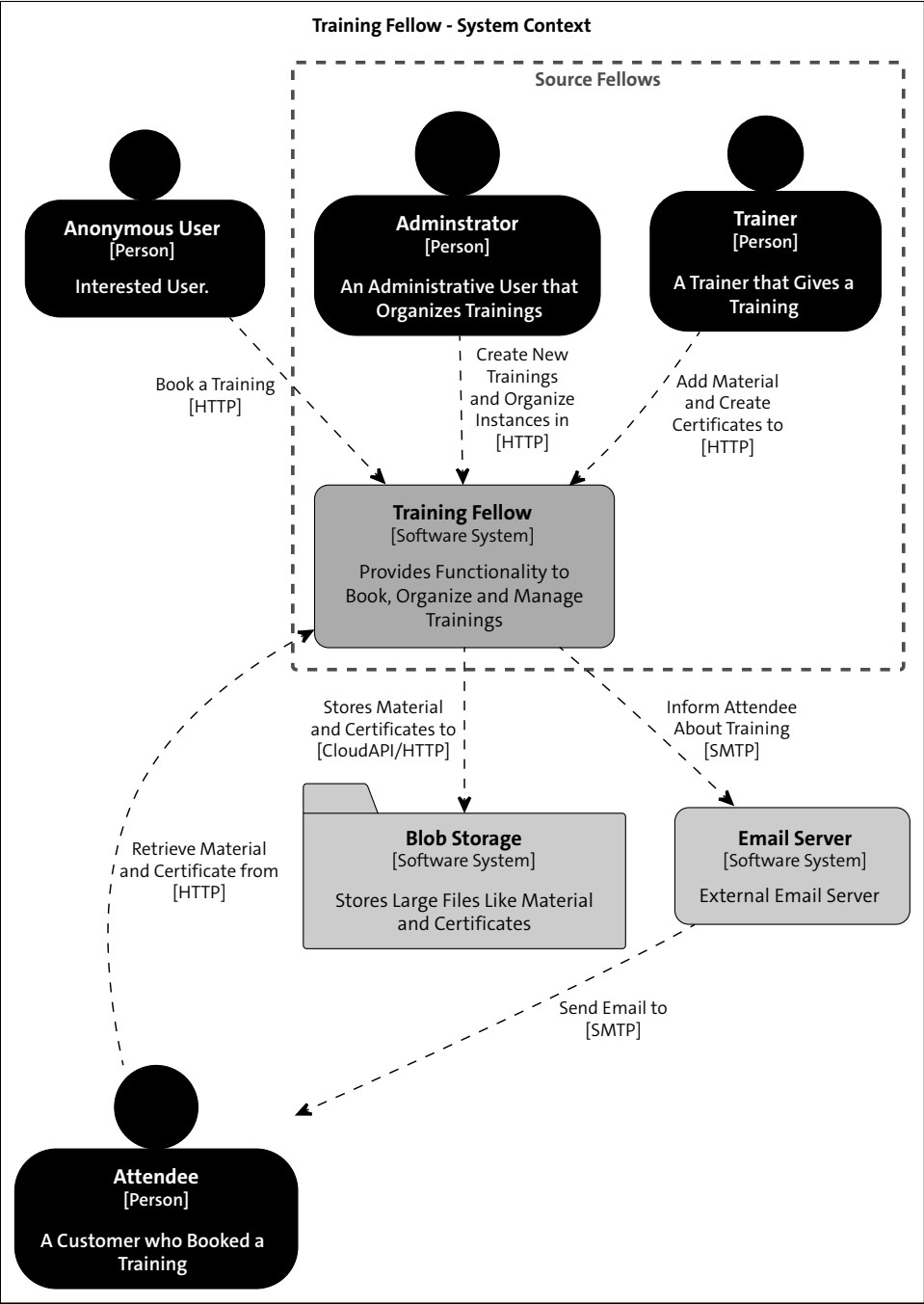


Figure 3.25 C4 System Context Example

3.4.2 Container (Level 2 or C2)

At the second level of abstraction, container diagrams indicate the rough structure of the software architecture and the responsibilities of the individual components it contains. The diagram already contains details of the technologies used, albeit at a fairly high level of abstraction.

The following questions should be answered by the container diagrams:

- ▶ How is the system's software architecture structured?
- ▶ Which technology decisions were made at a high level of abstraction?
- ▶ How are the responsibilities distributed across the individual components?
- ▶ How do the individual containers communicate with each other?
- ▶ In which container should functionality be implemented by the development team?

The example shown in Figure 3.26 refines the system context view of the Training Fellow sample application, showing that the application consists of three separate containers that communicate with each other asynchronously via a messaging system. In this case, the messages are exchanged between the container and the messaging system via a TCP/IP connection.

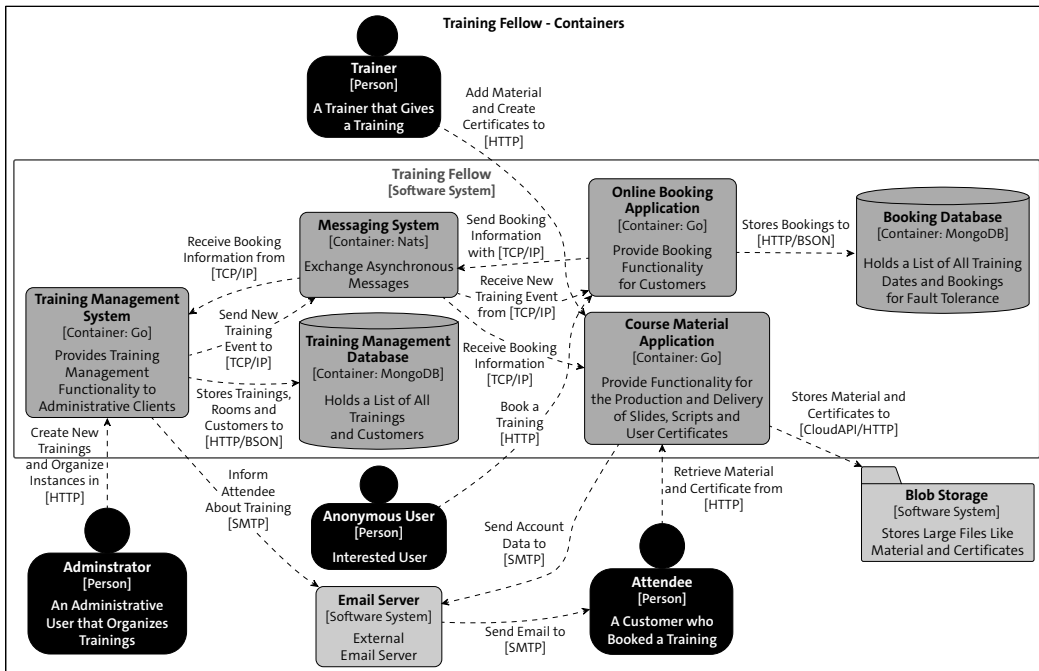


Figure 3.26 Example of a C4 Container Diagram

The responsibilities of each component are noted in each case so that extensions can be made at the appropriate points.

The diagram captures how the training management system (TMS) container is responsible for managing training courses, for example, and makes this functionality available to an administrative user via an HTTP connection. The container is implemented in Go and uses its own MongoDB database for its tasks, in which all training courses and customer data are stored. The diagram also shows that the TMS container sends status changes of training bookings to the participants via email and exchanges messages with the other containers via the messaging system. The TMS is informed about newly scheduled training dates and receives messages when new bookings have been made, which is indicated by the arrows pointing toward the container.

The level of detail of a container diagram is not explicitly specified and can be determined by the user. No technical details need to be provided, but certain decisions can be better understood by presenting more information.

3.4.3 Component (Level 3 or C3)

The third level of the C4 diagrams shows individual components with additional details on their technologies and internal structures. Figure 3.27 shows the Course Material application, which is responsible for creating and managing training materials and certificates.

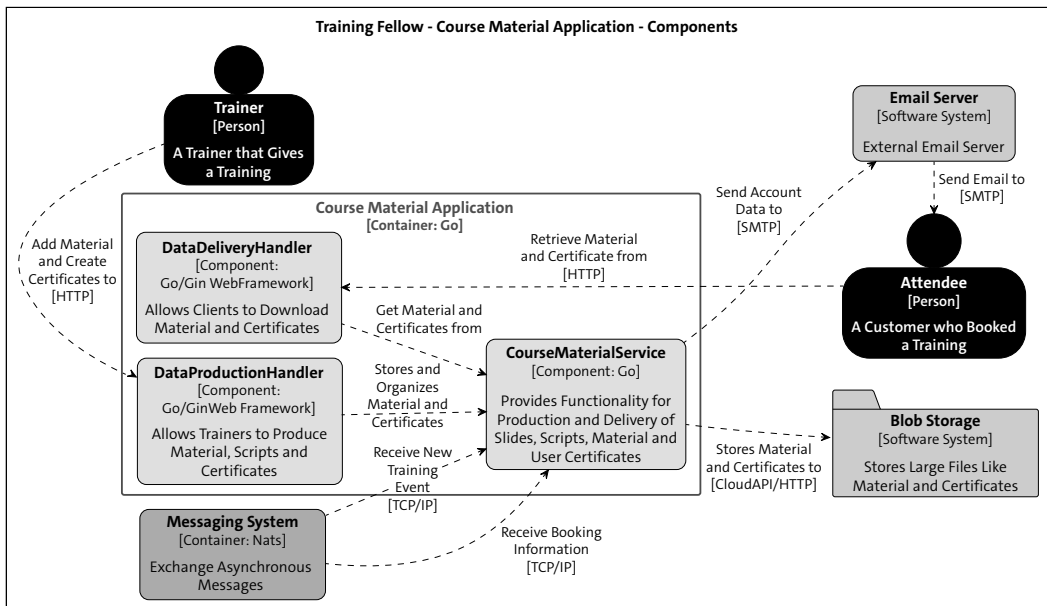


Figure 3.27 The Course Material Component in the C4 Model

The diagram always shows just one container and is intended to answer the following questions:

- What components does a container consist of?

- Are all components assigned to a container?
- Is it clear on an abstract level how the software works?

Three internal components are listed for the Course Material application in our example: two handler implementations, which are responsible for HTTP-based communication with users, and a service component, which contains the business logic. Each component has its own responsibilities as well as additional information on the internal technical details. The container diagram shows that our app is a Go-based application, and from the additional details, we can clearly see that the *Gin* web framework is used for its handler implementations.

No infrastructure components or *cross-cutting concerns*, such as those required for logging, are listed in this sample diagram because these components are not business-critical for the use case. If this information is important or critical, the required components can be included, or their use can be indicated by an additional comment. You can also display the affected components separately by color-coding and describing the coding in a legend.

The same rule applies to shared functionalities. If a functionality is business critical, the information should be included; if not, a reference is usually sufficient.

3.4.4 Code (Level 4 or C4)

The last and highest level of detail in the C4 diagrams are found in code-level diagrams. These diagrams should show how an individual component is set up or structured at the code level and how it works internally.

UML diagrams or excerpts from UML diagrams can be used for representations at this level. This approach is particularly suitable for class diagrams or sequence diagrams.

Incidentally, Simon Brown recommends not including this level in the model since the content can largely be created automatically from the source code using tools: The effort involved in manually creating the level is too high compared to its benefits.

Generating C4 Diagrams

No tools are required or necessary for creating C4 models. However, in Section 3.5.2 and Section 3.5.3, I present the *PlantUML* and *Structurizr* tools, which you can use to describe and visualize C4 models.

3.5 Doc-as-Code

In the past, software documentation or architecture documentation would often be created and managed in parallel with development work using separate tools, such as a standalone UML editor, Microsoft PowerPoint, Microsoft Word, or Wiki systems like

Atlassian Confluence. These tools cannot be integrated into the development process, or only integrated poorly, and developers are often forced to leave their familiar environment (i.e., the development environment) and switch to the documentation tools for their work. The necessary tools may not be available for their development platform and must then require virtual machines or remote connections.

Tools that are difficult or even impossible to integrate into the development process have disadvantages: New or adapted features, for example, must be consistently documented or subsequently documented separately. Even if the development team has the necessary discipline, a great deal of effort is required to keep the documentation up to date and complete.

In addition, some tools prevent team members from collaborating in a meaningful way because their file formats cannot be edited by several people at the same time or because manual workflows are required in the release process. If, for example, UML diagrams are integrated as PNG images, they cannot be worked on collaboratively.

Doc-as-code, on the other hand, takes the approach of viewing documentation as source code and creating, editing, and managing it accordingly. The individual documents are created in lightweight text formats and versioned and managed in the source code repository used. Different versions can be compared with the usual development tools and restored to a specific state if necessary.

Changes to the documents, like source code changes, can be coordinated with team members and integrated via pull requests or review processes, for example.

Storing documents within a source code repository provides advantages, not only during creation and editing: Various output formats or scopes can be generated from the predominantly plain text formats by integrating them into existing build processes. A special output document with the appropriate content can be provided for each target group from the same text source.

You can also aggregate information from multiple such as text files and source code. Automatically created UML diagrams can be integrated directly into the documentation and don't require separate updating or redundant storage.

Instead, with each release of an application, up-to-date documentation is automatically created and delivered. The documentation itself becomes an artifact to be delivered and is integrated into the agile development process known as *continuous documentation*.

In the following sections, I will introduce you to formats and tools that you can use to pursue a doc-as-code approach.

3.5.1 AsciiDoc

AsciiDoc is a text format for creating plain text documents for structured documentation. A human-readable, platform-independent markup language, *AsciiDoc* is similar to *Markdown*, but with many additional functions and a clearly defined syntax.

AsciiDoc was developed to bridge the gap between plain text and complex markup languages such as HTML or LaTeX. Various output formats can be generated from the documents, such as HTML, PDF, Word, or EPUB.

AsciiDoc and Asciidoctor

AsciiDoc is a simple text markup language for creating technical content. You can find the language specification at <https://asciidoc.org/>. Plugins are available for common development environments that provide support for editing and allow documents to be previewed.

The transformation of AsciiDoc documents into the various source formats is performed by *Asciidoctor* (<https://asciidoctor.org/>), which can be easily integrated into a separate build pipeline with its many extensions.

The AsciiDoc format is quite similar to other markup languages. The text file shown in Listing 3.21 can be generated as a PDF using Asciidoctor, for example, as shown in Figure 3.28.

```
= Hello, AsciiDoc!
```

The syntax description can be found at <https://asciidoc.org>[AsciiDoc].

```
== Section Title
```

```
* A list item  
* Another list item
```

```
[source,go]  
----  
fmt.Println("Hello, World!")  
----
```

Listing 3.21 AsciiDoc Example

The advantages of AsciiDoc when creating technical documents, such as architecture documentation, include the tooling, which can be easily integrated into your own build pipeline, and the standardized syntax. This syntax contains all known markups (such as tables, lists, and more), but also provides for the insertion of external content as well.

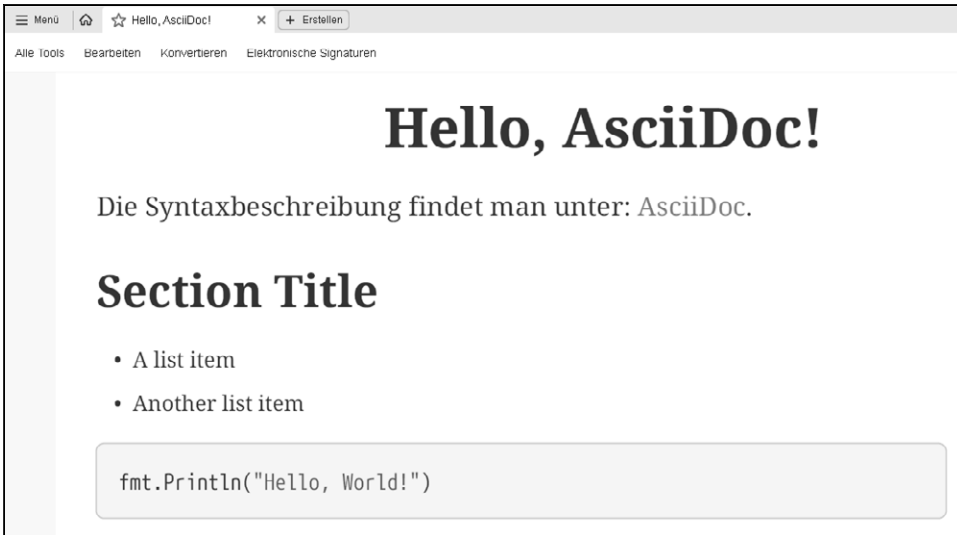


Figure 3.28 PDF Document Generated from the AsciiDoc File

The core of the tooling is the *Asciidoctor* text processor that converts AsciiDoc into various formats such as HTML and PDF.

Some of the advantages of AsciiDoc include the following:

- ▶ A defined syntax, not multiple grammars
- ▶ Generation of various output formats possible
- ▶ Extensive tooling for integration into the build pipeline
- ▶ Easy inclusion of content from external files (images, graphics, source code, AsciiDoc documents, and more)
- ▶ Possible use of variables in documents
- ▶ Creation of target group-oriented documentation through the customized aggregation of content

3.5.2 PlantUML

With the help of the *PlantUML*, an open-source project, you can create various graphical representations from simple text descriptions. Several UML diagrams are supported, as well as other diagram types such as *wireframes* or *mind maps*.

Strictly speaking, PlantUML is a drawing tool, as the models or drawings created are not validated or checked for consistency.

PlantUML Diagram Types

The following diagrams and representations are supported by PlantUML:

- ▶ Sequence diagrams
- ▶ Class diagrams
- ▶ Activity diagrams
- ▶ Deployment diagrams
- ▶ Timing diagrams
- ▶ Displaying YAML (YAML Ain't Markup Language) data
- ▶ Salt/wireframe
- ▶ Gantt charts
- ▶ Work breakdown structures (WBSs)
- ▶ Entity relationship diagrams
- ▶ Use case diagrams
- ▶ Object diagrams
- ▶ Component diagrams
- ▶ State diagrams
- ▶ Displaying JavaScript Object Notation (JSON) data
- ▶ Network diagrams
- ▶ ArchiMate diagrams
- ▶ Mind maps
- ▶ Math

The diagrams can either be created in a text editor or in a development environment or generated from source code using various tools.

As soon as the text descriptions are ready, they can be transformed into a graphical representation using the available command-line tool, for example, in PNG or SVG files. PlantUML takes over the entire layout of the diagrams unless you override its defaults with your own configuration.

Plugins with syntax highlighting and a direct preview of the diagrams are available for most development environments.

The syntax is quite similar for the different diagram types. Elements are defined directly or implicitly, and they are connected to each other via relationships.

The example shown in Listing 3.22 is a *sequence diagram* in which the Alice and Bob elements are defined implicitly and communicate using calls described as arrows (-->). Figure 3.29 shows the generated diagram.

```

@startuml
Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response
Alice -> Bob: Another authentication Request
Alice <-- Bob: Another authentication Response
@enduml

```

Listing 3.22 PlantUML Example of a Sequence Diagram

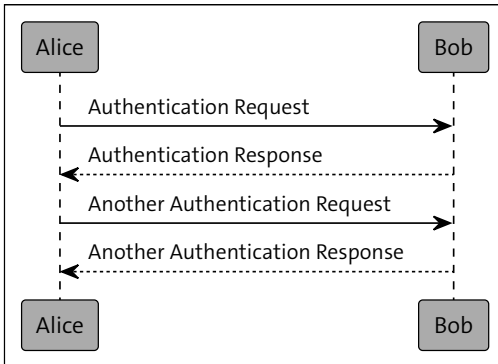


Figure 3.29 Generated PlantUML Sequence Diagram

An explicit definition of elements is useful for class diagrams, for example, as soon as the class has attributes and methods.

The example shown in Listing 3.23 is the explicit definition of class A with the `counter` attribute and the abstract `start` method. Class B is implicitly defined and extends class A. Figure 3.30 shows the generated result.

```

@startuml
class A {
{static} int counter
+void {abstract} start(int timeout)
}
note right of A::counter
This member is annotated
end note
note right of A::start
This method is now explained in a UML note
end note

A <|-- B

@enduml

```

Listing 3.23 Definition of a Class in PlantUML: Explicit and Implicit

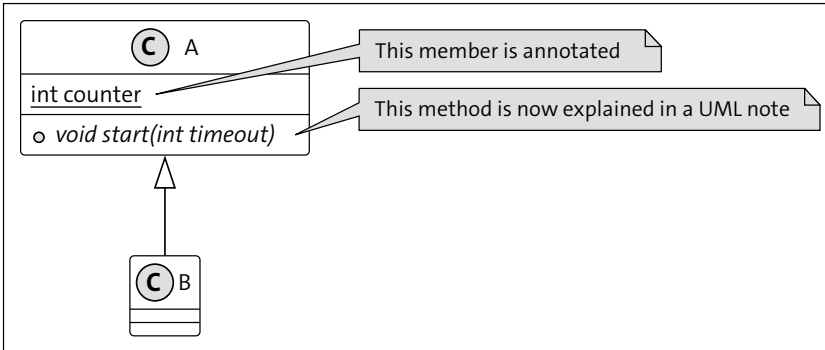


Figure 3.30 Generated PlantUML Class Diagram

PlantUML diagrams can be integrated easily into AsciiDoc documentation and, under certain circumstances, even generated directly from the source code during a build run using plugins. In this way, the diagrams always show the current status of the application, and the *doc-as-code* approach is expanded to become also a *diagrams-as-code* approach.

Mermaid as an Alternative to PlantUML

As an alternative to *PlantUML*, the open-source tool *Mermaid* is becoming increasingly popular. Mermaid is a JavaScript-based diagram and chart tool that also processes text files and displays them graphically. Although the syntax of both tools is similar, some users consider Mermaid more user-friendly and easier to learn, offering an even more intuitive syntax and additional graphical editors.

You can find this tool at <https://mermaid.js.org/>.

3.5.3 Structurizr

Structurizr is a *DSL (Domain Specific Language)* for describing complete C4-based software architecture models as text files and managing them as code. Although you can also use other tools, such as PlantUML, Structurizr is a particularly simple method of modeling.

For each architecture model created, you can generate several views or diagrams with different levels of detail, as shown in Figure 3.31.

The starting point for the model is what's called a *workspace*, in which the *component*, *container*, or *softwareSystem* components known from C4 are defined in the area of a *model*. Various views and their content are then defined below the *views* element.

The example shown in Listing 3.24 is an architecture model that contains the *User* user and the software system named *Software System*. The software system consists of the *Web Application* and *Database* containers. The relationships between the components

are described via arrows (->) and are included in the model. In our example, User calls the Web Application container.

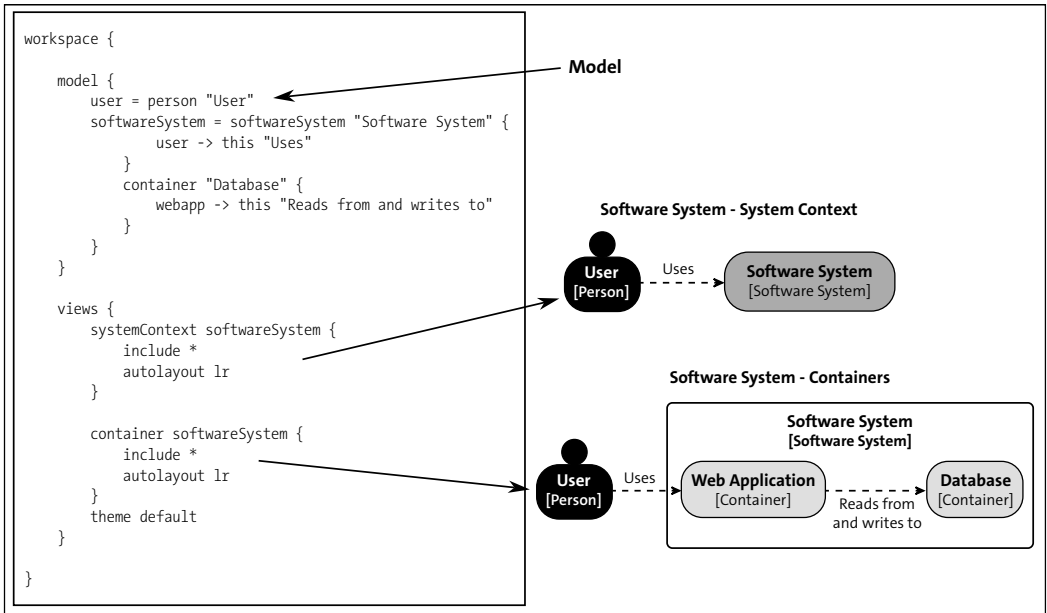


Figure 3.31 Structurizr Model and Views

Two views are defined in the `views` area, in each of which all components are to be displayed.

The diagrams are generated either via the *Structurizr* command-line tool or via a server that is either self-managed or used as a cloud service.

Structurizr diagrams can also be easily integrated into AsciiDoc documents and generated from code using tools.

```
workspace {
  model {
    user = person "User"
    softwareSystem = softwareSystem "Software System" {
      webapp = container "Web Application" {
        user -> this "Uses"
      }
      container "Database" {
        webapp -> this "Reads from and writes to"
      }
    }
  }
  views {
    systemContext softwareSystem {
```

```
        include *
    }
    container softwareSystem {
        include *
    }
}
}
```

Listing 3.24 Example of a Structurizr Architecture Model



Kristian Köhler

Software Architecture and Design

The Practical Guide to Design Patterns

- Understand the fundamentals of good software design
- Master architecture, design, application organization, documentation, and more
- Learn to use essential design patterns by following detailed code examples



rheinwerk-computing.com/6144

We hope you have enjoyed this reading sample. You may recommend or pass it on to others, but only in its entirety, including all pages. This reading sample and all its parts are protected by copyright law. All usage and exploitation rights are reserved by the author and the publisher.

Kristian Köhler is a software architect and developer with a passion for solving problems using efficient, well-structured software. He is the managing director of Source Fellows GmbH.

ISBN 978-1-4932-2743-3 • 488 pages • 09/2025

E-book: \$54.99 • Print book: \$59.95 • Bundle: \$69.99



Rheinwerk
Publishing