6
Software Development Principles and Practices

One of the main goals for software architects is to design high-quality software applications. There are a number of software design principles and best practices that can be applied to achieve that goal.

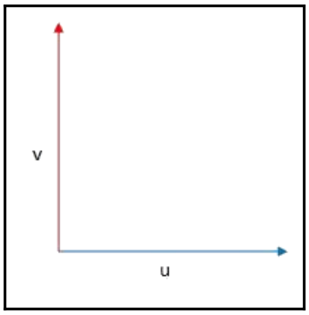
Software architects can apply these principles and practices when designing software architectures and encourage developers to use them in their implementations. These principles and practices are used to improve quality, simplify maintenance, increase reusability, find defects, and make software systems easier to test.

In this chapter, we will cover the following topics:

- Designing orthogonal software systems, including a focus on loose coupling and high cohesion
- Minimizing complexity in a software system by following principles such as KISS, DRY, information hiding, YAGNI, and Separation of Concerns (SoC)
- The SOLID design principles, which include the Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and the Dependency Inversion Principle (DIP)
- Using **Dependency Injection** (**DI**) to provide dependencies to a class
- Using unit testing to improve the quality of a software system
- Ensuring that development environments can be set up easily
- Practice of pair programming
- Reviewing deliverables, such as code reviews, formal inspections, and walkthroughs

Designing orthogonal software systems

In geometry, two Euclidean vectors are orthogonal if they are perpendicular (form a right angle of 90 degrees). The two vectors meet at the origin point, but do not intersect. The two vectors are independent of each other:



Software that is well designed is orthogonal in that its modules are independent of each other. Ideally, changes to one module in a software system should not require changes to another module. Software systems will undergo many changes during their lifetime and designing them with this in mind provides a number of benefits, including increased productivity for those who work on them and lowered risk of introducing defects when changes are made. Designing orthogonal systems may have higher upfront costs, but over time, a highly maintainable and extendable system will be worth it.

Orthogonal systems are designed so that their elements are loosely coupled and highly cohesive. Let's look at the concepts of coupling and cohesion in more detail.

Loose coupling

Coupling is the degree to which a software module depends on another software module. Coupling between modules is a measure of how closely connected they are, and it can either be loose, sometimes described as low or weak, or it can be tight, sometimes referred to as high or strong. The degree of coupling between modules reflects the quality of their design.

Software modules that are tightly coupled are more complex, which decreases their maintainability. Tight coupling makes modifying the code more difficult because a change in a tightly coupled module will likely require changes in other modules. This introduces a higher degree of risk as there is a greater likelihood that a new defect could be introduced if a software module is modified.

It is also easy to engage in parallel development if the code is loosely coupled. One developer can work on one part of the application independent of another developer who is working on a different part of the application.

Modules that are loosely coupled do not have as many dependencies with other modules. Changes to tightly coupled modules will take more time and effort due to the interdependencies with other modules. As the number of modules that are affected by a change increases, it will take longer for developers to make the modifications, and for testers to test the changes. Tight coupling also reduces reusability. It is harder to reuse a module when dependent modules must be included in the reuse.

Types of coupling

There are different types of coupling. The following are details regarding those types, in order of the tightest (least desirable) to loosest (most desirable) coupling. It should be noted that it is possible for two modules to be coupled in more than one way. In those situations, the coupling type is determined by the worst, or tightest, coupling type.

Content coupling

Content coupling is the highest type of coupling. It is considered so bad that it is also referred to as pathological coupling. It occurs when one module directly references the internal or private information in another module. For example, it exists when one module accesses or changes private data in another module.

Modules should never be designed to have this type of coupling. If modules have content coupling, they should be refactored so that there is a proper level of abstraction. The modules should not directly rely on the internal workings of each other.

Common coupling

Common coupling, also known as global coupling, is a high level of coupling. This type of coupling is highly undesirable. Although sometimes it is unavoidable, modules should be designed to minimize the existence of this type of coupling.

Modules exhibit common coupling when they share the same global data, such as a global variable. It is perfectly acceptable to share configuration data throughout an application. However, as a general rule, if you are going to use other types of global data, it is better to use something that has a fixed value, such as a constant, rather than a variable whose value can vary at runtime.

External coupling

External coupling is another type of high coupling. It exists when multiple modules share the same part of an environment that is external to the software. This could come in the form of having to use an external data format, interface, communication format, tool, or device.

Sometimes, external dependencies are imposed and unavoidable, but we should still seek to limit the number of modules that have those dependencies. Doing so will ensure that if the external dependency changes, only a limited number of modules are affected.

Control coupling

Control coupling is a moderate type of coupling. Two modules exhibit control coupling when one module controls the internal logic of the other by passing it information. An example of this is when a module passes a control flag to another module, which uses it to control its flow.

This type of coupling may be acceptable, but an effort should be made to make it known that the coupling exists so that the modules can be tested together. It is beneficial to detect any problems with either of the modules earlier rather than later.

Stamp coupling (data-structured coupling)

Stamp coupling is a fairly low type of coupling. It is also known as data-structure coupling because it occurs when modules share a composite data structure. By composite data structure, we mean that it is data that has some internal structure to it, such as a record.

When a composite data structure is shared between two modules, some of the fields in the data structure may not even be used. For example, a module passes a composite data structure to another module, which then just uses one field in it.

It is similar to data coupling, except that the data shared is a composite data type rather than primitive data values and that not all of the values shared may be used.

Data coupling

Data coupling occurs when two modules share some data which are just primitive data values. It is another low type of coupling. A common type of data coupling is when a module calls a method on another module, and inputs and outputs are shared in the form of method parameters and the return value.

When two modules need to interact, this is a common and acceptable type of coupling. Unlike stamp collecting, where some of the values in the shared composite data structure may not be used, all of the parameters in data coupling are used. If any parameters are not needed, they should be removed.

Message coupling

Modules exhibit message coupling when one module calls a method on another and does not send any parameters. The only coupling is on the name of the method, but nothing else. It is the lowest type of coupling.

No coupling

There are, of course, situations where there is no coupling between modules. This is when two modules have no direct communication at all. It is an ideal that allows the two modules to be implemented, tested, and maintained independently.

The Law of Demeter (LoD) / principle of least knowledge

The **Law of Demeter** (**LoD**), or **principle of least knowledge**, is a design principle related to loose coupling. In order to minimize coupling between software modules, the principle can be followed when designing software.

The principle follows the *only talk to your friends* idiom, which keeps coupling loose by limiting a module's communication with other modules. Ideally, a method should only call other methods in the same object, in objects that were passed into it, in direct component objects, in objects that it created/instantiated, or in objects in a global variable that are accessible.

Another one of the tenets of LoD is that a software module should know as little as possible about other modules. This will ensure its independence from other modules, allowing coupling to remain loose. See the *Information hiding* section for a principle that helps to achieve this goal.

Designing for loose coupling

During designs and implementations, your goal as a software architect should be to minimize the amount of coupling that exists between modules. Modules should be designed to be as independent as possible.

Coupling can be reduced by eliminating or reducing the number of unnecessary dependencies. For any coupling that must exist, it should be the lowest type that is necessary. Loose coupling reduces complexity and increases maintainability and reusability.

Coupling typically affects the level of cohesion, so that loose coupling correlates with high cohesion and tight coupling correlates with low cohesion.

High cohesion

Cohesion is the degree to which the elements inside a module belong together. It is the strength of the relationships of elements within a module, and how united they are in their purpose. Cohesion is a qualitative measure of the consistency of purpose within a module.

There are different types of cohesion, and those that reflect a higher level of cohesion are preferable. Highly cohesive modules have a single, well-defined purpose, and reflect a better quality of design.

Software modules with low cohesion are harder to maintain. If a module contains multiple unrelated functions, changes to it are more likely to require changes in other modules. This will require extra time and effort, not just in development, but also testing. The extra complexity in modules with low cohesion make it more likely that defects may be introduced when they are modified. They may also be harder to understand, making them more difficult to modify.

Reusability is lessened for modules with low cohesion. Modules with low cohesion, performing many disparate functions, are less likely to be reused for other purposes. A module that works together as a logical unit with a clear purpose is more likely to be reused.

Types of cohesion

The level of cohesion in a module is represented by the type of cohesion. Let's examine the different types of cohesion, from lowest (least desirable) to highest (most desirable).

Coincidental cohesion

Coincidental cohesion occurs when elements in a module are grouped arbitrarily. There is no relationship among the different elements, making it the lowest (worst) type of cohesion. Sometimes, you will see this type of cohesion in a *utilities* or *helpers* class where a number of unrelated functions have been placed together.

Coincidental cohesion should be avoided and, if it is encountered in a module, the module should be refactored. Each part of the module should be moved to an existing or new module where it would make logical sense for it to exist.

Logical cohesion

Modules exhibit logical cohesion when elements are grouped together because they are related in some way logically. Even though the functionality of logically cohesive modules might be of the same general category, they may be different in other ways. For this reason, this type of cohesion is considered low. While better than coincidental cohesion, these types of modules are not very cohesive.

An example of logical cohesion would be a module that contains a set of functions that handles I/O for the application. While they are related logically, the nature of the various functions would be quite different. They would be more cohesive if each type of I/O was handled by a separate module.

Temporal cohesion

Temporal cohesion exists when the elements of a module are grouped together based on when they are processed. This can occur when different elements are grouped together simply because they need to be executed at a single moment in time. This is another type of low cohesion.

An example of temporal cohesion is grouping a bunch of elements together because they are all related to system startup, system shutdown, or the handling of a system error. Even though the elements are related temporally, they are only weakly related to each other. This makes the module harder to maintain and reuse.

The elements should be grouped into different modules, with each module designed for a single purpose.

Procedural cohesion

A module exhibits procedural cohesion when its elements have been grouped together because they always execute in a particular sequence. For example, payment processing for a customer placing an order might involve the following steps being executed in a particular sequence:

- Gathering payment information
- Validating payment method details
- Checking whether funds are available or whether there is enough available credit

- Persisting the order in a database
- Checking inventory levels
- Creating a back order or canceling an order based on inventory
- Sending the order for fulfillment
- Sending an email confirmation to the customer

Although the various parts are all related by the order of execution, some of the individual activities are quite distinct from each other.

This type of cohesion is considered moderate. Although it is an acceptable level of cohesion, it is not ideal. If possible, refactoring can be performed to improve the level of cohesion.

Communicational cohesion

Communicational cohesion occurs when parts of a module are grouped together because they use the same set of inputs and outputs. If a module has different elements that have been grouped together because they access and modify the same data structure, it would demonstrate communicational cohesion.

For example, a data structure that represents the contents of a customer's shopping basket might be used by a variety of elements in a single module. The elements might calculate discounts, shipping, and taxes based on the same data structure.

This level of cohesion is moderate and usually considered acceptable.

Sequential cohesion

Sequential cohesion exists when the different parts of a module are grouped together because the output of one part serves as the input for another part. Modules of this type have a moderate level of cohesion.

An example of a module that is sequentially cohesive would be one that is responsible for formatting and validating a file. The output of an activity that formats a raw record becomes the input for an activity that then validates the fields in that record.

Functional cohesion

Functional cohesion occurs when elements of a module are grouped together because they are united for a single, well-defined purpose. All of the elements in the module work together to fulfill that purpose. Functional cohesion in a module is ideal and is the highest type of cohesion.

Functional cohesion promotes the reusability of a module and makes it easier to maintain. Examples of functionally cohesive modules include one that is responsible for reading a particular file and one that is responsible for calculating shipping costs for an order.

Designing for high cohesion

Software architects should design modules to have high cohesion. Each module should have a single, well-defined purpose. The elements contained in the module should be related and contribute to that purpose.

If there are auxiliary elements contained in a module that are not directly related to the main purpose, consider moving them to either a new module or an existing module that has the same purpose of the element being moved.

Cohesion and coupling are related in that high cohesion correlates with loose coupling and low cohesion correlates with tight coupling.

Minimizing complexity

Building software is inherently complex and a number of problems result from complexity. Higher levels of complexity in software:

- Cause delays in schedules
- Lead to cost overruns
- May cause the software to behave in unintended ways or lead to an unanticipated application state
- May create security loopholes or prevent security issues from being discovered in a timely fashion
- Are a predictive measure of lower levels of some quality attributes, such as lower maintainability, extendibility, and reusability

In *The Mythical Man-Month*, Fred Brooks divides the problems facing software engineering into two categories, *essential* and *accidental*:

> *"All software construction involves essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity, and accidental tasks, the representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints."*

Accidental difficulties are problems that are just inherent to the production of software in general. They are problems that software engineers can fix and may not even be directly related to the problem they are trying to solve. Improvements in programming languages, frameworks, design patterns, **integrated development environments** (**IDEs**), and software development methodologies are just some examples of progress over the years in eliminating or reducing accidental difficulties.

Essential difficulties are the core problems that you are trying to solve and they can't simply be removed to reduce complexity. Software development teams spend more time on essential complexities than accidental ones.

We try to manage and minimize the complexity, whether it is accidental or essential. As it has probably become apparent by now, a recurring theme in this book is the importance of managing and minimizing complexity. It has a direct relationship with the quality of the software and is therefore a major focus for software architects.

Minimizing complexity in software helps to eliminate or manage both accidental and essential difficulties. Some of the principles related to minimizing complexity include KISS, DRY, information hiding, YAGNI, and SoC.

KISS principle – "Keep It Simple, Stupid"

The **KISS principle**, which is an acronym for *Keep It Simple, Stupid*, has been used in numerous contexts to convey the idea that systems generally work best if they are kept simple. The principle is applicable to the design of software systems. A development team should strive to not overcomplicate their solutions.

Variations of the acronym include *Keep It Short, Simple*, *Keep It Simple, Stupid*, *Keep It Simple, Straightforward*, and *Keep It Simple, Silly*. All of them have the same basic meaning, which is to express the value of simplicity in designs.

Origin of KISS

The creation of the principle is typically credited to the late Kelly Johnson, who was an aeronautical and systems engineer. Among other accomplishments, he contributed to aircraft designs for the Lockheed Corporation (now known as Lockheed Martin after its merger with Martin Marietta).

Though the principle is commonly communicated as *Keep It Simple, Stupid*, Kelly's original version didn't have a comma in it. The word *Stupid* wasn't intended to refer to a person.

Kelly introduced the principle by explaining to the engineers that the jet aircraft they were designing needed to be something that a man in the field could fix with basic training and common tools. The design needed to satisfy this requirement, which is understandable given that the aircraft might need to be repaired quickly in a combat situation.

Applying KISS to software

Simplicity is a highly desirable quality in software systems, and this includes their designs as well as their implementations. Making software more complicated than it needs to be lowers its overall quality. Greater complexity reduces maintainability, hinders reusability, and may lead to an increase in the number of defects.

Some ways to follow the KISS principle in software include:

- Eliminating duplication as much as possible (see the *DRY – "Don't Repeat Yourself"* section)
- Eliminating unnecessary features (see the *YAGNI – "You Aren't Gonna Need It"* section)
- Hiding complexity and design decisions (see the *Information hiding* section)
- Following known standards when possible and minimizing deviations and surprises

Even after a module is implemented, if you see a method or class that could be made simpler, consider refactoring it if you have the opportunity.

Don't make it overly simple

In a quest for simplicity, we cannot oversimplify a design or implementation, though. If we reach a point that it negatively affects the ability to deliver on required functionality or quality attributes, we have gone too far.

Keep the following quote, attributed to Albert Einstein, in mind when designing software: *Everything should be made as simple as possible, but not simpler.*

DRY – "Don't Repeat Yourself"

The **DRY principle** stands for *Don't Repeat Yourself* and strives to reduce duplication in a codebase. Duplication is wasteful and makes a codebase unnecessarily larger and more complex. This makes maintenance more difficult. When code that has been duplicated needs to be changed, modifications are required in multiple locations. If the changes applied everywhere are not consistent, defects may be introduced. Software architects and developers should avoid duplication whenever possible.

When a design violates the DRY principle, it is sometimes referred to as a WET (Write Everything Twice) solution (or *Waste Everyone's Time* or *We Enjoy Typing*).

Copy-and-paste programming

In poorly written codebases, code duplication often results from **copy-and-paste programming**. This happens when a developer needs the exact same or very similar logic, which exists somewhere else in the system, so they duplicate (copy and paste) the code. This violates the DRY principle and lowers the quality of the code.

Copy-and-paste programming can sometimes be acceptable and serve a useful purpose. Code snippets, which are small blocks of reusable code, may speed up development. Many IDEs and text editors provide snippet management to make the use of snippets easier for developers. However, beyond the appropriate application of snippets, it is usually not a good idea to copy and paste your application code in multiple places.

Magic strings

Magic strings are strings that appear directly in your code. Sometimes, these strings are needed in multiple places and are duplicated, violating the DRY principle. Maintenance of these strings can become a nightmare because if you want to change the value of the string, you have to change it in multiple places. The problem is exacerbated when the string is used, not just in multiple places within the same class, but within multiple classes.

There are many examples of magic strings, from exception messages, settings in configuration files, parts of a file path, or a web URL. Let's look at an example where the magic string value represents a cache key. This serves as a good example because this is a case where a magic string might be duplicated multiple times within the same class and even within the same method:

```
public string GetFilePath()
{
    string result = _cache.Get("FilePathCacheKey");
```

```
        if (string.IsNullOrEmpty(result))
        {
            _cache.Put("FilePathCacheKey", DetermineFilePath());
            result = _cache.Get("FilePathCacheKey");
        }

        return result;
    }
```

This key is repeated multiple times, increasing the possibility of a typo resulting in a defect. In addition, if we ever want to change the cache key, we will have to update it in multiple places.

To follow the DRY principle, let's refactor this code so that the cache key is not repeated. First, let's declare a constant at the class level for the magic string:

```
    private const string FilePathCacheKey = "FilePathCacheKey";
```

Now, we can use that constant in our `GetFilePath` method:

```
    public string GetFilePath()
    {
        string result = _cache.Get(FilePathCacheKey);

        if (string.IsNullOrEmpty(result))
        {
            _cache.Put(FilePathCacheKey, DetermineFilePath());
            result = _cache.Get(FilePathCacheKey);
        }

        return result;
    }
```

Now, the string is declared in just one location. If you are going to place a magic string in a constant, you should think about where the constant should be declared. One consideration is the scope of its use. It may be appropriate to declare it within the scope of a particular class, but in some cases, a broader or narrower scope will make more sense.

Although placing a magic string in a constant is a good technique for a variety of situations, it is not always ideal. This decision also depends on the type of string and its purpose. For example, if the string is a validation message, you might want to place it in a resource file. If there are any internationalization requirements, placing translatable strings, such as validation messages, in a resource file will facilitate translating the messages into different languages.

How to avoid duplication

DRYness can be achieved by being mindful and taking action when appropriate. If you find yourself copying and pasting code, or simply writing code that is identical or similar to existing code, think about what you are trying to accomplish and how it can be made reusable.

Duplication in logic can be eliminated by abstraction. This concept is referred to as the **abstraction principle** (or the **principle of abstraction**). The principle is consistent with the DRY principle and is a way to reduce duplication. The code that is needed in multiple places should be abstracted out, and the locations that need it can then be routed through the abstraction. Some refactoring may be necessary to make it generic enough to be reused, but it is worth the effort. Once the logic is centralized, if it needs to be modified in the future, perhaps to fix a defect or to enhance it in some way, you will be able to make the changes in a single location.

As we saw in the case of magic strings, duplication with values can be eliminated by placing the value in a central location, such as the declaration of a constant.

If there is duplication in a process, it may be possible to reduce it through automation. Manual unit testing, builds, and integration processes can be eliminated by automating those processes. The automation of tests and builds will be discussed further in `Chapter 13`, *DevOps and Software Architecture*.

Don't make things overly DRY

When attempting to follow the DRY principle, be careful not to consolidate disparate items that just happen to be duplicates in some way. If two or more things are duplicates, it may be that they are just *coincidentally repetitive*.

For example, if two constants have the same value, that does not mean they should be combined into one constant for the sake of eliminating duplication. If the constants represent distinct concepts, they should remain separate.

Information hiding

Information hiding is a principle that advocates for software modules to be designed such that they hide implementation details from the rest of the software system. The idea of information hiding was introduced by D.L. Parnas in *On the Criteria to Be Used in Decomposing Systems into Modules*, which was published in 1972.

Information hiding decouples the internal workings of a module from the places in the system that call it. The details of a module that do not need to be revealed should be made inaccessible. Information hiding defines constraints related to what properties and behaviors can be accessed. Callers interact with the module's public interface and are protected from the implementation details.

There are a number of reasons to abide by the principle of information hiding.

Reasons for information hiding

Information hiding is useful at all levels of design. Only exposing the details that need to be known reduces complexity, which improves maintainability. Unless you are specifically interested in the internal details, you do not need to concern yourself with them.

Another one of the key reasons for information hiding is to hide design decisions from the rest of the software system. This is particularly beneficial if the design decision might change. By hiding a design decision, if the decision needs to be changed, it minimizes the amount and extent of the modifications that will be necessary. It provides the flexibility to make changes later if it is necessary to do so.

Whether the design decision is to use a particular API, represent data in a certain way, or use a particular algorithm, the modifications necessary to change that design decision should be kept as localized as possible.

What needs to be exposed/hidden?

You and your team should really think about the properties and behaviors (methods) that need to be exposed for a module. Everything else can be hidden. Through the use of a public interface, we can define what we want to make available.

Information hiding assists with defining public interfaces. Rather than lazily exposing most of a class, it forces us to consider what really needs to be made public. The public interface defines a contract that the implementation must follow, and allows others to know *what* is available. It is up to the implementation to decide *how* it is accomplished.

YAGNI – "You Aren't Gonna Need It"

YAGNI, which stands for *You Aren't Gonna Need It*, or *You Ain't Gonna Need It*, is a principle from the software development methodology of **Extreme Programming** (**XP**). XP is one of the first agile methods and was the dominant one until the rise of the popularity of Scrum. YAGNI is similar to the KISS principle in that they both aim for simpler solutions, with YAGNI focusing on a specific aspect, which is the removal of unnecessary functionality and logic.

Avoid over-engineering a solution

The idea behind YAGNI is that you should only implement functionality when you need it and not just because you think you may need it some day. Ron Jeffries, one of the co-founders of XP, once said:

"Always implement things when you actually need them, never when you just foresee that you need them."

Following the YAGNI principle helps you to avoid over-engineering a solution. You don't want to spend time on future scenarios that are unknown. The problem with implementing a feature that you think might eventually be needed is that quite often the feature ends up not being needed or the requirements for it change.

Code that is not written equates to time and money that is saved. Spending time and money on a feature you don't need takes away from time and money you could have spent on something that you do need. Resources are finite, and using them on something that is unnecessary is a waste. As was the case with code duplication, adding unnecessary logic to an application increases its size and complexity, which reduces maintainability.

Situations where YAGNI doesn't apply

YAGNI applies to presumptive features, as in functionality that is not currently needed. It does not apply to code that would make the software system easier to maintain and modify later. In fact, following YAGNI means you may be changing the system later to add a feature, so the system should be well designed for this purpose. If a software system is not maintainable, making changes later may be difficult.

You may come across times where, in hindsight, a change made sooner would have prevented more expensive changes later. This may be particularly true for software architects if the change is architecture related. Design decisions made for architecture are among the earliest decisions made, and having to change them later can be costly.

It can sometimes be difficult to foresee which changes should have been made before they were needed. However, for the most part, following YAGNI is beneficial. Even in the case of an architecture change, a good architecture design reduces complexity and makes it easier to make changes. It also makes it more likely that when a change is needed, it can be limited in scope and may not even require architectural changes.

As software architects gain more experience, they become more adept at spotting exceptions to the YAGNI principle where a particular change should be made before it is needed.

Separation of Concerns (SoC)

Concerns are the different aspects of functionality that the software system provides. **Separation of Concerns** (**SoC**) is a design principle that manages complexity by partitioning the software system so that each partition is responsible for a separate concern, minimizing the overlap of concerns as much as possible.

Following the principle involves decomposing a larger problem into smaller, more manageable concerns. SoC reduces complexity in a software system, which reduces the effort needed to make changes and improves the overall quality of the software.

When the DRY principle is followed, and logic is not repeated, a SoC is usually a natural result as long as the logic is organized properly.

SoC is a principle that can be applied to multiple levels in a software application. At the architecture level, software applications can follow a SoC by separating different logic such as user-interface functionality, business logic, and infrastructure logic. An example of an architecture pattern that separates concerns at this level is the **Model-View-Controller** (**MVC**) pattern, which we will cover in the next chapter.

We can apply SoC at a lower level, such as with classes. If we were providing order processing functionality in a software system, the concern of validating credit card information shouldn't exist in the same place as the concern for updating inventory. They are distinct concerns that should not be placed together. At this level, it is related to the Single Responsibility Principle, which we will discuss shortly.

An example of separating concerns by language in web programming is **Hypertext Markup Language** (**HTML**), **Cascading Style Sheets** (**CSS**), and **JavaScript**. They complement each other with one being focused on the content of web pages, one for the presentation, and one for the behavior.

Following SOLID design principles

SOLID design principles focus on creating code that is more understandable, maintainable, reusable, testable, and flexible. SOLID is an acronym that represents five separate software design principles:

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Software architects should be familiar with SOLID principles and apply them in their designs and implementations. They should realize, though, that the principles are guidelines, and while you should strive to follow them, you may not always be able to accomplish that fully. Use your judgement as to when, and to what degree, these principles should be followed.

Now, let's explore the five design principles that make up SOLID in more detail.

Single Responsibility Principle (SRP)

The **Single Responsibility Principle** (**SRP**) states that each class should have only one responsibility, meaning it should do one thing and do that thing well. A responsibility is a reason to change, so each class should have only one reason to change. If we group together the functions that need to change for the same reason, and separate out the things that change for other reasons, we can create a class that follows this principle.

If a class has multiple responsibilities, there is a likelihood that it is used in a greater number of places. When one responsibility is changed, not only do we run a higher risk of introducing defects into other responsibilities in the same class, but there is a greater number of other classes that might be impacted.

By following the single responsibility principle, if we need to change a particular responsibility, that change would be located in a single class. This is the way to create an orthogonal software system.

Applying this principle does not necessarily mean, as some posit, that each class should only have a single public method. Although it does reduce the size of classes, the goal is to have each class have a single responsibility. Fulfilling a single responsibility may require multiple public methods.

This principle is related to the SoC principle because, as concerns are separated from each other, it facilitates the creation of classes that have a single responsibility. Following the DRY principle also helps us to abide by the SRP. By removing duplicate code and placing it in a single location so that it can be reused, the classes that need the logic do not have to repeat it and therefore do not need to be responsible for it.

Let's take a look at an example of the SRP. It is written in C#, although you will get the idea even if you do not use that particular programming language. We have an email service that is responsible for sending out emails. There is a requirement to log information to a log file, so it also contains logic to open, write to, and close a log file on a file system:

```
public class EmailService : IEmailService
{
    public SendEmailResponse SendEmail(SendEmailRequest request)
    {
        if (request == null)
            throw new ArgumentNullException(nameof(request));

        SendEmailResponse response = null;

        try
        {
            // Logic to send email

            // Log info about sent email
            LogInfo("Some info message");
        }
        catch (Exception ex)
        {
            // Log details about error
            LogError("Some error message");
        }

        return response;
    }

    private void LogInfo(string message)
    {
        // Logic to write to file system for logging
    }

    private void LogError(string message)
    {
        // Logic to write to file system for logging
    }
}
```

As you can see in this simple example, this class has more than one responsibility: sending out emails as well as handling the logging. This means that it has more than one reason to change. If we wanted to change how emails were sent out, or allow for the logging to target cloud file storage instead of a file on a local file system, both would require changes to the same class.

This violates the SRP. Let's refactor this class so that it is only responsible for one thing:

```
public class EmailService : IEmailService
{
    private readonly ILogger _logger;

    public EmailService(ILogger logger)
    {
        if (_logger == null)
            throw new ArgumentNullException(nameof(logger));

        _logger = logger;
    }

    public SendEmailResponse SendEmail(SendEmailRequest request)
    {
        if (request == null)
            throw new ArgumentNullException(nameof(request));

        SendEmailResponse response = null;

        try
        {
            // Logic to send email

            // Log info about sent email
            _logger.LogInfo("Info message");
        }
        catch (Exception ex)
        {
            // Log details about error
            _logger.LogError($"Error message: {ex.Message}");
        }

        return response;
    }
}
```

Now, the `EmailService` class is only responsible for sending out emails. The logic for logging has been abstracted out to an interface. This dependency is injected in through the class's constructor, and the implementation will be responsible for how logging works.

This class is now only responsible for a single thing and therefore only has one reason to change. Only changes related to the sending of emails will require modifications to this class. It no longer violates the SRP.

Open/Closed Principle (OCP)

The **Open/Closed Principle** (**OCP**) states that software components, such as classes, should be open for extension but closed for modification. When requirements change, the design should minimize the amount of changes that need to occur on existing code. We should be able to extend a component by adding new code without having to modify existing code that already works.

When Dr. Bertrand Meyer first came up with the principle in his book *Object Oriented Software Construction*, it focused on using implementation inheritance as the solution. If new functionality is needed, a new subtype is created and the base class and any existing subtypes could remain unchanged.

Software engineer Robert C. Martin, popularly known as Uncle Bob, redefined the principle in his article *The Open-Closed Principle*, and later in his book *Agile Software Development, Principles, Patterns, and Practices*, by stressing the importance of abstraction and the use of interfaces. Using interfaces, we can change implementations as needed. In this way, we can change behavior without having to modify existing code that relies on the interfaces.

Let's take a look at an example. In this program, we have a `Shape` class with `Rectangle` and `Circle` classes that inherit from it. A `Canvas` class has methods that allow us to draw the shapes:

```
public class Canvas
{
    public void DrawShape(Shape shape)
    {
        if (shape is Rectangle)
            DrawRectangle((Rectangle)shape);

        if (shape is Circle)
            DrawCircle((Circle)shape);
    }

    public void DrawRectangle(Rectangle r)
    {
        // Logic to draw a rectangle
    }

    public void DrawCircle(Circle c)
```

```
        {
            // Logic to draw a circle
        }
    }
```

If a new request comes in that requires us to be able to draw a new shape, such as a triangle, we will have to modify the `Canvas` class. This class has not been designed to be closed for modification and violates the OCP.

Developers will need to understand the `Canvas` class in order to add a new shape. Modifications to the `Canvas` class will require unit tests to be revisited and introduces the possibility that existing functionality could be broken.

Let's refactor this poor design so that it no longer violates the OCP:

```
public interface IShape
{
    void Draw();
}
public class Rectangle : IShape
{
    public void Draw()
    {
        // Logic to draw rectangle
    }
}

public class Circle : IShape
{
    public void Draw()
    {
        // Logic to draw circle
    }
}

public class Canvas
{
    public void DrawShape(IShape shape)
    {
        shape.Draw();
    }
}
```

The `Canvas` class is now much smaller, with each shape now having its own implementation of how to draw itself. If there was a requirement to add a new shape, we can create a new `Shape` class that implements the `IShape` interface without having to make any changes to the `Canvas` class or any of the other shapes. It is now open for extension but closed for modification.

Liskov Substitution Principle (LSP)

Inheritance is one of the four pillars of **object-oriented programming** (**OOP**). It allows subclasses to inherit from a base class (sometimes referred to as the parent class), which includes the properties and methods of the base class. When you first learned about inheritance, you may have been taught about "is a" relationships. For example, "Car is a Vehicle," if Car was a base class and Vehicle was a subtype of that base class.

The **Liskov Substitution Principle** (**LSP**) is an object-oriented principle that states that subtypes must be substitutable for their base types without having to alter the base type. If a subtype is inherited from a base class, we should be able to substitute the subclass for that base class without any issues. Subtypes extending a base class should do so without changing the behavior of the base class. When the LSP is violated, it makes for confusing code that is hard to understand.

For a given base class or an interface the base class implements, the subtypes of that base class should be usable through the base class or an interface the base class implements. The methods and properties of the base class should make sense and work as intended for all of the subtypes. If the classes work without an issue and behave as expected, the subtypes are substitutable for the base class. When the LSP is violated, this is not the case. Although the code may compile, unexpected behavior or runtime errors may be experienced.

A classic example to illustrate the LSP is that of a `Rectangle` class and a `Square` class. In geometry, a square is a type of rectangle, so every square is a rectangle. The only difference is that with a square all of the sides have the same length.

We can model this by creating a `Rectangle` class, which is a base class, and a `Square` class, which is a subtype that inherits from it:

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    private int _width;
    private int _height;

    public override int Width
    {
        get { return _width; }
        set
        {
            _width = value;
            _height = value;
        }
    }

    public override int Height {
        get { return _height; }
        set
        {
            _width = value;
            _height = value;
        }
    }
}
```

As you can see from this code sample, the `Square` class overrides how the width and height are set to ensure that they remain equal:

```
    Rectangle rect = new Rectangle
    {
        Width = 5,
        Height = 4
    };
    Console.WriteLine(rect.CalculateArea());

    Square sqr = new Square
    {
        Width = 4
    };
    Console.WriteLine(sqr.CalculateArea());

    Rectangle sqrSubstitutedForRect = new Square
    {
        Width = 3,
        Height = 2
    };
    Console.WriteLine(sqrSubstitutedForRect.CalculateArea());

    Console.ReadLine();
```

The area of the `rect` object will calculate to 20 and the area of the `sqr` object will calculate to 16, as expected. However, the area of the `sqrSubstitutedForRect` object will calculate to 4 and not 6. As these classes are currently designed, the `Square` subtype is really not substitutable for the `Rectangle` base class.

This code will compile but it violates the SRP and leads to confusing results. This is a simplistic example, but you start to get the idea. With complex class hierarchies, violations of the SRP can lead to defects, some of which can be difficult to solve.

Interface Segregation Principle (ISP)

Interfaces define methods and properties, but do not provide any implementation. Classes that implement an interface provide the implementation. Interfaces define a contract, and clients can use them without concerning themselves with their implementation details. The implementation can change and as long as a breaking change is not made to the interface, the client does not need to change their logic.

The **Interface Segregation Principle** (**ISP**) states that clients should not be forced to depend on properties and methods that they do not use. When designing software, we prefer smaller, more cohesive interfaces. If an interface is too large, we can logically split it up into multiple interfaces so that clients can focus on only the properties and methods that are of interest to them.

When interfaces are too large and attempt to cover too many aspects of functionality, they are known as *fat* interfaces. The ISP is violated when classes are dependent on an interface with methods they do not need. Violation of the ISP increases coupling and makes maintenance more difficult.

Let's look at an example where we are creating a system for a business that sells books. We create the following interface and class for the products:

```
public interface IProduct
{
    int ProductId { get; set; }
    string Title { get; set; }
    int AuthorId { get; set; }
    decimal Price { get; set; }
}

public class Book : IProduct
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public decimal Price { get; set; }
}
```

Now, let's say that the business owners want to start selling physical disks of movies. The properties needed are very similar to `IProduct`, so an inexperienced developer might use the `IProduct` interface for their `Movie` class:

```
public class Movie : IProduct
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId {
        get => throw new NotSupportedException();
        set => throw new NotSupportedException();
    }
    public decimal Price { get; set; }
    public int RunningTime { get; set; }
}
```

`AuthorId` doesn't make sense for movies, but in this example, the developer decides to just mark the property as not supported. This is one of the *code smells* for violation of the ISP. If a developer of a class that is implementing an interface finds themselves having to mark properties or methods as not supported/not implemented, perhaps the interface needs to be segregated.

The `Movie` class also needs to represent the running time of the movie, which isn't a property needed by the `Book` class. If it is added to the `IProduct` interface, all classes that implement that interface will need to be modified. This is another code smell indicating that there may be an issue with the design of the interface.

If we were to refactor this code so that it no longer violates the ISP, we would separate the `IProduct` interface into more than one so that the single "fat" interface is separated into multiple smaller and more cohesive ones:

```
public interface IProduct
{
    int ProductId { get; set; }
    string Title { get; set; }
    decimal Price { get; set; }
}

public interface IBook : IProduct
{
    int AuthorId { get; set; }
}

public interface IMovie : IProduct
{
    int RunningTime { get; set; }
}

public class Book : IBook
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public decimal Price { get; set; }
}

public class Movie : IMovie
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public decimal Price { get; set; }
    public int RunningTime { get; set; }
}
```

We can see that the `IProduct` interface only contains the properties needed by all products. The `IBook` interface inherits from `IProduct`, so that any class implementing `IBook` will need to implement the `AuthorId` property in addition to everything in `IProduct`. Similarly, `IMovie` inherits from `IProduct` and contains the `RunningTime` property that is only needed for movie implementations.

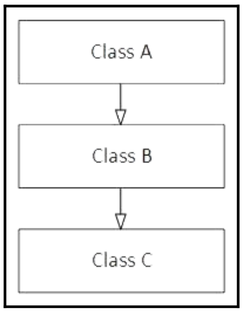
We are no longer in violation of the ISP, and classes implementing these interfaces do not have to deal with properties and methods that are not of interest to them.

Dependency Inversion Principle (DIP)

The **Dependency Inversion Principle** (**DIP**) is a principle that describes how to handle dependencies and write loosely coupled software. In their book *Agile Principles, Patterns, and Practices in C#*, Robert C. Martin and Micah Martin state the principle as follows:

> *"The high-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *Abstractions should not depend upon details. Details should depend upon abstractions."*

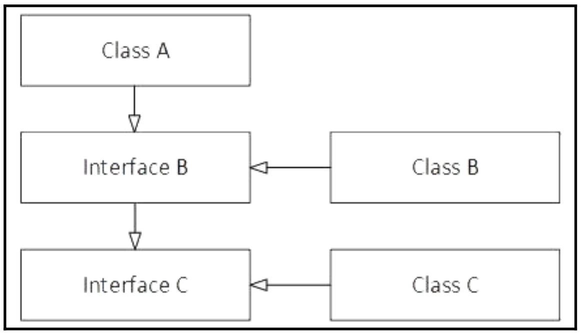
For example, let's say that **Class A** depends on **Class B** and **Class B** depends on **Class C**:



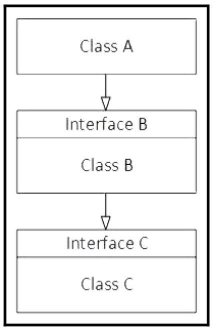
In a direct dependency graph, at compile time, **Class A** references **Class B** which references **Class C**. At runtime, the control flow will go from **Class A** to **Class B** to **Class C**. **Class A** and **Class B** will have to instantiate, or *new up*, their dependencies. This creates tightly coupled code that is difficult to maintain and test. Changes to one of the dependencies may require changes to the classes that use those dependencies.

Another disadvantage of this approach is that the code is not unit-testable because of its dependencies. We will not be able to create mock objects for dependencies because we are referencing concrete types rather than abstractions. There is no way to *inject* the mock objects so that we can create true unit tests that are not dependent on other classes.

Rather than high-level classes being dependent on lower-level classes, they should depend on abstractions through an interface. The interfaces do not depend on their implementations. Instead, the implementations depend on the interfaces:



With an inverted dependency graph, at compile time, **Class A** depends on an abstraction (**Interface B**), which in turn depends on an abstraction (**Interface C**). **Class B** and **Class C** implement **Interface B** and **Interface C**, respectively:



At run time, the flow of control goes through the interfaces, and each interface has an implementation.

The DIP is closely related to the inversion of control principle, with the inversion of control being applied to dependencies.

Inversion of Control (IoC)

Inversion of Control (**IoC**) is a design principle in which a software system receives the flow of control from reusable code, such as a framework. In traditional procedural programming, a software system would call into a reusable library. The IOC principle inverts this flow of control by allowing the reusable code to call into the software system.

Developers are now so familiar with using a variety of frameworks, and even multiple frameworks on a single project, that the principle of IoC is no longer a novel one. Although the principle of IoC can be applied to many more things than just dependencies, it has become closely related to dependencies for its IOC over them.

This is the reason why DI containers were originally, and sometimes still are, referred to as IoC containers. DI containers are frameworks that provide DI functionality, and we will discuss them shortly.

Dependency Injection (DI)

Dependency Injection (**DI**) is a technique that provides dependencies to a class, thereby achieving dependency inversion. Dependencies are passed (injected) to a client that needs it. There are a number of benefits to using DI in a software application.

Benefits of DI

DI removes hardcoded dependencies and allows them to be changed, at either runtime or compile-time. If the implementation of a dependency is determined at runtime rather than compile-time, this is known as late binding, or runtime binding. As long as we are programming to an interface, the implementation can be swapped out.

DI allows us to write loosely coupled code, making applications easier to maintain, extend, and test. As we know, when we need to make a change, loosely coupled code allows us to make a change in one part of our application without it affecting other areas of our application.

Testability increases in software applications that use DI. Loosely coupled code can be tested more easily. Code is written to depend on abstractions and not concrete implementations, so dependencies can be mocked with unit testing frameworks. Following the LSP, a class is not dependent on a concrete type; it is only dependent on the interface. As a result, we can inject mock objects for the dependencies by using the interface and writing unit tests.

Parallel development is made easier with DI. Developers can work on different pieces of functionality in parallel. Since the implementations are independent of each other, as long as the shared interfaces are agreed upon, development can occur at the same time. This is particularly beneficial on larger projects with multiple teams. Each team can work independently and share interfaces for the functionality that must be integrated.

DI patterns

The following are some of the patterns available that can be used for DI:

- Constructor injection
- Property injection
- Method injection
- Service Locator

The following section provides details on these various patterns.

Constructor injection

Constructor injection is a technique in which dependencies are passed through a class's constructor. This is an excellent way to inject dependencies because the dependencies are made explicit. The object cannot be instantiated without its dependencies.

If a class can be instantiated and its methods called, but the functionality does not work properly because one or more dependencies have not been provided, classes are being dishonest with their clients. It is a better practice to explicitly require dependencies.

The following example shows the constructor injection pattern:

```
public class Employee : Person
{
    private readonly ILogger _logger;
    private readonly ICache _cache;

    // Dependencies are injected via the constructor,
    // including the base class
    public Employee(ILogger logger, ICache cache,
        IOrgService orgService)
        : base(logger, orgService)
    {
        if (logger == null)
            throw new ArgumentNullException(nameof(logger));
        if (cache == null)
            throw new ArgumentNullException(nameof(cache));

        _logger = logger;
        _cache = cache;
    }
}

public class Person
{
    private readonly ILogger _logger;
```

```
        private readonly IOrgService _orgService;

        // Dependencies are injected via the constructor
        public Person(ILogger logger, IOrgService orgService)
        {
            if (logger == null)
                throw new ArgumentNullException(nameof(logger));
            if (orgService == null)
                throw new ArgumentNullException(nameof(orgService));

            _logger = logger;
            _orgService = orgService;
        }
    }
```

In this example, you can see that the `Employee` class has three dependencies: instances of `ILogger`, `ICache`, and `IOrgService`. The `Person` class, which is the base class of `Employee`, has two dependencies: instances of `ILogger` and `IOrgService`.

Constructor injection is used to provide instances of all of the dependencies for both classes. In this example, notice that the `Employee` class passes the dependencies that the `Person` class needs to it. Both the `Employee` and the `Person` classes have a dependency to `ILogger` and only the `Employee` class has a dependency to `ICache`. The `Employee` class doesn't even use the `orgService` instance directly, which is why it is not assigned to a class-level variable, but since its base class needs it, it is a dependency that is injected in and then passed on to its base class (`Person`).

The dependencies that are injected in are assigned to `readonly` variables. In the C# language, the `readonly` keyword indicates that the field can only be assigned as part of its declaration or in the constructor. We will not be assigning an instance of the dependency anywhere else other than the constructor with constructor injection, so it can be marked as `readonly`.

There is a guard clause that ensures that the required dependencies are not null. If they are, an exception is thrown. If a valid instance is passed in, it is assigned to a private variable so that the instance can be used later in the logic of the class.

Property injection

Property injection allows clients to supply a dependency through a public property. If you need to provide callers with the ability to provide an instance of a dependency, such as to override the default behavior, you can use this injection pattern.

The first time the getter is called, if the dependency has not already been supplied, a default instance should be provided through lazy initialization:

```
public class Person
{
    private IOrgService _orgService;
    public IOrgService OrgService
    {
        get
        {
            if (_orgService == null)
            {
                // Lazy initialization of default
                _orgService = new OrgService();
            }
            return _orgService;
        }
        set
        {
            if (value == null)
            {
                throw new ArgumentNullException(nameof(value));
            }
            // Only allow dependency to be set once
            if (_orgService != null)
            {
                throw new InvalidOperationException();
            }
            _orgService = value;
        }
    }
}
```

If you want a dependency to only be supplied once through a setter, a check can be performed in the setter, as is done in the example.

For the lazy initialization, you will need a way to get the default instance. It is not ideal to instantiate, or *new up*, the dependency in the class. However, the default value could be provided by another means, such as constructor injection. The property would give you the flexibility to provide a different instance at runtime.

The property injection pattern provides dependencies in an implicit way. If a default instance is not provided in the getter when the dependency has not been previously set and the logic needs the dependency to work properly, an error or unexpected results could occur. If you are going to use this pattern over an explicit one, such as constructor injection, a default instance should be provided in some way.

Unlike the example for constructor injection, notice that the private class-level variable to hold the dependency (`_orgService`) does not have the `readonly` keyword. In order to use property injection, we need the ability to set the variable outside of the variable's declaration and the constructor.

Method injection

Method injection is similar to property injection, except a dependency is provided through a method rather than a property:

```
public class Person
{
    private IOrgService _orgService;

    public void Initialize(IOrgService orgService)
    {
        if (orgService == null)
            throw new ArgumentNullException(nameof(orgService));

        _orgService = orgService;
    }
}
```

Service Locator

The service locator pattern uses a locator object that encapsulates logic to determine and provide an instance of the dependencies that are needed. Although it will vary depending on your implementation, a sample call with a service locator might look something like the following, resulting in an instance being provided based on the specified interface:

```
var cache = ServiceLocator.GetInstance<ILogger>();
```

Using the service locator pattern to get dependencies is considered to be an anti-pattern by some people because it hides a class's dependencies. As opposed to constructor injection, where we can see the dependencies in the public constructor, we would have to look at the code to find dependencies being resolved through the service locator. Hiding dependencies in this way can lead to runtime or compile-time issues, and make it more difficult to reuse the code. This is particularly true if we do not have access to the source code, which might be the case if we are using code from a third party. It is preferable to use an explicit method to acquire an instance of a dependency.

DI containers

A **Dependency Injection** container, sometimes referred as **DI container** or **IoC container**, is a framework that helps with DI. It creates and injects dependencies for us automatically.

It is not necessary to use a DI container in order to take advantage of DI. However, using a DI container makes handling dependencies much easier. Unless your application is very small in size, leveraging a DI container will eliminate the repetitive grunt work of doing it manually. If you were to write some generic code to automate some of it, you are essentially creating your own container when you could use an existing framework built for that purpose. Dependencies go *n* levels deep and things can get complicated quickly.

If you do opt to use a DI container, there is a variety of containers available. While there are differences among them, they all come with some similar, basic functionality that will facilitate DI. Many of them will also come with other advanced features and your needs will dictate which features you use. Getting DI containers up and running to use the basic features is typically an easy and straightforward process.

Helping your team succeed

One of the many goals of a software architect is to help their team succeed and this can come in many forms. In this section, we will cover some of the practices that a software architect can put into place to help team members succeed. They include unit testing, ensuring that a development environment is easy to set up, pair programming, and code reviews.

Unit testing

Testing is one of those things that some developers do not enjoy doing, even though it is essential to develop high-quality software applications. Developers should test early and often, which is a practice of many agile software development methodologies.

If the practice of writing and executing unit tests is not already part of the organization, software architects should institute the practice. In this section, we will go over some of the essential details of unit testing.

What are unit tests?

Unit testing is the practice of testing the smallest testable units of a software system, such as methods, to ensure that they are working properly. It plays an important role in increasing the quality of your software. Functionality in a software system is decomposed into discrete, testable behaviors that are then tested as units.

Some of the principles we have discussed, such as creating loosely coupled code, following principles such as DRY, SoC, and single responsibility, and using techniques such as DI, create code that is independent and decoupled from dependencies. These qualities make our code testable and allow us to focus a unit test on a single unit.

Benefits of unit testing

There are important benefits to unit testing your software, such as improving the quality of the software. With good unit tests, defects can be found before code is checked in or before a build is attempted. By testing early and often, bugs can be fixed without affecting other code.

One way to make it easier to test early and often is to automate the unit testing. Unit testing is ideal for automation. While developers will want to manually execute unit tests as they change code or prior to check-ins, unit tests can be automatically executed as part of some process, such as a build process. We will discuss automation and build processes in more detail in `Chapter 13`, *DevOps and Software Architecture*.

Debugging is made easier with regular unit testing because the source of the bug can be narrowed down to recent changes. Unit tests also serve as a form of documentation. By looking at the unit tests, one can begin to understand what functionality a particular class provides and how it is supposed to work.

Properties of a good unit test

There are various properties that make for a good unit test. Keep them in mind when writing your unit tests. Unit tests should be atomic, deterministic, automated and repeatable, isolated and independent, easy to set up and implement, and fast.

Atomic

Unit tests should only test a single assumption about a small piece of functionality. This assumption should focus on a behavior of the unit being tested, with the unit typically being a single method. Therefore, multiple tests are necessary to check all of the assumptions for a given unit. If you are testing multiple assumptions in a single test or calling multiple methods in a single test, the scope of your unit test is probably too large.

Deterministic

Unit tests should be deterministic. If no code changes are made, unit tests should yield the same result every time they are executed.

Automated and repeatable

The benefits of unit testing are fully realized when the execution of unit tests are automated and repeatable. This will allow unit tests to be executed as part of a build process.

Isolated and independent

Each unit test should have the ability to be run independently of other tests and in any order. If tests are isolated and independent, it should be possible to execute any unit test at any time.

Unit tests should not depend on anything except the class we are testing. It should not rely on other classes, nor should it depend on things such as connecting to a database, using a hardware device, accessing files on a file system, or communicating across a network.

With a testing framework and a DI framework, we can mock dependencies for our unit tests. A mock object is a simulated object that is instantiated, perhaps with the help of a testing/mocking framework, that mimics the behaviors of the real object. By mocking objects for the dependencies of the class being tested, we keep the unit test independent of other classes. We can control the mock and specify what it will return based on some input.

Easy to set up and implement

Unit tests should be easy to set up and implement. If they are not, this is a *code smell*, which is a symptom in the system that may indicate a larger problem. The problem could be in the way that the unit being tested is designed or in the way that the test is being written.

Although we want a high amount of unit test coverage, we do not want developers to spend inordinate amounts of time writing a single unit test. Unit tests should be easy to write.

Fast

Unit tests should execute quickly. A complex software system that has adequate test coverage will have a large amount of unit tests. The execution of unit tests should not slow down development or build processes. Based on the other desirable properties of unit tests, such as being atomic, isolated, and independent, they should execute quickly.

The AAA pattern

The AAA pattern is a common unit testing pattern. It is a way to arrange and organize test code to make unit tests clear and understandable, and consists of separating each unit test method into three sections: *Arrange*, *Act*, and *Assert*.

Arrange

In this section of the unit test method, you *arrange* any preconditions and inputs for the test. This includes initializing values and setting up mock objects.

Depending on the unit test framework and programming language that you are using, some of them provide a way to specify methods that will execute prior to the execution of the unit tests for that test class. This allows you to centralize arrangement logic that you want to execute prior to the execution of all of your tests.

However, each unit test method should have an arrange section where you are performing initialization for that particular test.

Act

The Act section of a unit test method is where you *act* on the unit that is being tested, referred to as the **System Under Test** (**SUT**). Logic should invoke the method being tested, passing in values, such as mock objects, that were previously arranged.

Assert

The Assert part of a unit test method is where you *assert* that the results are what you expect. It verifies that the method executed and behaved as expected.

Naming conventions for unit tests

When naming your unit test classes and methods, you should follow a naming convention. This provides not just consistency, but allows your tests to be a form of documentation. Unit tests are intention-revealing in that they describe the expected behavior. If meaningful names are provided, everyone can know something about the purpose of the test class and its methods just by looking at the names.

Unit test class names

The naming of the test classes themselves and the namespaces they are in depend on the naming conventions that your project follows for classes. However, you should consider putting the name of the class being tested (the **System Under Test** (**SUT**) in the name of the class. For example, if you are testing a class named `OrderService`, consider naming the unit test class `OrderServiceTests`.

Unit test method names

The unit test methods should be given meaningful names that provide, at a glance, their purpose. Characteristics such as the method being tested, some indication as to the specific condition(s) and input(s) of the test, and the expected result of the test are all useful to provide.

For example, if we are testing the `CalculateShipping` method on the `OrderService` class, we might have test method names such as:

- `CalculateShipping_NullOrder_ThrowsArgumentNullException`
- `CalculateShipping_ValidOrder_CalculatesCorrectAmount`
- `CalculateShipping_ExpeditedShipping_CalculatesCorrectAmount`

The exact naming convention for unit test methods is up to you, but the important point is to come up with one and to use it consistently in order to provide meaningful information to those who will be looking at them.

Code coverage for unit tests

Code coverage is a measure of how much of the source code is being covered by tests. Many tools exist that will help you to calculate code coverage. Software architects should stress the importance of aiming for exhaustive test coverage to ensure that all code paths are tested.

Keep in mind that the code coverage percentage is just part of the consideration when deciding whether or not you have adequate coverage. The code coverage percentage will let you know what percentage of the paths are covered (as in paths that are executed at least once). However, this does not mean that all of the important scenarios concerning your functionality is covered. Code coverage calculations do not consider the range of values that are possible for the various inputs, and additional tests may be necessary to cover different situations.

Keeping unit tests up to date

Unit tests are a form of documentation for your system, describing its functionality and the expected behavior. Software architects should encourage their team to not only execute unit tests regularly but to keep them up to date. As requirements change or new functionality is added, unit tests need to be changed or added.

After changes are made, developers should modify any tests that need to be changed and then execute all of the unit tests to ensure there are no unintended consequences.

One thing that I have seen slip through the cracks before regarding keeping unit tests up to date are bug fixes. If a bug is found, the unit tests did not cover that particular scenario. One or more unit tests should be created that incorporate the situation into the tests to ensure it remains fixed. It is a concept expressed by the saying that *bugs should only be found once*.

Teams that are diligent about updating unit tests for changed requirements and new functionality sometimes miss this important point. As a software architect, you can remind your teams of it. The knowledge gained from discovering and resolving that bug should not be lost. Once it is documented in the tests, from that point forward, the tests will check for that type of bug to ensure that it never returns.

Setting up development environments

Software architects should review the process of setting up a new development environment in order to minimize the amount of time that it takes. Most teams do not want to add any extra time to a schedule that may already be tight. The process should not be more difficult than necessary.

New developers may join the team or existing team members may need to start working on a different machine. All too often, setting up a new development environment to the point where a developer can start coding takes an inordinate amount of time. There may be complexities involved with the setup, but it should be made as easy as possible.

Make sure that the process of granting access to new team members, installing the necessary software (for example, development tools), getting the latest code from version control, compiling the code, and running the application is a smooth process. There are tools available that can create and deploy images of environments for physical and virtual development machines.

Sometimes there are subtle things that need to be done, such as making certain changes to a configuration file, in order to get the application working on a development machine. This type of knowledge can sometimes become *tribal knowledge*, or information that is known by some individuals but is unwritten and not known by everyone.

The goal is to get the developer up and running as quickly as possible so that they can focus on the real complexities of their job. If something is making the process difficult, examine the reasons behind it so that action can be taken to improve it. There may be ways to improve the process from both organizational and technical perspectives.

Providing a README file

A good `README` file should be made available for all projects within an organization. It allows developers to review the steps required to set up an environment. The `README` file should include the following:

- A brief description of the software project
- Instructions indicating what other software, such as development tools, need to be installed; this includes the location of setup files for the software that needs to be installed and any license keys or other information required for the installation
- Any special configuration that is necessary, such as how to point the application to a particular database
- Information on how to connect to version control in order to perform operations such as getting the latest codebase and checking in changes
- A creation date for the `README` file and/or a version number of the software it is intended for so that readers can know whether they are looking at the correct version
- Any relevant licensing information
- Contact information for more help

Pair programming

Pair programming is an agile software development technique where two developers work together on the same deliverable, whether a technical design or coding. In the case of programming, the person who is coding is the *driver* while the other person, who is observing, is the *navigator*. The roles of *driver* and *navigator* can alternate at a prescribed interval (for example, every hour) or the roles can be switched at any time that the two people feel is appropriate. Regardless of the role, each person should be an active participant.

Benefits of pair programming

As a software architect, you may want to consider using this technique with your team as it yields a number of benefits.

Pair programming can improve code quality. Having an additional set of eyes looking at the work may allow the pair to notice a problem or an opportunity that would not have been apparent if each person was working alone. Also, the driver will have a tendency to be more careful when coding with someone else watching, which may lead to better code overall.

Working collaboratively during pair programming to accomplish a goal can be beneficial. If the two individuals have different skillsets, they can bring both to bear on the work, and some of these skills will be transferred. In addition to getting work done, sessions can act similar to a training session in this regard. Working together also helps to enforce and spread knowledge of things such as coding standards.

Pair programming allows developers to become more familiar with the codebase. It provides opportunities for more than one person to be knowledgeable about a particular part of the system. Eventually, if something has to be changed with the code, more than one person will be familiar with it. Pair programming tends to create a culture of collective ownership of the codebase.

To fully realize this benefit, it is a good practice to rotate pairs and not just have the same two people always pair up. More knowledge will be shared when the pairs are rotated.

Pair programming can serve as training for less experienced developers or those who may be new to the project. Using this technique provides an opportunity for a software architect or senior developer to teach someone.

Software architects are encouraged to participate in pair programming. Rather than become an *ivory tower architect*, isolated from the rest of the team, software architects should work closely with team members. By staying active in the codebase, software architects will stay immersed in the project. In addition, software architects can share their knowledge and experience while pair programming with the developers on the team.

Using pair programming when it is needed

The use of pair programming does not have to be an all-or-nothing proposition. Although some teams might choose to do all of their development work in pairs, it certainly does not have to be done that way.

Pair programming sessions can be conducted as often as you like. It may not be beneficial to pair program to accomplish an easy task. You may choose to do it when there is a particular reason to do so, such as bringing two resources together that complement each other due to different skillsets or pairing up a software architect with a junior developer so that the session can serve as a learning experience.

Reviewing deliverables

Software architects are responsible for following an organization's review process and may have a role in shaping the process. Completed deliverables should be reviewed to ensure that they are correct and to identify any potential problems.

It is important for management and software architects to establish a culture within the organization so that the review process is viewed in a positive manner. It is important for team members to understand that the focus of reviews is on helping each other find defects, to learn, and to foster communication among the team.

Some of the methods that can be used to review deliverables include code reviews, formal inspections, and walkthroughs.

Code reviews

Code reviews are evaluations of code, typically conducted by peers. Code reviews involve one or more people, other than the developer of the code being reviewed, who examine code changes to find any problems.

The main focus of code reviews is to find both technical and business logic defects. However, code is also reviewed for other things, such as ensuring coding standards are being followed and finding opportunities for improvement.

An organization should have a process in place for code reviews. Some IDEs, software repositories, and other tools provide functionally that facilitate collaborative code reviewing. The process typically involves the following:

- The author requests a code review from one or more people
- The requested reviewers either accept or decline the request for a review
- Communication takes place between reviewers and the author so that comments and feedback can be exchanged
- Any defects that are found are recorded, assigned (typically to the author), and corrected
- Fixes to defects are tested

Software architects should have some degree of involvement with code reviews. The extent of their involvement may vary depending on the project and the organization, but it is beneficial for the software architect to stay involved at this level.

Reviewers should try not to review too many **lines of code** (**LOC**) at any given time. There is only so much a typical person can process at a time, and once most people go beyond five hundred lines or so, their effectiveness drastically diminishes.

It may be helpful to use a checklist during code reviews. The checklist can serve as a reminder of issues to look out for that have been found to be problematic in the past. Defects that are found during a review need to be recorded so that the issues are not left unaddressed.

Formal inspections

Formal inspections, as you might imagine from the name, are a more structured way of reviewing deliverables. It is a group review method and has been found to be quite effective at finding defects. The main goal of formal inspections is to evaluate and improve the quality of the software system.

Formal inspections are meetings in which deliverables, such as a design or code, are reviewed in order to find defects. Formal inspections are scheduled in advance, and participants are invited to the meeting.

Roles for a formal inspection

In a formal inspection, invited participants are assigned a role that they will play. Roles include leader/moderator, author, reviewer, and recorder/scribe. The following is the description for all the roles:

Leader/moderator: A moderator facilitates the meeting and is responsible for obtaining a productive review. They ensure the meeting progresses at an appropriate pace. They should encourage participation when necessary and follow up on any action items after the meeting is over. They may be required to summarize or provide a report on the inspection.

Author: The author is the person who created the design or wrote the code that is being reviewed in the inspection. Their role can be rather limited during the meeting. They may be required to explain anything that is unclear or to provide reasons why something that appears to be a defect is actually not one.

Reviewer: One or more individuals, other than the author, serve as reviewers for the inspection. They can prepare for inspections by reviewing the deliverables before the meeting takes place. Any notes that were taken during preparation should be brought to the meeting so that those can be communicated at that time.

As is the case with other types of reviews, it may be helpful for reviewers to have a checklist of items that they want to focus on for the review. Reviewers should be technically competent and can give positive as well as negative feedback.

Recorder/scribe: The recorder, or scribe, is responsible for taking notes during the meeting. They should record any defects found as well as action items. Although a scribe could also be a reviewer, the moderator and the author should not play the role of scribe.

Inspection meeting and follow-up

During the actual inspection meeting, reviewers should focus on identifying defects and not on solutions. After the meeting, it is typical that an inspection report is produced, summarizing the results of the inspection.

Any defects that were found during the inspection should be placed in a backlog or otherwise assigned to someone for resolution, such as the author. Follow-up should take place, usually by the moderator, to ensure that any action items were completed.

Walkthroughs

A **walkthrough** is an informal method of review. In a walkthrough, the author of a design or code deliverable hosts a meeting in which they guide reviewers through the deliverable.

Unlike a formal inspection, participants are not assigned specific roles (other than the host/author). Walkthroughs are flexible and an organization can choose how they want to organize a walkthrough based on their needs.

Participants can prepare for a walkthrough by looking at the deliverable beforehand. The focus of the walkthrough is to identify potential defects. Although the focus is not to correct any defects found, unlike formal inspections, the group can decide to allow suggestions of changes that can be made to the deliverable. Similar to formal inspections, management should not attend a walkthrough so as not to influence the meeting in any way.

Walkthroughs have been found to not be as effective as other review methods for evaluating and improving deliverables. However, they do allow a larger number of reviewers to participate at once. This provides an opportunity to get feedback from a more diverse group.

Summary

Even though software engineering is a relatively new discipline compared to other types of engineering, a number of principles and practices have been established to create high-quality software systems.

We learned that to design orthogonal software systems that can be extended while minimizing the impact to existing functionality, we need to focus on loose coupling and high cohesion. To minimize complexity in our software applications, a number of principles can be applied, such as KISS, DRY, information hiding, YAGNI, and SoC.

The SOLID design principles, which include the SRP, OCP, LSP, ISP, and DIP, can be used to create code that is more understandable, maintainable, reusable, testable, and flexible. A number of practices, such as unit testing, pair programming, and reviewing deliverables can be used to identify defects and improve the quality of software systems.

Software architecture patterns are reusable solutions that can be used to solve recurring problems. In the next chapter, we will go over some of the common software architecture patterns so that you will be aware of them and can apply them appropriately to your software applications.