Clean Code Cookbook

Recipes to Improve the Design and Quality of Your Code

Maximiliano Contieri

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Table of Contents

CHAPTER 1

Clean Code

When Martin Fowler defined refactoring in his book *Refactoring: Improving the Design of Existing Code*, he showed the strengths and benefits as well as the reasons behind refactoring. Fortunately, after more than two decades, most developers know the meaning of refactoring and code smells. Developers deal with tech debt on a daily basis and refactoring has become a core part of software development. In his foundational book, Fowler used refactorings to address code smells. This book walks through some of these in the form of semantic recipes to improve your solutions.

1.1 What Is a Code Smell?

A code smell is a symptom of a problem. People tend to think the presence of code smells is proof that the whole entity needs to be taken apart and rebuilt. This is not the spirit of the original definition. Code smells are simply indicators of improvement opportunities. A code smell doesn't necessarily tell you what is wrong; it is telling you to pay special attention.

This book's recipes provide some solutions to those symptoms. Like with any cookbook, recipes are optional and code smells are guidelines and heuristics, not rigid rules. Before applying any recipe blindly, you should understand the problems and evaluate the cost and benefits of your own design and code. Good design involves balancing guidelines with practical and contextual considerations.

1.2 What Is Refactoring?

Going back to Martin Fowler's book, he gives two complementary definitions:

> *Refactoring* (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
>
> *Refactoring* (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

Refactorings were invented by William Opdyke in his 1992 PhD thesis, "Refactoring Object-Oriented Frameworks", and became popular after Fowler's book. They have evolved since Fowler's definition. Most modern IDEs support automatic refactorings. These are safe and make structural changes without changing the behavior of the system. This book has many recipes with automatic, safe refactorings and in addition also includes semantic refactors. Semantic refactors are not safe since they might change part of the behavior of the system. You should apply the recipes with semantic refactors carefully since they can break your software. I will indicate whether a recipe has semantic refactoring in the applicable recipes. If you have good behavioral code coverage, you can be confident you won't break important business scenarios. You should not apply refactoring recipes at the same time you correct a defect or develop a new feature.

Most modern organizations have strong test coverage suites in their continuous integration/continuous delivery pipelines. See *Software Engineering at Google* by Titus Winters et al. (O'Reilly 2020) to find out if you have these test coverage suites.

1.3 What Is a Recipe?

I use the term recipe lightly. A recipe is a set of instructions to create or change something. The recipes in this book work best if you understand the spirit of the recipe so you can apply it with your own flavors. Other recipe books in this series are more concrete with step-by-step solutions. To utilize the recipes in this book, you will need to translate them into your programming language and design solution. The recipe is a vehicle for teaching you how to understand a problem, identify the consequences, and improve your code.

1.4 Why Clean Code?

Clean code is easy to read, understand, and maintain. It is well-structured, concise, and uses meaningful names for variables, functions, and classes. It also follows best practices and design patterns and favors readability and behavior over performance and implementation details.

Clean code is very important in all evolving systems where you make changes daily. It remains particularly relevant in certain environments where it is not possible to implement updates as quickly as desired. This includes embedded systems, space probes, smart contracts, mobile apps, and many other applications.

Classical refactoring books, websites, and IDEs focus on refactors that don't change the system's behavior. This book has some recipes for scenarios like this, like safe renames. But you will also find several recipes related to semantic refactors where you change the way you solve some problems. You will need to understand the code, the problems, and the recipe to make appropriate changes.

1.5 Readability, Performance, or Both

This book is about clean code. Some of its recipes are not the most performant ones. I choose readability over performance when in conflict. For example, I have dedicated a whole chapter (Chapter 16) to premature optimization to address performance problems without enough evidence.

For performance mission-critical solutions the best strategy is to write clean code, cover it with tests, and then improve the bottlenecks using Pareto rules. The Pareto principle applied to software states that by tackling 20% of critical bottlenecks you will improve your software performance by 80%. If you improve 20% of the bad performance, it will likely increase system speed by 80%.

This method discourages you from making premature optimization changes lacking evidence-based scenarios because that will result in little improvement and will damage clean code.

1.6 Software Types

Most of the recipes in this book are targeted at backend systems with complex business rules. The simulator you are going to build starting in Chapter 2 is perfect for it. Since recipes are domain agnostic, you can also use most of them for frontend development, databases, embedded systems, blockchains, and many other scenarios. There are also specific recipes with code samples for UX, frontend, smart contracts, and other specific domains (see Recipe 22.7, "Hiding Low-Level Errors from End Users", for example).

1.7 Machine-Generated Code

Do you need clean code now that there are a lot of available tools for computer-generated code? The answer in 2023 is: yes. More than ever. There are a lot of commercial software coding assistants. However, they don't (yet) have full control; they are the copilots and helpers, and humans are still the ones making design decisions.

At the time of writing this book, most commercial and artificial intelligence tools write anemic solutions and standard algorithms. But they are amazingly helpful when you don't remember how to make a small function and they are handy to translate between programming languages. I've heavily used them while writing this book. I haven't mastered the 25+ languages I've used in the recipes. I've translated and tested several code snippets in different languages with many assistant tools. I invite you also to use all the available tools to translate some of this book's recipes into your favorite language. Tools are here to stay, and future developers will be technological centaurs: half human, half machine.

1.8 Naming Considerations Throughout the Book

Throughout the book, I use the following terms interchangeably:

- Methods/functions/procedures
- Attributes/instance variables/properties
- Protocol/behavior/interface
- Arguments/collaborators/parameters
- Anonymous functions/closures/lambdas

The differences among them are subtle and sometimes language dependent. I add a note when it is needed to clarify usage.

1.9 Design Patterns

This book assumes that the reader has a basic understanding of object-oriented design concepts. Some of the recipes in this book are based on popular design patterns, including those described in the "Gang of Four" *Design Patterns* book. Other recipes feature lesser-known patterns such as *null object* and *method object*. Additionally, this book includes explanations and guidance on how to replace patterns that are now considered antipatterns, such as the *singleton* pattern in Recipe 17.2, "Replacing Singletons".

1.10 Programming Language Paradigms

According to David Farley:

> Our industry's obsession with languages and tools has been damaging to our profession. This doesn't mean that there are no advances to be had in language design, but most work in language design seems to concentrate on the wrong kinds of things, such as syntactic advances rather than structural advances.

The clean code concepts presented in this book can be applied to a variety of programming paradigms. Many of these ideas have roots in structured programming and functional programming, while others come from the object-oriented world. These concepts can help you write more elegant and efficient code in any paradigm.

I will use most of the recipes in object-oriented languages and build a simulator (named *MAPPER*) using objects as metaphors for real-world entities. I reference MAPPER frequently throughout the book. Many recipes will lead you to think of behavioral and declarative code (see Chapter 6, "Declarative Code") instead of implementation code.

1.11 Objects Versus Classes

Most of this book's recipes talk about objects and not classes (though there's an entire chapter about classification: see Chapter 19, "Hierarchies"). For example, Recipe 3.2 is titled "Identifying the Essence of Your Objects" instead of "Identifying the Essence of Your Class." This book talks about using objects to map real-world objects.

The way you create these objects is accidental; you can use classification, prototyping, factories, cloning, etc. Chapter 2 discusses the importance of mapping your objects and the imperative to model things you can see in the real world. To create objects many languages use *classes*, which are artifacts and not obvious in the real world. You need them if you are using a classification language. But they are not the main focus of the recipes.

1.12 Changeability

Clean code is not just about ensuring that your software functions correctly, but also about making it easy to maintain and evolve. Again, according to Dave Farley's *Modern Software Engineering*, you must be experts at learning and making software ready for change. This is a major challenge for the tech industry, and I hope that this book will assist you in keeping up with these developments.

CHAPTER 2

Setting Up the Axioms

2.0 Introduction

Here's a common definition of software:

> The instructions executed by a computer, as opposed to the physical device on which they run (the "hardware").

Software is defined by its opposite; everything that is not hardware. This is not exactly a good definition of what software actually is. Here's another popular definition:

> Software, instructions that tell a computer what to do. Software comprises the entire set of programs, procedures, and routines associated with the operation of a computer system. The term was coined to differentiate these instructions from hardware—i.e., the physical components of a computer system. A set of instructions that directs a computer's hardware to perform a task is called a program, or software program.

Many decades ago, software developers realized that software is much more than instructions. Throughout this book, you will think about system behavior and come to realize that the primary purpose of software is:

> To mimic something that happens in a possible reality.

This idea returns to the origins of modern programming languages like *Simula*.

Simula

Simula was the first object-oriented programming language to incorporate classification. Its name clearly indicated that the purpose of software construction was to create a simulator. This is still the case with most computer software applications today.

In science, you build simulators to understand the past and forecast the future. Since Plato's time, human beings have tried to build good models of reality. You can define software as the construction of a simulator with the acronym *MAPPER*:

> Model: Abstract Partial and Programmable Explaining Reality

This acronym will appear frequently throughout the book. Let's dive into what makes up MAPPER.

2.1 Why Is It a Model?

A model is a result of viewing a certain aspect of reality through a specific lens and perspective, using a particular paradigm. It is not the ultimate, unchanging truth, but rather the most accurate understanding you currently have based on your current knowledge. The goal of a software model, like any other model, is to predict real-world behavior.

Model

A *model* explains the subject it is describing using intuitive concepts or metaphors. The final goal of a model is the understanding of how something works. According to Peter Naur, "To program is to build theory and models."

2.2 Why Is It Abstract?

The model emerges from the sum of the parts. And you cannot fully understand it by looking at the isolated components. The model is based on contracts and behavior, and they don't necessarily detail how you should make things happen.

2.3 Why Is It Programmable?

You should run your model in a simulator reproducing the desired conditions, which could be a Turing model (such as modern commercial computers), a quantum computer (future computers), or any other type of simulator capable of keeping up with the model's evolution. You can program the model to respond to your actions in certain ways and then observe how it evolves on its own.

Turing Model

A computer based on the *Turing model* is a theoretical machine that is able to perform any computable task for which a set of instructions, or algorithm, can be written. The Turing machine is considered the theoretical foundation of modern computing, and it serves as a model for the design and analysis of actual computers and programming languages.

2.4 Why Is It Partial?

In order to model the problem that interests you, you will only consider a partial aspect of reality. It is common in scientific models to simplify certain aspects that are not relevant in order to isolate a problem. When conducting scientific experiments, you need to isolate certain variables and fix the rest in order to test hypotheses.

In your simulator, you will not be able to model the entire reality, only a relevant portion of it. You do not need to model the entire subject of observation (i.e., the real world), just the interesting behavior. Many of the recipes in this book address the problem of overdesigning models by including unnecessary details.

2.5 Why Is It Explanatory?

The model must be declarative enough to observe its evolution and help you to reason about and predict behavior in the reality you are modeling. It should be able to explain what it is doing and how it is behaving. Many modern machine learning algorithms do not provide information about how they arrive at their output values (sometimes even having hallucinations), but models should be able to explain what they have done, even if they do not reveal the specific steps taken to achieve it.

To Explain

Aristotle said that "to explain is to find the causes." According to him, every phenomenon or event has a cause or series of causes that produce or determine it. The goal of science is to identify and understand the causes of natural phenomena and, from there, to predict how they will behave in the future.

For Aristotle, "to explain" consisted of identifying and understanding all of these causes and how they interact with each other to produce a particular phenomenon. "To predict," on the other hand, refers to the ability to use this knowledge of causes to predict how a phenomenon would behave in the future.

2.6 Why Is It About Reality?

The model has to reproduce conditions that occur in an observable environment. The ultimate goal is to forecast the real world, like any simulation. You will hear a lot about reality, the real world, and real-world entities in this book. The real world will be your ultimate source of truth.

2.7 Inferring the Rules

Now that you have a starting point for what software is, you can begin to infer good modeling and design practices. The MAPPER principles will appear throughout the book's recipes.

In the following chapters, you will continue to discover principles, heuristics, recipes, and rules to build excellent software models starting with the simple axiom introduced here: Model: Abstract Partial and Programmable Explaining Reality. The working definition of software for this book is "a simulator honoring the MAPPER acronym."

Axiom

An *axiom* is a statement or proposition that is assumed to be true without proof. It allows you to build a logical framework for reasoning and deduction, by establishing a set of fundamental concepts and relationships that can be used to derive further truths.

2.8 The One and Only Software Design Principle

If you build the entire paradigm for software design on a single minimal rule, you can keep it simple and make excellent models. Being minimalist and axiomatic means you can derive a set of rules from a single definition:

> The behavior of each element is part of the architecture insofar as that behavior can help you reason about the system. The behavior of elements embodies how they interact with each other and with the environment. This is clearly part of our definition of architecture and will have an effect on the properties exhibited by the system, such as its runtime performance.
>
> —Bass et al., *Software Architecture in Practice*, 4th Edition

One of the most underrated quality attributes of software is being predictable. Books often teach you that software should be fast, reliable, robust, observable, secure, etc. Being predictable is rarely one of the top five design priorities. As a thought experiment, try to imagine designing object-oriented software following just one principle (as illustrated in Figure 2-1): "Each domain object must be represented by a single object in the computable model and vice versa." Then try to derive all design rules, heuristics, and recipes from that single premise to make your software predictable following this book's recipes.

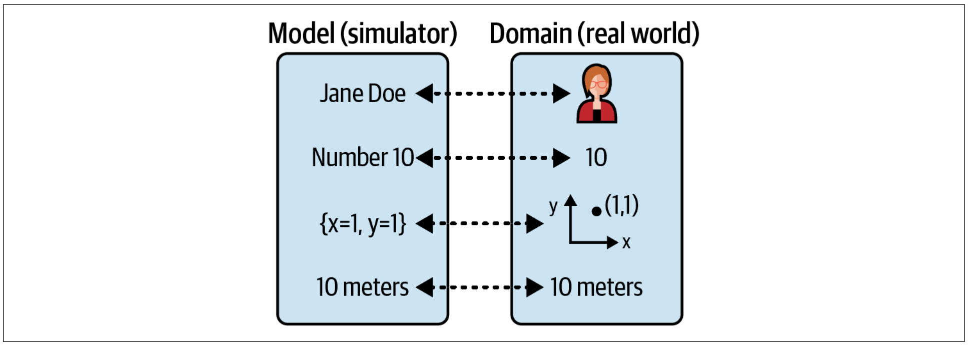


Figure 2-1. The relationship between objects of the model and entities of the real world is 1 to 1

The Problem

You will come to understand by reading these clean code recipes that most of the language implementations used in the industry ignore the single axiom rule, causing enormous problems. Most contemporary languages are designed to address implementation difficulties that arose in the construction of software three or four decades ago when resources were scarce and computations needed to be optimized by the programmer. Now, these problems are present in very few domains. The recipes in this book will help you recognize, understand, and deal with these problems.

Models to the Rescue

When building models of any kind, it's important to simulate the conditions that occur in the real world. You can track each element of interest in the simulation and stimulate it to observe whether they change in the same way as they do in the real world. Meteorologists use mathematical models to predict and forecast the weather, and many scientific disciplines rely on simulations. Physics seeks unifying models to understand and predict real-world rules. With the advent of machine learning, you also build opaque models to visualize behavior in real life.

The Importance of the Bijection

In mathematics, a *bijection* is a function that is one-to-one, meaning that it maps every element in the domain to a unique element in the range and every element in the range can be traced back to a unique element in the domain. In other words, a bijection is a function that establishes a *one-to-one correspondence* between the elements of two sets.

An *isomorphism*, on the other hand, is a stronger type of correspondence between two mathematical structures that preserves the structure of the objects being related. Specifically, an isomorphism is a bijective function that preserves the operations of the structures. A bijection is a one-to-one correspondence between two sets.

In the domain of software, you will always have one and only one object representing a real-world entity. Let's see what happens if you don't comply with the principles of bijection.

Common Cases That Violate Bijection

Here are four common cases that violate the principle of bijection.

Case 1

You have an object in your computable model to represent more than one real-world entity. For example, many programming languages model algebraic measures using just the scalar magnitude. Here's what happens in this scenario, as illustrated in Figure 2-2.

- You can represent *10 meters* and *10 inches* (two completely different entities in the real world) by a single object (*the number 10*).
- You could add them together, getting the model to show that the *number 10* (representing *10 meters*) plus the *number 10* (representing *10 inches*) is equal to the *number 20* (representing who knows what).

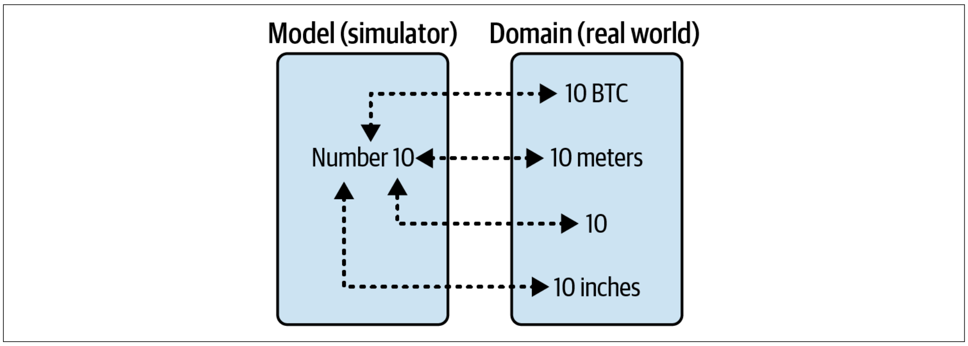


Figure 2-2. The number 10 represents more than one real-world entity

The bijection is broken and this generates problems not always captured in time. Because it is a semantic problem, the error usually occurs long after the failure as in the famous case of the Mars Climate Orbiter.

Mars Climate Orbiter

The *Mars Climate Orbiter* was a robotic space probe launched by NASA in 1998 with the goal of studying the Martian climate and atmosphere. The mission was ultimately unsuccessful due to a problem with the spacecraft's guidance and navigation system. The spacecraft's thrusters were programmed to use metric units of force, while the ground control team was using English units of force. This error caused the spacecraft to come too close to the planet's surface, and it was destroyed upon entry into the Martian atmosphere. The problem with the Mars Climate Orbiter was a failure to properly coordinate and convert units of measurement, leading to a catastrophic error in the spacecraft's trajectory. The probe exploded by mixing different units of measurement. It was a major setback for NASA and it cost the agency $125 million. The failure also led to a number of changes at NASA, including the creation of a new office of safety and mission assurance (see Recipe 17.1, "Making Hidden Assumptions Explicit").

Case 2

The computable model represents the same real-world entity with two objects. Suppose in the real world, there is an athlete *Jane Doe* who competes in one discipline but who is also a judge in another athletic discipline. A single person in the real world should be a single object in the computable model. You need to model just the minimum behavior to fulfill your partial simulation.

If you have two different objects (a competitor and a judge) that represent Jane Doe, you will sooner or later have inconsistencies if you assign some responsibility to one of the two and do not see it reflected in the other (as illustrated in Figure 2-3).

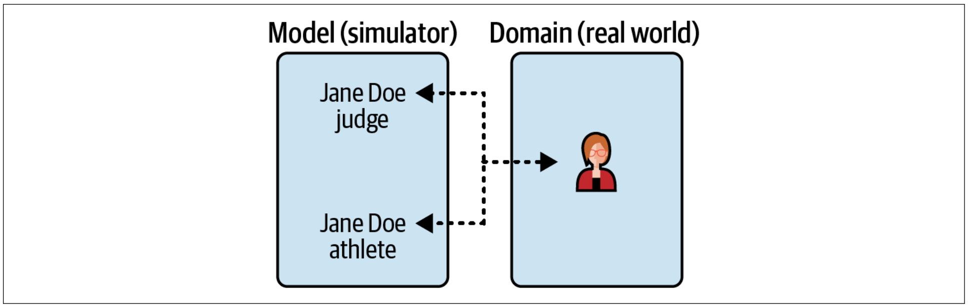


Figure 2-3. Jane Doe is represented in the model by two different entities

Case 3

A bitcoin wallet can be represented as an anemic object with some properties regarding address, balance, etc. (see Recipe 3.1, "Converting Anemic Objects to Rich Objects") or as a rich object (with responsibilities such as receiving transactions, writing to a blockchain, answering the balance, etc.) since they are related to the same concept.

You must stop seeing entities as data structures with attributes; think of them instead as objects, and understand that they are the same object fulfilling different roles based on the context in which they are interacting. Chapter 3, "Anemic Models", includes several recipes to help you reify your objects and convert them into behavioral entities.

Object Reification

Object *reification* is a process in which an abstract concept or idea is given concrete form representing a specific concept or idea as well as providing behavior to anemic and data-oriented objects. By giving a concrete form to abstract concepts through the creation of objects, you will be able to manipulate and work with these concepts in a systematic and structured way.

Case 4

In most modern object programming languages, a *date* can be constructed by creating it from its *day, month, and year*. If you input the date November 31, 2023, many popular programming languages will gently return a valid object (probably December 1, 2023).

This is presented as a benefit but hides certain errors that may occur during data loading. The error will raise when running a nightly batch processing these invalid dates far from the root cause, violating the fail fast principle (see Chapter 13, "Fail Fast").

Fail Fast Principle

The *fail fast principle* states you should break execution as early as possible when there is an error instead of ignoring it and failing as a consequence later on.

The Effect on Language When You Build Models

This book is about improving dirty, nondeclarative, cryptic, and prematurely optimized code. The knowledge of the world is defined by the language you speak, and you need good metaphors for your objects and behaviors, as posited by the Sapir-Whorf hypothesis.

Sapir-Whorf Hypothesis

The *Sapir-Whorf hypothesis*, also known as the theory of linguistic relativity, suggests that the structure and vocabulary of a person's language can influence and shape their perception of the world around them. The language you speak not only reflects and represents reality but also plays a role in shaping and constructing it. This means that the way you think and experience the world is partially determined by the language you use to describe it.

If you think of objects as "data holders" your models will lose the *bijective* property and will break the MAPPER. You will find many related recipes in Chapter 3, "Anemic Models". If you treat objects as data keepers, your computational model (the software you are creating) will not be able to accurately predict and simulate the real world. Your customers will notice the software no longer helps them do their job. This is a common source of software defects (commonly—and badly—named bugs).

Bug

The term *bug* is a common industry misconception. In this book, I talk instead about defects. The original bugs were related to external insects entering warm circuits and messing with the software output. This is no longer the case. It is recommended to employ the term *defect* as it pertains to something introduced rather than an external invader.

CHAPTER 3

Anemic Models

Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.

—Bertrand Meyer, *Object-Oriented Software Construction*

3.0 Introduction

Anemic domain models, or simply anemic objects, are objects consisting of a bunch of attributes without real behavior. An anemic object is often referred to as a "data object" because it mainly serves to store data, but it lacks any meaningful methods or operations that can be performed on that data.

Exposing data to the outside world through *getters* and *setters* can violate the principle of *encapsulation*, as it allows external sources to access and potentially modify the data within an object rather than keeping it contained within the object itself. This can make the object more susceptible to corruption or unintended changes.

The anemic way of designing can also lead to a more procedural style of programming, where the main focus is on manipulating data rather than encapsulating it within objects that have meaningful behavior. This book encourages you to create *rich* objects. Rich objects have a more robust set of behaviors and methods that allow them to perform meaningful operations and provide a single point of access, avoiding repeated logic.

Encapsulation

Encapsulation refers to protecting the responsibilities of an object. You can usually achieve this by abstracting the actual implementation. It also provides a way to control access to an object's methods. In many programming languages, it is possible to specify the visibility of an object's properties and methods, which determines whether they can be accessed or modified by other parts of the program. This allows developers to hide the internal implementation details of an object and only expose the behavior that is necessary for other parts of the program to use.

3.1 Converting Anemic Objects to Rich Objects

Problem

You want to protect your objects from external manipulation and expose behavior instead of data and structures to restrict change propagation.

Solution

Convert all your attributes to private.

Discussion

Imagine your domain has evolved and you need to keep up with your customer business rules in the music world. Songs now need to have a genre associated with them and you need to add it. You also want to protect their essential attributes. Firstly, here is a class definition representing Song metadata in the real world:

```
public class Song {
   String name;
   String authorName;
   String albumName;
}
```

In this example, you treat the `Song` as a tuple (a fixed collection of elements). The goal here is to manipulate its attributes in several repeated places throughout the code so that you can edit the artist or album. Since song manipulation is repeated you will have multiple impacts and places of change.

Change the visibility of your attributes from `public` to `private`:

```
public class Song {
   private String name;
   private Artist author; // Will reference rich objects
   private Album album; // instead of primitive data types

   public String albumName() {
      return album.name() ;
   }
}
```

From now on, you will need to rely on `Song`'s public behavior favoring encapsulation.

If you change the visibility to `private`, you cannot access attributes until you add new methods to handle them. External manipulation is usually duplicated across the system (see Recipe 10.1, "Removing Repeated Code"). If your objects have `public` attributes, they will mutate and change in unexpected ways.

According to your MAPPER defined in Chapter 2, you should design objects based on their behavior, rather than creating anemic objects that only model data. Note that some languages like Java, C++, C#, or Ruby have attribute visibility, while others like JavaScript don't have it at all. Some, like Python or Smalltalk, follow cultural conventions where visibility is documented but not enforced. This is not a safe refactoring as defined in Chapter 1. Changing an attribute's privacy may break existing dependencies.

> Applying this and many other recipes requires a comprehensive test suite acting as a safety net for potential defects while changing the code's behavior. Michael Feathers addresses how to create and maintain this safety net in his book *Working Effectively with Legacy Code*.

Related Recipes

Recipe 3.5, "Removing Automatic Properties"

Recipe 10.1, "Removing Repeated Code"

3.2 Identifying the Essence of Your Objects

Problem

You want to create invariants (see Recipe 13.2, "Enforcing Preconditions") from your objects and keep them valid all the time.

Solution

Don't allow changes to the essential attributes or behavior. Set the essential attributes during object creation and protect them from changing once you create your object.

Discussion

Every real-world entity has essential behavior. The behavior determines that it is a certain object and no other one. This behavior is present in the *bijection* mapped to the real world. The essence is an object's DNA. It cannot change once it is born; you cannot manipulate it. Objects can mutate in accidental ways, not in essential ones.

Essence and Accident

In his book *The Mythical Man-Month*, computer scientist Fred Brooks uses the terms "accidental" and "essential" to refer to two different types of complexity in software engineering using Aristotle's definition.

"Essential" complexity is inherent in the problem being solved and cannot be avoided since it is the complexity that is necessary for the system to function as intended and present in the real world. For example, the complexity of a space landing system is essential because it is required to safely land a rover.

"Accidental" complexity arises from the way in which the system is designed and implemented, rather than from the nature of the problem being solved. It can be reduced by creating good designs. Unneeded accidental complexity is one of the biggest issues in software and you will find many solutions in this book.

Let's see what happens when you change the month of a `Date`:

```
const date = new Date();
date.setMonth(4);
```

This is invalid in the real world since the month is essential to any date but is valid in many programming languages. Changing the month makes it another date in the real world. Calling `setMonth()` with a single argument will only set the month value, leaving the day and year values unchanged.

Every other object relying on the date now changes as a consequence of the ripple effect. If you have a payment due on the date you created earlier and then change the date's essence, then your payment will be silently affected. A better solution would be to change the `payment` reference to a new date by invoking an actual protocol like `defer()`. This change will only affect the `payment` object and cause a restricted ripple effect.

Ripple Effect

The *ripple effect* refers to the way in which a change or modification to one part of a system can have unintended consequences on other parts of the system. If you make a change to a particular object, it could potentially affect other parts of the system that depend on it. This could result in errors or unexpected behavior in those other parts of the system.

Here's a better option:

```
const date = new ImmutableDate("2022-03-25");
// The date essence is identified and you will never change it from now on
```

Once you create the date, it is immutable. You can trust it to forever be mapped to the same real-world entity. It's important to model which attributes/behaviors are essential and which are accidental: what is *essential* in the real world must be *essential* in your model and vice versa.

Many modern languages fail to identify and protect the essence of objects. The `Date` class is a common example. Nevertheless, there are alternative robust packages available for calendar manipulation in most of them.

Related Recipes

Recipe 5.3, "Forbidding Changes in the Essence"

Recipe 17.11, "Avoiding the Ripple Effect"

3.3 Removing Setters from Objects

Problem

You want to protect your objects from external manipulation using setters and favor immutability.

Solution

After you make your attributes private (see Recipe 3.1, "Converting Anemic Objects to Rich Objects"), remove all the setters.

Discussion

This is a classic example of a `Point` class with setters:

```java
public class Point {
   protected int x;
   protected int y;

   public Point() { }

   public void setX(int x) {
        this.x = x;
   }

   public void setY(int y) {
        this.y = y;
  }
}

Point location = new Point();
// At this moment, it is not clear which points are represented
// It is coupled to the constructor decision
// Might be null or some other convention

location.setX(1);
// Now you have point(1,0)

location.setY(2);
// Now you have point(1,2)

// If you are setting essential properties move
// them to the constructor and remove the setter method
```

Here is a shorter version after you apply the recipe and remove the setters:

```java
public class Point {
   public Point(int x, int y) {
        this.x = x;
        this.y = y;
     // You remove the setters
   }

Point location = new Point(1, 2);
```

Setters favor mutability (see Chapter 5, "Mutability") and anemic models. You should only change your objects by invoking a method whose secondary effect is the change. This also follows the "Tell, don't ask" principle.

"Tell, Don't Ask" Principle

The "Tell, don't ask" principle defines a way to interact with objects by invoking their methods instead of asking for their data.

Adding setters to an object makes it mutable in unexpected ways, and you need to add invariant integrity controls in multiple places. This leads to repeated code (see Recipe 10.1, "Removing Repeated Code"). Methods with the name *`setXXX()`* violate MAPPER naming since they seldom exist in the real world, and you should never mutate objects in their essence, as you will see in Chapter 4. Many languages support mutating dates (using, for example, `date.setMonth(5)`), but these operations create a ripple effect on every object relying on the mutated ones.

Related Recipes

Recipe 3.5, "Removing Automatic Properties"

Recipe 3.7, "Completing Empty Constructors"

Recipe 3.8, "Removing Getters"

3.4 Removing Anemic Code Generators

Problem

You are using anemic code generators but want to have more control over your attributes to make them rich objects, avoid code duplication, and focus on behavior instead of data.

Solution

Remove code wizards and generators. If you want to avoid repetitive work you need to apply Recipe 10.1, "Removing Repeated Code" and create an intermediate object with the repeated behavior.

Discussion

Code wizards were trending during the '90s. Many projects were measured using lines of code, and having large codebases was a virtue. You used automated code generators where you put a class template with attributes and they generated the code automatically. This led to code duplication and hard-to-maintain systems.

Today, code assistants like *Codex*, *Code Whisperer*, *ChatGPT*, or *GitHub Copilot* create code in a similar way when prompted. As we have discussed in previous recipes, you should avoid anemic code, and this is what is produced by the current state of the art of artificial intelligence code assistants.

Here is an example of an anemic class generated with metaprogramming (see Chapter 23, "Metaprogramming"):

```
AnemicClassCreator::create(
    'Employee',
    [
        new AutoGeneratedField(
            'id', '$validators->getIntegerValidator()'),
        new AutoGeneratedField(
            'name', '$validators->getStringValidator()'),
        new AutoGeneratedField(
            'currentlyWorking', '$validators->getBooleanValidator()')
    ]);
```

Metaprogramming creates magic setters and getters under the hood:

```
getId(), setId(), getName(), ….
// validation is not explicit
```

You load the class with a class autoloader:

```
$john = new Employee;
$john->setId(1);
$john->setName('John');
$john->setCurrentlyWorking(true);

$john->getName();
// returns 'John'
```

To apply this recipe, you need to make the code explicit, readable, and debuggable:

```
final class Employee {
    private $name;
    private $workingStatus;

    public function __construct(string $name, WorkingStatus $workingStatus) {
        // Constructor and initialization code goes here
    }

    public function name(): string {
        return $this->name;
        // This is not a getter.
        // It is Employee's responsibility to tell her/his name
        // Accidentally, you have implemented an attribute with the same name
    }
}

// You have no magic setters or getters
// All methods are real and can be debugged
// Validations are implicit
// since the WorkingStatus object is valid by construction

$john = new Employee('John', new HiredWorkingStatus());
$john->name(); // returns 'John'
```

While this seems tedious, finding missing abstractions to deal with explicit code is much better than generating anemic code.

Related Recipes

Recipe 10.1, "Removing Repeated Code"

Recipe 23.1, "Removing Metaprogramming Usage"

3.5 Removing Automatic Properties

Problem

You have code using automatic properties favoring fast and uncontrolled creation of anemic objects before thinking about behavior.

Solution

Remove automatic properties. Create by hand the only one you need and implement clear behavior on the MAPPER.

Discussion

Anemic objects and setters violate integrity controls and the fail fast principle. There is an entire chapter on this topic (Chapter 13, "Fail Fast"). This is a `Person` object with an automatic name property creating `getName()` and `setName()` anemic accessors:

```
class Person
{
  public string name
  { get; set; }
}
```

The automated properties tool favors anemic objects. The solution is to make this explicit:

```
class Person
{
  private string name;

  public Person(string personName)
  {
    name = personName;
    // immutable
    // no getters, no setters
  }

  // ... more protocol, probably accessing private variable name
}
```

Setters and *getters* are bad industry practices. This is a language feature and many IDEs also favor this practice. You need to think carefully before exposing your properties for accidental convenience.

Some languages provide explicit support to build anemic models and *DTOs* (see Recipe 3.6, "Removing DTOs"), and you need to understand the consequences of using these kinds of features; the first step is to stop thinking about properties and focus solely on behavior.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.3, "Removing Setters from Objects"

Recipe 3.4, "Removing Anemic Code Generators"

Recipe 3.6, "Removing DTOs"

Recipe 3.8, "Removing Getters"

Recipe 4.8, "Removing Unnecessary Properties"

3.6 Removing DTOs

Problem

You want to have full objects and transfer them between layers.

Solution

Use real objects and avoid data objects. You can transfer anemic data with arrays or dictionaries instead, and if you need to transfer partial objects you can use *proxies* or *null objects* (see Recipe 15.1, "Creating Null Objects") to break the reference graph.

Discussion

DTOs

A *DTO (Data Transfer Object)* is used to transfer data between different layers of an application. It is a simple, serializable, and immutable object that carries data between the application's client and server. The only purpose of a DTO is to provide a standard way of exchanging data between different parts of the application.

DTOs or *data classes* are widely used anemic object tools. They don't enforce business rules, and the logic to deal with them is often duplicated. Some architectural styles favor creating many DTOs to mirror real objects. This solution pollutes the namespace with anemic objects and makes it harder to maintain systems. Once you need to change an object, you also must update the DTO, so there is a lot of duplicate effort.

DTOs are anemic, may carry inconsistent data, enforce code duplication, pollute namespaces (see Recipe 18.4, "Removing Global Classes") with useless classes, and create ripple effects. Also, data integrity is harder to enforce since it can be duplicated across the system.

Here's a domain class `SocialNetworkProfile` with its associated DTO `SocialNetworkProfileDTO` to carry information from one place to another:

```
final class SocialNetworkProfile {

    private $userName;
    private $friends; // friends is a reference to a large collection
    private $feed; // feed references the whole user feed

    public function __construct($userName, $friends, UserFeed $feed) {
        $this->assertUsernameIsValid($userName);
        $this->assertNoFriendDuplicates($friends);
        $this->userName = $userName;
        $this->friends = $friends;
        $this->feed = $feed;
        $this->assertNoFriendofMylsef($friends);
    }
}

// If you need to transfer to an external system you need
// to duplicate (and maintain) the structure

final class SocialNetworkProfileDTO {

    private $userName; // duplicated to be synchronized
    private $friends; // duplicated to be synchronized
    private $feed; // duplicated to be synchronized
    public function __construct() {
        // Empty constructor without validations
    }

    // No protocol, just serializers
}

// If you need to transfer to an external system you create an anemic DTO
$janesProfileToTransfer = new SocialNetworkProfileDTO();
```

Here's an explicit version without anemic DTOs:

```
final class SocialNetworkProfile {

    private $userName;
    private $friends;
    private $feed;

    public function __construct(
        $userName,
        FriendsCollection $friends,
        UserFeedBehavior $feed)
        {
            $this->assertUsernameIsValid($userName);
            $this->assertNoFriendDuplicates($friends);
            $this->userName = $userName;
            $this->friends = $friends;
            $this->feed = $feed;
            $this->assertNoFriendOfMyself($friends);
        }
    // lots of protocol associated with the profile
    // No serialization protocol
    // No behavior or attribute duplication
}

interface FriendsCollectionProtocol { }

final class FriendsCollection implements FriendsCollectionProtocol { }

final class FriendsCollectionProxy implements FriendsCollectionProtocol {
    // proxy protocol
    // travels as a lightweight object and can get contents when requested
}

abstract class UserFeedBehavior { }

final class UserFeed extends UserFeedBehavior { }

final class NullFeed extends UserFeedBehavior {
    // throws an error when requested for behavior
}

// If you need to transfer to an external system you create a valid object
$janesProfileToTransfer = new SocialNetworkProfile(
    'Jane',
    new FriendCollectionProxy(),
    new NullFeed()
);
```

You can check for *anemic* classes with no business object behavior (removing serializers, constructors, mutators, etc).

DTOs are a tool and an established practice in some languages. You should use them with care and responsibility, and if you need to disassemble your objects in order to send them away from your realms, you need to be extremely cautious since dismembered objects have no integrity considerations.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.7, "Completing Empty Constructors"

Recipe 16.1, "Avoiding IDs on Objects"

Recipe 17.13, "Removing Business Code from the User Interface"

See Also

Martin Fowler's blog post on DTO abuses

"Data Class" on Refactoring Guru

3.7 Completing Empty Constructors

Problem

You have empty constructors—where these empty objects do not exist in the real world—without some essential initialization, and you want to have complete and valid objects all the time.

Solution

Pass all your essential arguments when creating objects and use one complete and single constructor.

Discussion

Objects created without arguments are often mutable, unpredictable, and inconsistent. Nonparameterized constructors are a code smell of an invalid object that will dangerously mutate. Incomplete objects cause lots of issues and you should not change an object's essential behavior after you create it. If you can mutate them, all your references will be unreliable from this point. In Chapter 5, "Mutability", we will discuss this problem in detail. Many languages allow you to create invalid objects (for example an empty `Date`).

Here's an example of an anemic, mutable, and inconsistent person:

```
public Person();
// Anemic and mutable
// Does not have the essence to be a valid person
```

A better `Person` model with the essential attributes would be:

```
public Person(String name, int age) {
     this.name = name;
     this.age = age;
     }

// You 'pass' the essence to the object
// So it does not mutate
```

Stateless objects are a valid counterexample. You should not apply this recipe to them.

Some persistence frameworks in statically typed languages require an empty constructor. This is a bad decision. You can live with this, but it is a backdoor for bad models, and you should always create complete objects and make their essence immutable so they endure through time.

Every object needs its essence to be valid at inception. This is related to Plato's ideas about essential immutability. Accidental aspects like age and physical place can change. Immutable objects favor bijection and survive the passing of time.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.3, "Removing Setters from Objects"

Recipe 3.8, "Removing Getters"

Recipe 11.2, "Reducing Excess Arguments"

3.8 Removing Getters

Problem

You want to have control over your objects' access and hide accidental representation as much as possible to have the freedom to change without breaking software.

Solution

Remove your object's getters. Invoke some explicit method based on the behavior and not the data, and protect your implementation decisions. Use domain names instead.

Discussion

If you avoid the `getXXX` prefix, you favor information hiding and encapsulation principles and your design will quickly require change. Models with getters are usually tightly coupled and less encapsulated.

Information Hiding

Information hiding aims to reduce the complexity of a software system by separating its internal workings from its external interface. This allows the internal implementation of a system to change without affecting the way it is used by other systems or users.

One way to achieve information hiding is through the use of abstractions on the MAPPER, which provides a simplified view of a system's functionality and hides the underlying details.

This is a classic window implementation:

```
final class Window {
    public $width;
    public $height;
    public $children;

    public function getWidth() {
        return $this->width;
    }

    public function getArea() {
        return $this->width * $this->height;
    }

    public function getChildren() {
        return $this->children;
    }
}
```

The properties of a window should not be accessed by such getters. This is an incompatible but better version focusing on actual window behavior:

```
final class Window {
    private $width;
    private $height;
    private $children;

    public function width() {
        return $this->width;
    }

    public function area() {
        return $this->height * $this->width;
    }

    public function addChildren($aChild) {
        // Do not expose internal attributes
        return $this->children[] = $aChild;
    }
```

Getters coincide in certain scenarios with true responsibility. It will be reasonable for a window to return its color and it may accidentally store it as color. So, a `color()` method returning the attribute `color` might be a good solution. `getColor()` breaks the bijection rule since it is implementational and has no real counterpart on the MAPPER.

Some languages return getters with references to private objects. This breaks encapsulation. Some of them return internal collections (instead of copies), therefore a client can modify, add, or remove elements without invoking safer and more protective protocols. This also breaks Demeter's law.

Demeter's Law

Demeter's law is a principle stating that an object should only communicate with its immediate neighbors, and should not know the inner workings of other objects. To favor Demeter's law, you need to create objects that are loosely coupled, meaning that they are not highly dependent on each other. This makes the system more flexible and easier to maintain, as changes to one object are less likely to have unintended consequences on other objects.

An object should only access the methods of its immediate neighbors, rather than reaching into other objects to access their internals. This helps to reduce the level of coupling between objects and makes the system more modular and flexible.

```
public class MyClass {
  private ArrayList<Integer> data;

  public MyClass() {
    data = new ArrayList<Integer>();
  }

  public void addData(int value) {
    data.add(value);
  }

  public ArrayList<Integer> getData() {
    return data; // breaking encapsulation
  }
}
```

In this example in Java, the `getData()` method returns a reference to the internal data collection, instead of creating a copy of it. This means that any changes made to the collection outside the class will be directly reflected in the internal data, and may cause unexpected behavior or defects in the code.

It is important to note that this can be a security hole: a client can modify the internal state of the object without going through the intended behavior, i.e., the methods of the class (see Chapter 25, "Security").

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.3, "Removing Setters from Objects"

Recipe 3.5, "Removing Automatic Properties"

Recipe 8.4, "Removing Getter Comments"

Recipe 17.16, "Breaking Inappropriate Intimacy"

3.9 Preventing Object Orgy

Problem

You have code that violates the rules of other objects' encapsulated properties.

Solution

Protect the properties and expose only behavior.

Discussion

Object Orgy

An *object orgy* describes a situation in which objects are insufficiently encapsulated, allowing unrestricted access to their internals. This is a common antipattern in object-oriented design and can lead to increased maintenance and increased complexity.

If you see your objects as data holders, you will violate their encapsulation, but you shouldn't. As in real life, you should always ask for consent. Accessing other objects' properties breaks the information hiding principle and generates strong coupling.

Note that coupling to essential behavior, interfaces, and protocol is a better decision than coupling to data and accidental implementation.

Consider the familiar `Point` example:

```
final class Point {
    public $x;
    public $y;
}

final class DistanceCalculator {
    function distanceBetween(Point $origin, Point $destination) {
        return sqrt((($destination->x - $origin->x) ^ 2) +
            (($destination->y - $origin->y) ^ 2));
    }
}
```

Following is a more abstract `Point` that does not depend on how you store the information and follows the "Tell, don't ask" principle (see Recipe 3.3, "Removing Setters from Objects"). The `Point` has changed its internal and accidental representation, storing its position using polar coordinates:

```
final class Point {
    private $rho;
    private $theta;

    public function x() {
        return $this->rho * cos($this->theta);
    }

    public function y() {
        return $this->rho * sin($this->theta);
    }
}

final class DistanceCalculator {
    function distanceBetween(Point $origin, Point $destination) {
        return sqrt((($destination->x() - $origin->x() ^ 2) +
            (($destination->y() - $origin->y()) ^ 2)));
    }
}
```

If your classes are polluted with setters, getters, and public methods you will certainly have ways to couple to their accidental implementation.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.3, "Removing Setters from Objects"

3.10 Removing Dynamic Properties

Problem

You use properties in a class without declaring them.

Solution

Be explicit with your attributes.

Discussion

Dynamic properties are hard to read, their scope definition is not clear, and they can hide unnoticed typos. You should favor languages forbidding dynamic properties. They break type safety since it's easy to accidentally introduce typos or use the wrong property names. This can lead to runtime errors that can be difficult to debug, especially in larger codebases. They also hide possible name collisions since dynamic properties may have the same name as properties defined in the class or object, leading to conflicts or unexpected behavior.

Here is an example of an undefined property:

```python
class Dream:
    pass

nightmare = Dream()

nightmare.presentation = "I am the Sandman"
# presentation is not defined
# it is dynamic property

print(nightmare.presentation)
# Output: "I am the Sandman"
```

When you define it in the class:

```python
class Dream:
    def __init__(self):
        self.presentation = ""

nightmare = Dream()

nightmare.presentation = "I am the Sandman"

print(nightmare.presentation)
# Output: "I am the Sandman"
```

Dynamic properties are supported in many programming languages like PHP, Python, Ruby, JavaScript, C#, Objective-C, Swift, Kotlin, etc. and many of them have compiler options to avoid them. In these languages, dynamic properties can be added to objects at runtime, and accessed using the object's property accessor syntax.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.5, "Removing Automatic Properties"

CHAPTER 4

Primitive Obsession

The obsession with primitives is a primitive obsession.
—Rich Hickey

4.0 Introduction

Many software engineers think that software is about "moving data around"; object-oriented schools and textbooks focus on the data and attributes when teaching about modeling the real world. This was a cultural bias taught in universities during the '80s and '90s. Industry trends pushed engineers to create entity-relationship diagrams (ERDs) and reason about the business data instead of focusing on the behavior.

Data is more relevant than ever. Data science is growing and the world revolves around data. You need to create a simulator to manage and protect data and expose behavior while hiding information and accidental representation to avoid coupling. The recipes from this chapter will help you identify small objects and hide accidental representation. You will discover many cohesive small objects and reuse them in many different contexts.

Cohesion

Cohesion is a measure of the degree to which the elements within a single software class or module work together to achieve a single, well-defined purpose. It refers to how closely related the objects are to each other and to the overall goal of the module. You can see high cohesion as a desirable property in software design since the elements within a module are closely related and work together effectively to achieve a specific goal.

4.1 Creating Small Objects

Problem

You have big objects containing only primitive types as fields.

Solution

Find responsibilities for small objects in the MAPPER and reify them.

Discussion

Since the early days of computing, engineers map all they see to the familiar primitive data types such as *String*, *Integer*, and *Collection*. Mapping to those data types sometimes violates abstraction and fail fast principles. The Person's name has different behaviors than a string as you can see in the following example:

```
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }
}
```

The concept of names is reified:

```
public class Name {
    private final String name;

    public Name(String name) {
        this.name = name;
        // Name has its own creation rules, comparison, etc.
        // Might be different than a string
    }
}

public class Person {
    private final Name name;

    public Person(Name name) {
        // Name is created as a valid one,
        // you don't need to add validations here
        this.name = name;
    }
}
```

Take the five-letter word from the *Wordle* game as an example. A Wordle word does not have the same responsibilities as a `char(5)` and does not map on the bijection. If you want to create a Wordle game, you will see a bijection between a Wordle word different from a `String` or `char(5)`, since they don't have the same responsibilities. For example, it is not a `String`'s responsibility to find how many matches it has to the secret Wordle word. And it is not the responsibility of a Wordle word to concatenate.

Wordle

Wordle is a popular online word-guessing game where you have six attempts to guess a five-letter word selected by the game. You make each guess by entering a five-letter word, and the game indicates which letters are correct and in the correct position (marked with a green square) and which letters are correct but in the wrong position (marked with a yellow square).

In a very small number of mission-critical systems, there is a trade-off between abstraction and performance. But to avoid premature optimization (see Chapter 16, "Premature Optimization"), you should rely on modern computers and virtual machine optimizations and, as always, you need to stick to evidence in real-world scenarios. Finding small objects is a very hard task, requiring experience to do a good job and avoid overdesign. There's no silver bullet in choosing how and when to map something.

No Silver Bullet

The "no silver bullet" concept is a phrase coined by computer scientist and software engineering pioneer Fred Brooks in his 1986 essay "No Silver Bullet: Essence and Accidents of Software Engineering". Brooks argues that there is no single solution or approach that can solve all of the problems or significantly improve the productivity and effectiveness of software development.

Related Recipes

Recipe 4.2, "Reifying Primitive Data"

Recipe 4.9, "Creating Date Intervals"

4.2 Reifying Primitive Data

Problem

You have objects using too many primitive types.

Solution

Use small objects instead of primitive ones.

Discussion

Suppose you're building a web server:

```
int port = 8080;
InetSocketAddress in = open("example.org", port);
String uri = urifromPort("example.org", port);
String address = addressFromPort("example.org", port);
String path = pathFromPort("example.org", port);
```

This naive example has many problems. It violates the "Tell, don't ask" principle (see Recipe 3.3, "Removing Setters from Objects") and the fail fast principle. Moreover, it does not follow the MAPPER design rule and violates the subset principle. There is code manipulation duplicated everywhere that is needed to use these objects since it does not clearly separate the "what" from the "how."

The industry is very lazy when it comes to creating small objects and also separating the what and the how since it takes some extra effort to discover such abstractions. It's important to look at the protocol and behavior of small components and forget trying to understand the *internals* of how things work. A bijection-compliant solution might be:

```
Port server = Port.parse(this, "www.example.org:8080");
// Port is a small object with responsibilities and protocol

Port in = server.open(this); // returns a port, not a number
URI uri = server.asUri(this); // returns an URI
InetSocketAddress address = server.asInetSocketAddress();
// returns an Address
Path path = server.path(this, "/index.html"); // returns a Path
// all of them are validated small bijection objects with very few and precise
// responsibilities
```

Related Recipes

Recipe 4.1, "Creating Small Objects"

Recipe 4.4, "Removing String Abuses"

Recipe 4.7, "Reifying String Validations"

Recipe 17.15, "Refactoring Data Clumps"

See Also

"Primitive Obsession" on Refactoring Guru

4.3 Reifying Associative Arrays

Problem

You have anemic associative *(key/value)* arrays representing real-world objects.

Solution

Use arrays for rapid prototyping and use objects for serious business.

Discussion

Rapid Prototyping

Rapid prototyping is used in product development to quickly create working prototypes to validate with the end user. This technique allows designers and engineers to test and refine a design before creating consistent, robust, and elegant clean code.

Associative arrays are a handy way to represent anemic objects. If you encounter them in the code, this recipe will help you to reify the concept and replace them. Having rich objects is beneficial to clean code so you can fail fast, maintain integrity, avoid code duplication, and gain cohesion.

Many people suffer from primitive obsession and believe this is overdesign. Designing software is about making decisions and comparing trade-offs. The performance argument is invalid nowadays since modern virtual machines can efficiently deal with small short-lived objects.

Here is an example of anemic and primitive obsession code:

```
$coordinate = array('latitude'=>1000, 'longitude'=>2000);
// They are just arrays. A bunch of raw data
```

This is more accurate according to the *bijection concept*:

```
final class GeographicCoordinate {
    function __construct($latitudeInDegrees, $longitudeInDegrees) {
        $this->longitude = $longitudeInDegrees;
        $this->latitude = $latitudeInDegrees;
    }
}

$coordinate = new GeographicCoordinate(1000, 2000);
// Should throw an error since these values don't exist on Earth
```

You need to have objects that are valid from inception:

```
final class GeographicCoordinate {
    function __construct($latitudeInDegrees, $longitudeInDegrees) {
        $this->longitude = $longitudeInDegrees;
        $this->latitude = $latitudeInDegrees;
    }
}

$coordinate = new GeographicCoordinate(1000, 2000);
// Should throw an error since these values don't exist on Earth

final class GeographicCoordinate {
    function __construct($latitudeInDegrees, $longitudeInDegrees) {
        if (!$this->isValidLatitude($latitudeInDegrees)) {
            throw new InvalidLatitudeException($latitudeInDegrees);
            }
         $this->longitude = $longitudeInDegrees;
         $this->latitude = $latitudeInDegrees;
        }
    }
}

$coordinate = new GeographicCoordinate(1000, 2000);
// throws an error since these values don't exist on Earth
```

There is an obscure small object (see Recipe 4.1, "Creating Small Objects") to model the latitude:

```
final class Latitude {
    function __construct($degrees) {
        if (!$degrees->between(-90, 90)) {
            throw new InvalidLatitudeException($degrees);
        }
    }
}

final class GeographicCoordinate {

    function distanceTo(GeographicCoordinate $coordinate) { }
    function pointInPolygon(Polygon $polygon) { }
}

// Now you are in the geometry world (and not in the world of arrays anymore).
// You can safely do many exciting things.
```

When creating objects, you must not think of them as *data*. This is a common misconception. You should stay loyal to the concept of bijection and discover real-world objects.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

4.4 Removing String Abuses

Problem

You have too many parsing, exploding, regex, string comparison, substring search, and other string manipulation functions.

Solution

Use real abstractions and real objects instead of accidental string manipulation.

Discussion

Don't abuse strings. Favor real objects. Find absent protocols to distinguish them from strings. This code does a lot of primitive string manipulations:

```
$schoolDescription = 'College of Springfield';

preg_match('/[^ ]*$/', $schoolDescription, $results);
$location = $results[0]; // $location = 'Springfield'.

$school = preg_split('/[\s,]+/', $schoolDescription, 3)[0]; //'College'
```

You can convert the code to a more declarative version:

```
class School {
    private $name;
    private $location;

    function description() {
        return $this->name . ' of ' . $this->location->name;
    }
}
```

By finding objects present in the MAPPER, your code is more declarative, more testable, and can evolve and change faster. You can also add constraints to the new abstractions. Using strings to map real objects is a primitive obsession and premature optimization symptom (see Chapter 16, "Premature Optimization"). Sometimes the strings version is a bit more performant. If you need to decide between applying this recipe and making low-level manipulations, always create real usage scenarios and find conclusive and significant improvements.

Related Recipes

Recipe 4.2, "Reifying Primitive Data"

Recipe 4.7, "Reifying String Validations"

4.5 Reifying Timestamps

Problem

Your code relies on timestamps while you just need sequencing.

Solution

Don't use timestamps for sequencing. Centralize and lock your time issuer.

Discussion

Managing timestamps across different time zones and with heavy concurrency scenarios is a well-known problem. Sometimes, you might confuse the problem of having sequential and ordered items with the (possible) solution of timestamping them. As always, you need to understand the *essential* problems to solve before guessing *accidental* implementations.

A possible solution is to use a centralized authority or some complex decentralized consensus algorithms. This recipe challenges the need for timestamps when you just need an ordered sequence. Timestamps are very popular in many languages and are ubiquitous. You need to use native timestamps just to model timestamps if you find them in the bijection.

Here are some problems with timestamps:

```
import time

# ts1 and ts2 stores the time in seconds
ts1 = time.time()
ts2 = time.time() # might be the same!!
```

Here's a better solution without timestamps since you just need sequencing behavior:

```
numbers = range(1, 100000)
# create a sequence of numbers and use them with a hotspot

# or
sequence = nextNumber()
```

Related Recipes

Recipe 17.2, "Replacing Singletons"

Recipe 18.5, "Changing Global Date Creation"

Recipe 24.3, "Changing Float Numbers to Decimals"

4.6 Reifying Subsets as Objects

Problem

You model objects in a superset domain and have lots of validation duplication.

Solutions

Create small objects and validate a restricted domain.

Discussion

Subsets are a special case of a primitive obsession smell. The subset objects are present on the bijection, therefore you must create them in your simulator. Also, when you try to create an invalid object, it should break immediately, following the fail fast principle (see Chapter 13, "Fail Fast"). Some examples of subset violations include: *emails* are a subset of *strings*, *valid ages* are a subset of *real numbers*, and *ports* are a subset of *integers*. Invisible objects have rules you need to enforce at a single point.

Take this example:

```
validDestination = "destination@example.com"
invalidDestination = "destination.example.com"
// No error is thrown
```

Here's a better domain restriction:

```
public class EmailAddress {
    public String emailAddress;

    public EmailAddress(String address) {
        string expressions = @"^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$";
        if (!Regex.IsMatch(email, expressions) {
          throw new Exception('Invalid email address');
        }
        this.emailAddress = address;
    }
}

destination = new EmailAddress("destination@example.com");
```

This solution should not be confused with the anemic Java version. You need to be loyal to the bijection of the real world.

Related Recipes

Recipe 4.2, "Reifying Primitive Data"

Recipe 25.1, "Sanitizing Inputs"

4.7 Reifying String Validations

Problem

You are validating a subset of `strings`.

Solution

Search for missing domain objects when validating strings and reify them.

Discussion

Serious software has lots of string validations. Often, they are not in the correct places, leading to fragile and corrupt software. The simple solution is to build only real-world and valid abstractions:

```
// First Example: Address Validation
class Address {
  function __construct(string $emailAddress) {
     // String validation on Address class violates
     // Single Responsibility Principle
     $this->validateEmail($emailAddress);
     // ...
   }

  private function validateEmail(string $emailAddress) {
    $regex = "/[a-zA-Z0-9_-.+]+@[a-zA-Z0-9-]+.[a-zA-Z]+/";
    // Regex is a sample / It might be wrong
    // Emails and Urls should be first class objects

    if (!preg_match($regex, $emailAddress))
    {
      throw new Exception('Invalid email address ' . emailAddress);
    }
  }
}

// Second Example: Wordle

class Wordle {
  function validateWord(string $wordleword) {
    // Wordle word should be a real world entity. Not a subset of Strings
  }
 }
```

Here's a better solution:

```
// First Example: Address Validation
class Address {
  function __construct(EmailAddress $emailAddress) {
     // Email is always valid / Code is cleaner and not duplicated
     // ...
   }
}

class EmailAddress {
  // You can reuse this object many times avoiding copy-pasting
  string $address;
  private function __construct(string $emailAddress) {
    $regex = "/[a-zA-Z0-9_-.+]+@[a-zA-Z0-9-]+.[a-zA-Z]+/";
    // Regex is a sample / It might be wrong
    // Emails and Urls are first class objects

    if (!preg_match($regex, $emailAddress))
    {
      throw new Exception('Invalid email address ' . emailAddress);
    }
    $this->address = $emailAddress;
  }
}

// Second Example: Wordle

class Wordle {
  function validateWord(WordleWord $wordleword) {
    // Wordle word is a real world entity. Not a subset of string
  }
 }

class WordleWord {
  function __construct(string $word) {
    // Avoid building invalid Wordle words
    // For example length != 5
  }
 }
```

Single-Responsibility Principle

The *single-responsibility principle* states that every module or class in a software system should have responsibility over a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class. In other words, a class should have only one reason to change.

The small objects are hard to find. But they follow the fail fast principle when you try to create invalid objects. The new reified object also follows the *single-responsibility principle* and the *don't repeat yourself principle*. Having these abstractions forces you to implement specific behavior that is already available in the objects it encapsulates. For example, a `WordleWord` is not a `String`, but you might need some functions.

Don't Repeat Yourself Principle

The *don't repeat yourself* (DRY) principle states that software systems should avoid redundancy and repetition of code. The goal of the DRY principle is to improve the maintainability, flexibility, and understandability of software by reducing the amount of duplicated knowledge, code, and information.

A counterargument about efficiency avoiding these new indirections is a sign of *premature optimization* unless you have concrete evidence of a substantial penalty with real-use scenarios from your customers. Creating these new small concepts keeps the model loyal to the bijection and ensures your models are always healthy.

SOLID Principles

SOLID is a mnemonic that stands for five principles of object-oriented programming. They were defined by Robert Martin and are guidelines and heuristics, not rigid rules. They are defined in the related chapters:

- Single-responsibility principle (see Recipe 4.7, "Reifying String Validations")
- Open-closed principle (see Recipe 14.3, "Reifying Boolean Variables")
- Liskov substitution principle (see Recipe 19.1, "Breaking Deep Inheritance")
- Interface segregation principle (see Recipe 11.9, "Breaking Fat Interfaces")
- Dependency inversion principle (see Recipe 12.4, "Removing One-Use Interfaces")

Related Recipes

Recipe 4.4, "Removing String Abuses"

Recipe 6.10, "Documenting Regular Expressions"

4.8 Removing Unnecessary Properties

Problem

You have objects created based on their properties instead of their behavior.

Solution

Remove accidental properties. Add the needed behavior and then add accidental properties to support the defined behavior.

Discussion

Many programming schools tell you to quickly identify object parts and then build functions around them. Such models are usually coupled and less maintainable than the ones created based on the desired behavior. Following *YAGNI*'s premise (see Chapter 12, "YAGNI"), you'll find many times you don't need these attributes.

Whenever they want to model a person or an employee, junior programmers or students add an attribute id or name without thinking if they are really going to need them. You need to add attributes "on demand" when there's enough behavioral evidence. Objects are not "data holders."

This is a classic teaching example:

```
class PersonInQueue
  attr_accessor :name, :job

  def initialize(name, job)
    @name = name
    @job = job
  end
end
```

If you start focusing on the behavior, you will be able to build better models:

```
class PersonInQueue

  def moveForwardOnePosition
    # implement protocol
  end
end
```

An amazing technique for behavior discovery is *test-driven development*, where you are forced to start iterating the behavior and protocol and deferring accidental implementation as much as you can.

Test-Driven Development

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first, the developer writes a failing automated test case that defines a desired improvement or new behavior, then produces minimal production code to pass that test and finally refactors the new code to acceptable standards. One of the main goals of TDD is to make the code easier to maintain by ensuring that it is well-structured and follows good design principles. It also helps to catch defects early in the development process, since each new piece of code is tested as soon as it is written.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.5, "Removing Automatic Properties"

Recipe 3.6, "Removing DTOs"

Recipe 17.17, "Converting Fungible Objects"

4.9 Creating Date Intervals

Problem

You have to model real-world intervals and you have information like "*from* date" and "*to* date," but no invariants like: "*from* date should be lower than *to* date."

Solution

Reify this small object and honor the MAPPER rule.

Discussion

This recipe presents a very common abstraction that you might miss and has the same problems you saw in this chapter's other recipes: missing abstractions, duplicated code, unenforced invariant (see Recipe 13.2, "Enforcing Preconditions"), primitive obsession, and violation of the fail fast principle. The restriction "from date should be lower than to date" means that the starting date of a certain interval should occur before the ending date of the same interval.

The "*from* date" should be a date that comes earlier in time than the "*to* date." This restriction is in place to ensure that the interval being defined makes logical sense and that the dates used to define it are in the correct order. You know it but forget to create the `Interval` object. Would you create a `Date` as a pair of three integer numbers? Certainly not.

Here is an anemic example:

```
val from = LocalDate.of(2018, 12, 9)
val to = LocalDate.of(2022, 12, 22)

val elapsed = elapsedDays(from, to)

fun elapsedDays(fromDate: LocalDate, toDate: LocalDate): Long {
    return ChronoUnit.DAYS.between(fromDate, toDate)
}

// You need to apply this short function
// or the inline version many times in your code
// You don't check fromDate to be less than toDate
// You can make accounting numbers with a negative value
```

After you reify the `Interval` object:

```
data class Interval(val fromDate: LocalDate, val toDate: LocalDate) {
    init {
        if (fromDate >= toDate) {
            throw IllegalArgumentException("From date must be before to date")
        }
        // Of course the Interval must be immutable
        // By using the keyword 'data'
    }

    fun elapsedDays(): Long {
        return ChronoUnit.DAYS.between(fromDate, toDate)
    }
}

val from = LocalDate.of(2018, 12, 9)
val to = LocalDate.of(2002, 12, 22)

val interval = Interval(from, to) // Invalid
```

This is a primitive obsession smell and is related to how you model things. If you find software with missing simple validations, it certainly needs some reification.

Related Recipes

Recipe 4.1, "Creating Small Objects"

Recipe 4.2, "Reifying Primitive Data"

Recipe 10.1, "Removing Repeated Code"

CHAPTER 5

Mutability

No person ever steps in the same river twice. For it's not the same river and he's not the same person.

—Heraclitus

5.0 Introduction

Since the beginning of the stored-program concept, you have learned that software is programs plus data. It is clear that without data there is no software. In object-oriented programming you build models that evolve over time, emulating the knowledge you learn by observing the reality you are representing. However, you manipulate and sometimes abuse those changes uncontrollably, violating the only important design principle by generating incomplete (and therefore invalid) representations and propagating the ripple effect with your changes.

In the functional paradigm, this is elegantly addressed by forbidding mutations. You can be (a little) less drastic. Being true to the bijection in your computable model as defined in Chapter 2, you should be able to distinguish when an object changes in terms of the *accidental* and forbid all *essential* changes (because they would violate the bijection principle).

Immutability is a strict property in functional programming, and many object-oriented languages are developing tools to favor it. Nevertheless, many of them have leftovers in the core classes like `Date` or `String`. Objects should know how to defend themselves against invalid representations. They are the powers against mutants.

Let's review the `Date` class in the most widely used languages in today's industry:

Go
: `Date` is a struct.

Java
: Mutable (deprecated).

PHP
: Mutable with setters abuse.

Python
: Mutable (all attributes are public in Python).

JavaScript
: Mutable with setters abuse.

Swift
: Mutable.

The representation of the time domain is probably one of the oldest and best challenges known to humanity. The fact that these *getters* are being deprecated according to the official documentation of some of these languages speaks to poor initial design in most modern languages.

A Different Approach to Mutability

A possible attack is to reverse the burden of proof. Objects are completely immutable unless otherwise stated. If they evolve, they must always do so in their *accidental* aspects. Never in their *essence*. This change should not be coupled with all the other objects that use it (see Recipe 3.2, "Identifying the Essence of Your Objects").

If an object has been complete since its creation, it will always be able to perform functions. An object must correctly represent the entity from its inception. If you work in a concurrent environment, it is essential that the objects are always valid. An object must be immutable if the entity it represents is immutable, and most real-world entities are immutable. The immutability property is part of the bijection.

These rules keep the model consistent with its representation. As a corollary of the demonstration, you can derive a series of rules:

- Corollary 1: Objects must be complete from their creation (Recipe 5.3, "Forbidding Changes in the Essence").
- Corollary 2: Setters should not exist (Recipe 3.3, "Removing Setters from Objects").

- Corollary 3: Getters should not exist (unless they exist in the real world and then the bijection is valid). It is not the responsibility of any real entity to reply to a *`getXXX()`* message since the `get()` responsibility is not part of any object behavior (Recipe 3.8, "Removing Getters").

5.1 Changing var to const

Problem

You have constant variables declared with `var`.

Solution

Choose wisely your variable names, scope, and mutability.

Discussion

Many languages support the concept of variables and constants. Defining a correct scope is key to following the fail fast principle. Most of them don't need variable declarations and some other languages allow you to state mutability, but you should be strict and explicit with your declarations and declare all variables `const` unless you need to change them.

You can see here how reassigning a variable is not raised as a problem where it should be:

```
var pi = 3.14
var universeAgeInYears = 13.800.000.000

pi = 3.1415 // no error
universeAgeInYears = 13.800.000.001 // no error
```

A correct approach would be to define them as `const`:

```
const pi = 3.14 // Value cannot mutate or change
let universeAgeInYears = 13.800.000.000 // Value can change

pi = 3.1415 // error. cannot define
universeAgeInYears = 13.800.000.001 // no error
```

With *mutation testing* by forcing a `const` declaration, you can check if a value remains constant and be more declarative by explicitly enforcing it. Some languages have conventions where constants are defined in uppercase. You should follow them since readability is always very important and you need to explicitly state your intentions and usages.

Mutation Testing

Mutation testing is a technique that you can use to value the quality of your unit tests. It involves introducing small, controlled changes (called "mutations") to the code you are testing and checking whether your existing unit tests can detect those changes. It can help you identify areas of the code where you need additional tests, and you can use it as a measure of the quality of your existing tests.

A mutation consists of changing a small part of the code (for example, negating a boolean, replacing an arithmetic operation, replacing a value to null, etc.) and seeing if any test fails.

Related Recipes

Recipe 5.2, "Declaring Variables to Be Variable"

Recipe 5.4, "Avoiding Mutable const Arrays"

5.2 Declaring Variables to Be Variable

Problem

You assign a value to a variable and use it, but never change it.

Solution

Use mutability selectors if your programming language allows it.

Discussion

You must honor the bijection mutability, change the variable to a constant, and be clear on its scope. You are always learning from the domain, and sometimes you might guess that a value can change with the MAPPER, as defined in Chapter 2. Later on, you learn it won't change and therefore, you need to promote it to a constant. This avoids magic constants (see Recipe 6.8, "Replacing Magic Numbers with Constants").

In the following code example, you can see a password defined as a variable where it will never change:

```
function configureUser() {
  $password = '123456';
  // Setting a password on a variable is a vulnerability
  $user = new User($password);
  // Notice variable doesn't change
}
```

You can apply the recipe and declare the value to be constant:

```
define("USER_PASSWORD", '123456')

function configureUser() {
  $user = new User(USER_PASSWORD);
}

// or

function configureUser() {
  $user = new User(userPassword());
}

function userPassword() : string {
  return '123456';
}
```

Software Linters

A *software linter* automatically checks source code for previously defined issues. The goal of a linter is to help you catch mistakes early in the development process before they become more difficult and costly to fix. You can configure your linter to check for a wide range of issues, including coding style, naming conventions, and security vulnerabilities. You can use most linters as plug-ins in your IDE, and they can also add value as steps in the continuous integration/continuous deployment pipeline. You can also achieve the same results with many generative machine learning tools like ChatGPT, Bard, and many others.

Many code linters can help you to check if the variable has just one assignment, and you can also perform mutation testing (see Recipe 5.1, "Changing var to const") and try to modify the variable to see if the automated tests break. You must challenge yourself and refactor when the variable scope is clear and you learn more about its properties and mutability.

Continuous Integration and Continuous Deployment

The *CI/CD* (continuous integration and continuous deployment) pipeline automates the process of software development, testing, and deployment. The pipeline is designed to streamline the software development process, automate tasks, improve code quality, and make it faster and more managed to deploy new features and fixes in several different environments.

Related Recipes

Recipe 5.1, "Changing var to const"

Recipe 5.6, "Freezing Mutable Constants"

Recipe 6.1, "Narrowing Reused Variables"

Recipe 6.8, "Replacing Magic Numbers with Constants"

5.3 Forbidding Changes in the Essence

Problem

Your objects mutate their essence.

Solution

Forbid changing essential attributes once they are set.

Discussion

As you have seen in Recipe 3.2, "Identifying the Essence of Your Objects", you should not change an object's essence after you create it if this is not possible in the real world. Favor immutable objects to avoid the ripple effect and favor referential transparency (see Recipe 5.7, "Removing Side Effects"). Objects should only mutate in accidental ways, not in essential ones.

Object Essence

Describing the essence of an object can be challenging as it requires a deep understanding of the domain in which it belongs. If you can remove a behavior and the object continues to perform the same, then the removed behavior is not essential. Since properties are coupled to behavior, they will follow the same rule. A car can change its color and will be the same car, but it would be difficult to change the model, serial number, etc. This is very subjective though because the real world is also subjective, and this is how engineering works. This is an engineering process and not a scientific one.

If you recall the `Date` metaphor:

```
const date = new Date();
date.setMonth(4);
```

The reference to the date object is constant and will always point to the same date. The reference cannot change, but the date object can, using any method changing its internal state, in this example, setMonth(). You need to remove all the setters changing essential attributes (see Recipe 3.3, "Removing Setters from Objects"):

```
class Date {
//  setMonth(month) {
//      this.month = month;
//  }
// REMOVED
}
```

From now on, the date cannot change and all references to it will remain bound to the original mapped date.

Removing the setters on production code can create defects. You can make small refactors that adjust object creation and run your automated tests.

Related Recipes

Recipe 17.11, "Avoiding the Ripple Effect"

5.4 Avoiding Mutable const Arrays

Problem

You declare an array as const but it can mutate.

Solution

Be very careful with language mutability directives and understand their scope.

Discussion

Some languages declare references to be constant, but this is not equivalent to immutability. You can use the spread operator in JavaScript instead:

```
const array = [1, 2];

array.push(3)

// array => [1, 2, 3]
// Wasn't it constant ?
// constant != immutable ?
```

The variable `array` is defined as a constant, meaning that its reference cannot be reassigned to a different value. However, when an array or an object is assigned to a constant variable, you can still modify the properties of the object or elements of the array, because the constant variable holds the reference to the object or array in memory, but not the array or object itself. Therefore, when you call `array.push(3)`, you are modifying the array that the constant variable `array` references, but not reassigning the variable itself.

Spread Operator

The *spread operator* in JavaScript is represented by three dots (...). It allows an iterable (such as an array or string) to be expanded in places where zero or more elements (or characters) are expected. For example, you can use it to merge arrays, copy arrays, insert elements into an array, or spread properties of an object.

Here's a more declarative example:

```
const array = [1, 2];

const newArray = [...array, 3]
// array => [1, 2] Didn't mutate
// newArray = [1, 2, 3]
```

The spread operator creates a shallow copy of the original array, so the two arrays are different and independent of each other. You should always favor immutability in your designs and take extra care with side effects since this is a "language feature."

Shallow Copy

A *shallow copy* is a copy of an object that creates a new reference to the same memory location where the original object is stored. Both the original object and its shallow copy share the same values. The changes you make to the values of one will be reflected in the other. Instead, a deep copy creates a completely independent copy of the original object, with its own properties and values. Any changes you make to the properties or values of the original object will not affect the deep copy, and vice versa.

Related Recipes

Recipe 5.1, "Changing var to const"

Recipe 5.6, "Freezing Mutable Constants"

See Also

"Spread Syntax (...)" on Mozilla.org

5.5 Removing Lazy Initialization

Problem

You use lazy initialization to retrieve expensive objects when you need them and not before.

Solution

Do not use lazy initialization. Use an object provider instead.

Discussion

Lazy Initialization

Using lazy initialization you delay the creation of an object or calculation of a value until it is actually needed, rather than making it immediately. This is typically used to optimize resource usage and improve performance by deferring the initialization process until the last possible moment.

Lazy initialization has several problems like concurrency issues and race conditions if multiple threads try to access and initialize the object at the same time. It also makes code more complex, as it is a classic example of premature optimization (See Chapter 16, "Premature Optimization"). On some occasions, this can lead to situations where one thread is waiting for another thread to initialize an object, which can cause a deadlock if the latter thread is also waiting for the first thread to initialize a different object.

Here's a very simple example using Python's built-in lazy initialization:

```
class Employee
  def emails
    @emails ||= []
  end

  def voice_mails
    @voice_mails ||= []
  end
end
```

In Python, you can use a property or descriptor to delay the creation of the resource until you access it for the first time. The `emails` method uses the ||= operator, which is a shorthand for "or equals." It assigns a value to a variable only if it is nil or false. In this case, it assigns an empty array `[]` to the instance variable `@emails` if it is nil.

Here is the same example with a language not explicitly supporting lazy initialization:

```
class Employee {
  constructor() {
    this.emails = null;
    this.voiceMails = null;
  }

  getEmails() {
    if (!this.emails) {
      this.emails = [];
    }
    return this.emails;
  }

  getVoiceMails() {
    if (!this.voiceMails) {
      this.voiceMails = [];
    }
    return this.voiceMails;
  }
}
```

You should remove the lazy initialization mechanism completely and initialize the essential attributes upon building as we have seen in other recipes:

```
class Employee
  attr_reader :emails, :voice_mails

  def initialize
    @emails = []
    @voice_mails = []
  end
end
# You can also inject a design pattern to externally deal
# with voice_mails so you can mock it in your tests
```

Lazy initialization is a common pattern you can use to check for a noninitialized variable. But you must avoid premature optimizations. If you have real performance problems, you should use a *proxy pattern*, *facade pattern*, or a more independent solution instead of *singletons*. Singleton is another antipattern often combined with lazy initialization (see Recipe 17.2, "Replacing Singletons").

Antipatterns

A software *antipattern* is a design pattern that may initially seem to be a good idea, but ultimately leads to negative consequences. They were originally presented as good solutions by many experts, but nowadays there's strong evidence against their usage.

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 17.2, "Replacing Singletons"

5.6 Freezing Mutable Constants

Problem

You declare something as a constant by using the `const` keyword. But you can mutate parts of it.

Solution

Use immutable constants.

Discussion

You likely learned to declare constants in your first course on computer programming. As always, it is not important that something is constant. The important thing is that it does not mutate. This recipe varies in different programming languages. JavaScript is renowned for not following the principle of least surprise. Therefore, the following behavior is not surprising at all:

```
const DISCOUNT_PLATINUM = 0.1;
const DISCOUNT_GOLD = 0.05;
const DISCOUNT_SILVER = 0.02;

// Since variables are constants you cannot reassign them
const DISCOUNT_PLATINUM = 0.05; // Error

// You can group them
const ALL_CONSTANTS = {
  DISCOUNT: {
    PLATINUM = 0.1;
    GOLD = 0.05;
    SILVER = 0.02;
  },
};
```

```
const ALL_CONSTANTS = 3.14; // Error

ALL_CONSTANTS.DISCOUNT.PLATINUM = 0.08; // NOT AN ERROR. OOPS!

const ALL_CONSTANTS = Object.freeze({
  DISCOUNT:
    PLATINUM = 0.1;
    GOLD = 0.05;
    SILVER = 0.02;
});

const ALL_CONSTANTS = 3.14; // Error

ALL_CONSTANTS.DISCOUNT.PLATINUM = 0.12; // NOT AN ERROR. OOPS!
```

You need to be cautious with the constants inside the constants:

```
export const ALL_CONSTANTS = Object.freeze({
  DISCOUNT: Object.freeze({
    PLATINUM = 0.1;
    GOLD = 0.05;
    SILVER = 0.02;
  }),
});

const ALL_CONSTANTS = 3.14; // Error

ALL_CONSTANTS.DISCOUNT.PLATINUM = 0.12; // ERROR
// Code works, but it is coupled and you cannot test it

Class TaxesProvider {
  applyPlatinum(product);
}

// Now you can couple to an interface (the protocol of taxes provider)
// Since class has no setters it is constant and immutable
// And you can replace it on tests
```

This tricky behavior happens in just a few languages like JavaScript. You can perform mutation testing (see Recipe 5.1, "Changing var to const") to find changed values as in the previous recipe, and you need to enforce mutability with the right tools.

The Principle of Least Surprise

The *principle of least surprise* or *principle of least astonishment* says that a system must behave in ways that are the least surprising to its users and consistent with the users' expectations. If you follow this principle, the user can easily predict what will happen when they interact with the system. As a developer, you should create more intuitive and easier-to-use software, leading to increased user satisfaction and productivity.

Related Recipes

Recipe 5.4, “Avoiding Mutable const Arrays”

Recipe 6.1, “Narrowing Reused Variables”

Recipe 6.8, “Replacing Magic Numbers with Constants”

5.7 Removing Side Effects

Problem

You have functions executing side effects.

Solution

Avoid side effects.

Discussion

Side effects bring coupling, unexpected results, and violate the principle of least surprise (see Recipe 5.6, “Freezing Mutable Constants”). They can also generate conflicts in a multiprocessing environment. You can favor referential transparency by carefully interacting only with yourself and your arguments.

Referential Transparency

Referential transparency functions always produce the same output for a given input and do not have any side effects, such as modifying global variables or performing I/O operations. In other words, a function or expression is referentially transparent if it can be replaced with its evaluated result without changing the behavior of the program. This is a fundamental concept in functional programming paradigms, where functions are treated as mathematical expressions that map inputs to outputs.

Here you can see a function affecting both a global variable and an external resource:

```
let counter = 0;

function incrementCounter(value: number): void {
  // Two side effects
  counter += value; // it modifies the global variable counter
  console.log(`Counter is now ${counter}`); // it logs a message to the console
}
```

By avoiding all side effects your function is reentrant and predictable:

```
let counter = 0;

function incrementCounter(counter: number, value: number): number {
  return counter + value; // Not too efficient
}
```

Most linters can warn you when something is accessing the global state or functions and creating side effects. Functional programming is amazing and can teach a lot about how to write clean code.

Related Recipes

Recipe 18.1, "Reifying Global Functions"

5.8 Preventing Hoisting

Problem

You use variables before their declaration.

Solution

Declare your variables and watch the scope.

Discussion

Hoisting damages readability and violates the principle of least surprise. You should always be explicit with your variable declarations, use `const` declarations when possible (see Recipe 5.1, "Changing var to const"), and declare variables at the beginning of the scope. Hoisting allows variable declarations to be moved to the top of their containing scope during the compilation phase. Variables declared with `var` and function declarations are "hoisted" to the top of their respective scopes automatically in several languages.

In this example you use a variable before defining it:

```
console.log(willBeDefinedLater);
// Output: undefined (but no error)

var willBeDefinedLater = "Beatriz";
console.log(willBeDefinedLater);
// Output: "Beatriz"
```

Using explicit `const` declaration:

```
const dante = "abandon hope all ye who enter here";
// Declaring a constant 'dante'
// with value "abandon hope all ye who enter here"

console.log(dante);
// Output: "abandon hope all ye who enter here"

dante = "Divine Comedy"; // Error: Assignment to constant variable
```

You can perform mutation testing to check if changing the scope of the variables brings unexpected results. Hoisting is yet another magic tool some compilers provide to favor lazy programmers. But it fights back during debugging time.

Related Recipes

Recipe 5.1, "Changing var to const"

Recipe 21.3, "Removing Warning/Strict Off"

See Also

"Hoisting" on Wikipedia

CHAPTER 6

Declarative Code

Behavior is the most important thing about software. It is what users depend on. Users like it when we add behavior (provided it is what they really wanted), but if we change or remove behavior they depend on (introduce bugs), they stop trusting us.

—Michael Feathers, *Working Effectively with Legacy Code*

6.0 Introduction

Declarative code is a type of programming code that describes *what* a program should do, rather than specifying the steps that the program should take to accomplish a task. The code focuses on the desired outcome (what), rather than the process of achieving that outcome (how). Declarative code is easier to read and understand than imperative code, which specifies the steps that a program should take to accomplish a task. The code is also more concise and focused on the ultimate outcome, rather than the specific details of how that result is achieved.

Declarative code is often used in programming languages that support functional programming, which is a programming paradigm that emphasizes the use of functions to describe the computation of a program. Examples of declarative programming languages include *SQL*, which is used for managing databases, and *HTML*, which is used to structure and format documents for the web.

Software development carries an inertia from times where you needed to write software in low-level languages due to restrictions of time and space. This is no longer the case as modern compilers and virtual machines are smarter than ever and they leave to you the important task of writing high-level, declarative, and clean code.

6.1 Narrowing Reused Variables

Problem

You are reusing the same variable in different scopes.

Solution

You shouldn't read and write the same variable for different purposes since you must try to define the minimal scope (lifetime) of all local variables.

Discussion

Reusing variables makes scopes and boundaries harder to follow and also prevents refactoring tools from extracting independent code blocks. When you program a script it is common that you reuse variables. After some cut-and-paste operations, you might get contiguous blocks. The root cause of the problem is copying code. You need to apply Recipe 10.1, "Removing Repeated Code", instead. As a rule of thumb, you should narrow the scope as much as possible as an extended scope leads to confusion and makes debugging harder.

In this code sample, you can see that the `total` variable is reused:

```
// print line total
double total = item.getPrice() * item.getQuantity();
System.out.println("Line total: " + total);

// print amount total
total = order.getTotal() - order.getDiscount();
System.out.println( "Amount due: " + total );

// 'total' variable is reused
```

You should narrow the scope of the variable and break it into two different blocks. You can accomplish that if you extract them using Recipe 10.7, "Extracting a Method to an Object":

```
function printLineTotal() {
  double lineTotal = item.getPrice() * item.getQuantity();
  System.out.println("Line total: " + lineTotal);
}

function printAmountTotal() {
  double amountTotal = order.getTotal() - order.getDiscount();
  System.out.println("Amount due: " + amountTotal);
}
```

As a general rule, you should avoid reusing variable names. Use more local, specific, and intention-revealing names.

Related Recipes

Recipe 10.1, "Removing Repeated Code"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 11.1, "Breaking Too Long Methods"

Recipe 11.3, "Reducing Excess Variables"

Intention-Revealing

Intention-revealing code clearly communicates its purpose or intention to other developers who may read or work with your code in the future. The goal of intention-revealing code is to make it more behavioral, declarative, readable, understandable, and maintainable.

6.2 Removing Empty Lines

Problem

You have big chunks of code separated by empty lines.

Solution

Use Recipe 10.7, "Extracting a Method to an Object", to break the blocks of behavior identified by the blank lines.

Discussion

Shorter functions favor readability, increment reuse, and follow the KISS principle. Here is an example where you can group contiguous chunks of code separated by blank lines:

```
function translateFile() {
    $this->buildFilename();
    $this->readFile();
    $this->assertFileContentsOk();  // A lot more lines

    // Empty space to pause definition
    $this->translateHyperlinks();
    $this->translateMetadata();
    $this->translatePlainText();

    // Yet another empty space
    $this->generateStats();
    $this->saveFileContents();  // A lot more lines
}
```

You can change it by using Recipe 10.7, "Extracting a Method to an Object", to get a shorter version, grouping the chunks:

```
function translateFile() {
    $this->readFileToMemory();
    $this->translateContents();
    $this->generateStatsAndSaveFileContents();
}
```

If you use a linter, you can set it up to warn you when you use blank lines and also when methods are too long. Empty lines are harmless but present an opportunity to break the code into small steps. If you break your code with comments instead (or in addition to) blank lines, this is a code smell asking for a refactor (see Recipe 8.6, "Removing Comments Inside Methods").

The KISS Principle

The *KISS principle* is an abbreviation for "Keep It Simple, Stupid." It advises that systems work best when they are kept simple rather than made complicated. Simpler systems are easier to understand, use, and maintain than complex ones, and are therefore less likely to fail or produce unexpected results.

Related Recipes

Recipe 8.6, "Removing Comments Inside Methods"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 11.1, "Breaking Too Long Methods"

See Also

You can find this recipe explained in detail in Robert Martin's book *Clean Code*.

6.3 Removing Versioned Methods

Problem

You have methods named with version timestamps like `sort`, `sortOld`, `sort20210117`, `sortFirstVersion`, `workingSort`, etc.

Solution

Remove versioning from the name and use version control software instead.

Discussion

Versioned functions hurt readability and maintainability. You should keep just one working version of your artifact (class, method, attribute) and leave time control to your version control system. If you have code using versioned methods like this:

```
findMatch()
findMatch_new()
findMatch_newer()
findMatch_newest()
findMatch_version2()
findMatch_old()
findMatch_working()
findMatch_for_real()
findMatch_20200229()
findMatch_thisoneisnewer()
findMatch_themostnewestone()
findMatch_thisisit()
findMatch_thisisit_for_real()
```

You should replace all occurrences with the simpler:

```
findMatch()
```

Like many other patterns, you might create an internal policy and communicate it clearly; you can also add automatic rules to find versioned methods with patterns. Time and code evolution management is always present in software development. Luckily, nowadays you have mature tools to address this problem.

Software Source Control System

A software source control system is a tool that allows developers to track changes made to the source code of a software project. You can work with many other developers at the same time on the same codebase, favoring collaboration, rollback of changes, and management of different versions of the code. At present, Git is the most widely utilized system.

Related Recipes

Recipe 8.5, "Converting Comments to Function Names"

6.4 Removing Double Negatives

Problem

You have a method named after a negative condition and you need to ensure it is not happening.

Solution

Always name your variables, methods, and classes with positive names.

Discussion

This recipe is for readability; your brain can be misdirected when reading negative conditions. Here is an example of a double negative:

```
if (!work.isNotFinished())
```

When you turn it positive:

```
if (work.isDone())
```

You can tell your linter to check for regular expressions like `!not` or `!isNot` as a warning. You need to trust the test coverage and create safe renames and other refactors.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 14.3, "Reifying Boolean Variables"

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 24.2, "Dealing with Truthy Values"

See Also

"Remove Double Negative" on Refactoring.com

6.5 Changing Misplaced Responsibilities

Problem

You have methods in the wrong objects.

Solution

Create or overload the proper objects to find the right place using the MAPPER.

Discussion

Finding responsible objects is a tough task. You must answer the question: "Whose responsibility is…?" If you talk to anybody outside the software world, they can give you a hint on where you should place every responsibility. Software engineers, on the contrary, tend to put behavior in strange places… like helpers!

Here are some examples for the add responsibility:

```
Number>>#add: a to: b
  ^ a + b

// This is natural in many programming languages, but unnatural in real life
```

Here's a different approach:

```
Number>>#add: adder
  ^ self + adder

// This won't compile in some programming languages
// because they usually forbid changing some behavior to base classes
// But it is the right place for the 'add' responsibility
```

There are a few languages where you can add the responsibility on primitive types and, if you put the responsibilities in the proper object, you will surely find them in the same place. Here's another example defining the PI constant:

```
class GraphicEditor {
  constructor() {
    this.PI = 3.14;
    // You shouldn't define the constant here
  }

  pi() {
    return this.PI;
    // Not this object's responsibility
  }

  drawCircle(radius) {
    console.log("Drawing a circle with radius ${radius} " +
    "and circumference " + (2 * this.pi()) * radius");
  }
}
```

When you move the responsibility to a RealConstants object you avoid repeated code:

```
class GraphicEditor {
  drawCircle(radius) {
    console.log("Drawing a circle with radius " + radius +
      " and circumference " + (2 * RealConstants.pi() * radius));
  }
}
// PI's definition is RealConstants (or Number or similar) responsibility

class RealConstants {
  pi() {
     return 3.14;
  }
}
```

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 17.8, "Preventing Feature Envy"

6.6 Replacing Explicit Iterations

Problem

You may have learned loops when you were first learning code. But enumerators and iterators are the next generation and you need a higher level of abstraction.

Solution

Don't use indices while iterating. Prefer higher-level collections.

Discussion

Indices often break encapsulation and are less declarative. If your language supports it, you should favor foreach() or high-order iterators, and you can use yield(), *caches, proxies, lazy loading,* and much more when you hide your implementation details.

Here is an example with the index *i* doing a structural iteration:

```
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

The following is more declarative and high level:

```
colors.forEach((color)  => {
  console.log(color);
});

// You use closures and arrow functions
```

There are some exceptions. If the problem domain needs the elements to be bijected (as defined in Chapter 2) to natural numbers like indices, the first solution is adequate. Remember to always find real-world analogs. This kind of smell does not ring a bell to many developers because they think this is a subtlety, but building clean code is about these few declarative things that can make a difference.

Related Recipes

Recipe 7.1, "Expanding Abbreviations"

6.7 Documenting Design Decisions

Problem

You have nontrivial decisions on code and you need to document the reasons.

Solution

Use declarative and intention-revealing names.

Discussion

You must be declarative on your design or implementation decisions, for example by extracting the decision and giving it a clear intention-revealing name. You should not use code comments since comments are "dead code" and can easily become outdated, and they don't compile at all. Just be explicit about the decision or convert the comment to a method. Sometimes you can find arbitrary rules that are not so easily testable. For example, if you cannot write a failing test, you need to have a function with an excellent and declarative name instead of a comment to warn about future changes.

Here is an example of an inexplicit design decision:

```
// You need to run this process with more memory
set_memory("512k");

run_process();
```

The following is explicit and clear, giving hints on the reasons for the memory increase:

```
increase_memory_to_avoid_false_positives();
run_process();
```

Code is prose. And design decisions should be narrative.

Related Recipes

Recipe 8.5, "Converting Comments to Function Names"

Recipe 8.6, "Removing Comments Inside Methods"

6.8 Replacing Magic Numbers with Constants

Problem

You have a method that makes calculations with lots of numbers without describing their semantics.

Solution

Avoid *magic numbers* without explanation. You don't know their source and you should be very afraid of changing them and breaking code.

Discussion

Magic numbers are a source of coupling. They are hard to test and hard to read. You should rename every constant with a semantic name (meaningful and intention-revealing) and replace them with parameters, so you can mock them (see Recipe 20.4, "Replacing Mocks with Real Objects") from the outside. A constant definition is often a different object than the constant user and luckily, many linters can detect number literals in attributes and methods.

Here is a well-known constant:

```
function energy($mass) {
    return $mass * (299792 ** 2);
}
```

If you rewrite the example you can get:

```
function energy($mass) {
    return $mass * (LIGHT_SPEED_KILOMETERS_OVER_SECONDS ** 2);
}
```

Related Recipes

Recipe 5.2, "Declaring Variables to Be Variable"

Recipe 5.6, "Freezing Mutable Constants"

Recipe 10.4, "Removing Cleverness from Code"

Recipe 11.4, "Removing Excessive Parentheses"

Recipe 17.1, "Making Hidden Assumptions Explicit"

Recipe 17.3, "Breaking God Objects"

6.9 Separating "What" and "How"

Problem

You have code looking at the internal gears of the clock instead of watching the clock hands.

Solution

Don't mess with implementation details. Be declarative, not imperative.

Discussion

Choosing names is important to avoid accidental coupling. Separating concerns can be a challenging task in the software industry, but functional software has the ability to withstand the test of time. On the contrary, implementational software brings coupling and is harder to change.

Sometimes, people document the changes using comments, but this is not a good solution since comments are seldom maintained (see Recipe 8.5, "Converting Comments to Function Names"). If you favor design for change and intentions your code will survive longer and function better.

In this code sample the move action is coupled to the pending task on `stepWork`:

```
class Workflow {
    moveToNextTransition() {
        // You couple the business rule with the accidental implementation
        if (this.stepWork.hasPendingTasks()) {
            throw new Error('Preconditions are not met yet..');
        } else {
            this.moveToNextStep();
        }
    }
}
```

Here's a better solution using this recipe:

```
class Workflow {
    moveToNextTransition() {
        if (this.canMoveOn()) {
            this.moveToNextStep();
        } else {
            throw new Error('Preconditions are not met yet..');
        }
    }

    canMoveOn() {
        // You hide accidental implementation 'the how'
        // under the 'what'
        return !this.stepWork.hasPendingTasks();
    }
}
```

You need to choose good names and add indirection layers when necessary to avoid premature optimization (see Chapter 16, "Premature Optimization"). The argument that you are wasting computational resources and you need to know the insights is not important. Besides, any modern virtual machine can cache or inline these additional invocation calls.

Related Recipes

Recipe 8.5, “Converting Comments to Function Names”

Recipe 19.6, “Renaming Isolated Classes”

6.10 Documenting Regular Expressions

Problem

You have magic regular expressions that are hard to understand.

Solution

Break your complex regular expressions into shorter and more declarative examples.

Discussion

Regular expressions hurt readability and prevent maintainability and testability; you should only use them for string validation. If you need to manipulate objects, don’t make them strings: create small objects using Recipe 4.1, “Creating Small Objects”.

Here is an example of a regular expression that is not declarative:

```
val regex = Regex("^\\+(?:[0-9][- -]?){6,14}[0-9a-zA-Z]$")
```

This is more declarative and easier to understand and debug:

```
val prefix = "\\+"
val digit = "[0-9]"
val space = "[- -]"
val phoneRegex = Regex("^$prefix(?:$digit$space?){6,14}$digit$")
```

Regular expressions are a valid tool. There are not many automated ways of checking for possible abusers. An allow list might be of help. They are also a great tool for string validation. You must use them in a declarative way and just for strings. Good names are very important to understand pattern meanings. If you need to manipulate objects or hierarchies, you should do it with objects unless you have a conclusive benchmark of *impressive* performance improvement.

Related Recipes

Recipe 4.7, “Reifying String Validations”

Recipe 10.4, “Removing Cleverness from Code”

Recipe 16.2, “Removing Premature Optimization”

Recipe 25.4, “Replacing Evil Regular Expressions”

6.11 Rewriting Yoda Conditions

Problem

You are testing expected values on the left part of your expression.

Solution

Write your conditions with the variable value on the left and the value to be tested on the right.

Discussion

Most programmers write the variable or condition first and the test value second. In fact, this is the correct order for assertions. In some languages, this style preference is used to avoid accidental assignment instead of equality comparison, which can result in a logic error in the code.

Here's an example of a Yoda condition:

```
if (42 == answerToLifeMeaning) {
  // prevents the accidental assignation typo
  // since '42 = answerToLifeMeaning' is invalid
  // but 'answerToLifeMeaning = 42' is valid
}
```

Here's what it should look like after you rewrite it:

```
if (answerToLifeMeaning == 42) {
  // might be mistaken with answerToLifeMeaning = 42
}
```

Always check for constant values on the left side of the comparison.

Related Recipes

Recipe 7.15, "Renaming Arguments According to Role"

6.12 Removing Comedian Methods

Problem

You have code or examples that might offend people.

Solution

Don't be informal or offensive. Be kind to your code and your readers.

Discussion

You need to write code in a professional way using meaningful names. Your profession has a creative side. Sometimes you might get bored and try to be funny, hurting the readability of your code and your reputation. Here's some nonprofessional code:

```
function erradicateAndMurderAllCustomers();
// unprofessional and offensive
```

A more professional version of the method:

```
function deleteAllCustomers();
// more declarative and professional
```

You can have a list of forbidden and profanity words and check them automatically or in code reviews. Naming conventions should be generic and should not include cultural jargon. You should write production code in a way that ensures future software developers (even your future self) will easily understand it.

Related Recipes

Recipe 7.7, "Renaming Abstract Names"

6.13 Avoiding Callback Hell

Problem

You have asynchronous code using callbacks that are excessively nested and difficult to read and maintain.

Solution

Don't process calls using callbacks. Write a sequence.

Discussion

You have callback hell when your code has nested multiple callbacks, resulting in a complex and difficult-to-read code structure. This is often seen in JavaScript when using asynchronous programming, where a callback function is passed as an argument to another function. The deep nesting generates code also called the Pyramid of Doom.

When you call the inner function, it may return a function that takes a callback, leading to a chain of nested callbacks that can quickly become difficult to follow and reason about.

This is a short example of callback hell:

```
asyncFunc1(function (error, result1) {
  if (error) {
      console.log(error);
  } else {
      asyncFunc2(function (error, result2) {
        if (error) {
           console.log(error);
        } else {
        asyncFunc3(function (error, result3) {
           if (error) {
             console.log(error);
           } else {
           // Nested callback continues...
         }
        });
      }
   });
  }
});
```

You can rewrite it this way:

```
function asyncFunc1() {
  return new Promise((resolve, reject) => {
      // Async operation
      // ...

      // If successful
      resolve(result1);

      // If error
      reject(error);
  });
}

function asyncFunc2() {
  return new Promise((resolve, reject) => {
      // Async operation
      // ...

      // If successful
      resolve(result2);

      // If error
      reject(error);
   });
}

async function performAsyncOperations() {
  try {
      const result1 = await asyncFunc1();
```

```
        const result2 = await asyncFunc2();
        const result3 = await asyncFunc3();

        // Continue with further operations
    } catch (error) {
        console.log(error);
    }
}

performAsyncOperations();
```

You can use promises and async/await to solve this problem, making the code more readable and easier to debug.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 14.10, "Rewriting Nested Arrow Code"

6.14 Generating Good Error Messages

Problem

You need to create good error descriptions, both for the developers that use your code (and yourself) and for the end users.

Solution

Use meaningful descriptions and suggest corrective actions. Showing your users this kindness will go a long way.

Discussion

Programmers are seldom UX experts. Nevertheless, you should use declarative error messages thinking about the end users and show those messages with clear exit actions. You must follow the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants"), applied to your users.

Here is a bad error description:

```
alert("Cancel the appointment?", "Yes", "No");

// No consequences and actions
// The options are not clear
```

You can change this to a more declarative error:

```
alert("Cancel the appointment? \n" +
      "You will lose all the history",
      "Cancel Appointment",
      "Keep Editing");

// The consequences are clear
// The choice options have context
```

Don't mask error situations using valid domain values, and make a clear distinction between a zero and an error. Review the following code hiding a network error and incorrectly showing a balance of 0, producing panic in the end user:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        return 0
```

This version is clearer and more explicit:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        raise BlockchainNotReachableError("Error reaching blockchain")
```

Declarative Exception Descriptions

Exception descriptions should mention the business rule and not the error. Good description: "The number should be between 1 and 100." Bad description: "Number out of bounds." What are the bounds?

You need to read all exception messages in code reviews and think of the end users when raising exceptions or showing messages.

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 17.13, "Removing Business Code from the User Interface"

Recipe 22.3, "Rewriting Exceptions for Expected Cases"

Recipe 22.5, "Replacing Return Codes with Exceptions"

6.15 Avoiding Magic Corrections

Problem

You have some sentences that are valid and magical in some languages but need to be more explicit and honor the fail fast principle.

Solution

Remove magic corrections from your code.

Discussion

Some languages hide problems under the rug and make magic corrections and obscure castings, violating the fail fast principle. You should be explicit and remove all ambiguity. Change this kind of magic sentence:

```
new Date(31, 02, 2020);

1 + 'Hello';

!3;

// This is valid in most languages
```

Here's an explicit solution:

```
new Date(31, 02, 2020);
// Throw an exception

1 + 'Hello';
// Type Mismatch

!3;
// Negating is a boolean operation
```

In Figure 6-1 you can see the unexpected result of adding a number to a string, which is not valid in the real world and should raise an exception.

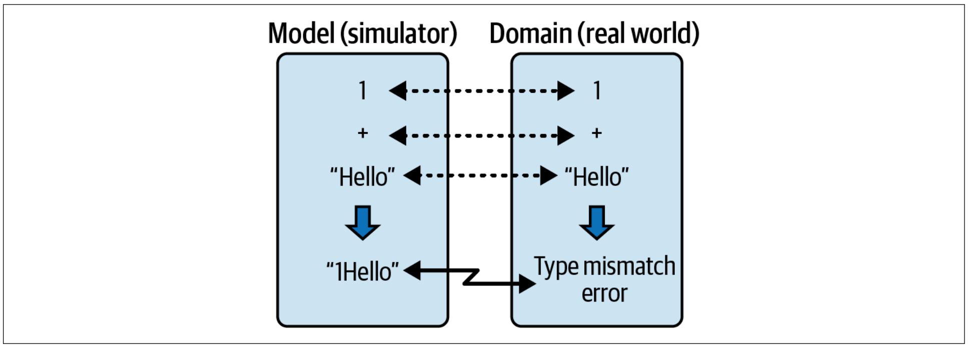


Figure 6-1. Executing the "+" method yields different results in the model and real world

Many of these problems are encouraged by languages themselves. You should be very declarative and explicit; don't abuse accidental solutions in languages, especially if they seem magic (as opposed to rational). Many programmers pretend to be smart by exploiting language features; this is unnecessarily complex code that tries to be clever, contrary to clean code.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 24.2, "Dealing with Truthy Values"

CHAPTER 7

Naming

There are only two hard things in Computer Science: cache invalidation and naming things.
—Phil Karlton

7.0 Introduction

Naming is a critical aspect of software development. It directly impacts the readability, understandability, and maintainability of code. You need to choose good names for objects, classes, variables, functions, etc. Good names help reduce confusion and errors and make it easier for other developers to use, modify, and debug code. Names are irrelevant for compilers and interpreters. But writing code is a human-centric activity. Poor naming choices can lead to confusion, misunderstandings, and defects. If a name is too generic or unclear, it may not accurately reflect the purpose or behavior of the code element it represents and makes it more difficult for other developers to understand how to use or modify it, leading to errors and wasted time.

7.1 Expanding Abbreviations

Problem

You have ambiguous abbreviated names.

Solution

Use strong, sufficiently long, unambiguous, and descriptive names.

Discussion

Bad naming is a problem programmers face in most software projects, and abbreviations are contextual and ambiguous. In the past, programmers used short names because memory was scarce but now you seldom face this problem. You should avoid abbreviations in all contexts: variables, functions, modules, packages, namespaces, classes, etc.

Review the following example in standard Golang naming conventions:

```
package main

import "fmt"
type YVC struct {
    id int
}

func main() {
    fmt.Println("Hello, World")
}
```

This kind of premature optimization (see Chapter 16, "Premature Optimization") of the text hurts readability and maintainability. In some languages, this bad practice is rooted and you cannot change it. If you have the freedom to do it, you should change the preceding code to this improved example:

```
package main

import "formatter"

type YoutTubeVideoContent struct {
    imdbMovieIdentifier int
}

function main() {
    formatter.Printline("Hello, World")
}
```

In Figure 7-1 you can see that `fmt` maps to several different concepts in the real world as many abbreviations do.

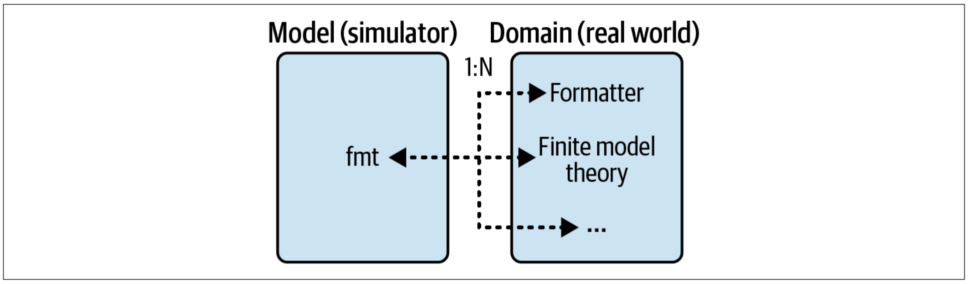


Figure 7-1. Names are abbreviated and ambiguous in the model, mapping to more than one possible concept

Computer science was born from mathematics. In math, the assignment of single-letter variables (*i*, *j*, *x*, *y*) is a good practice. The concept of reference arose from the variable and many people have wondered why mathematicians can work with such short variables, while computer scientists cannot. For mathematicians, once entered into a formula, variables lose all semantics and become indistinguishable. They have a lot of naming conventions and think very carefully about their names. The difference is that mathematicians always have a fully and formally defined local context, where one letter is enough to distinguish the variables.

This is not the case in programming, where your brain might waste a lot of energy figuring out what the meaning of an abbreviation is and even then you may sometimes be mistaken. As a reminder, write software for humans, not for compilers. An ambiguous name can have more than one meaning. For example, */usr* stands for *universal system resources,* not *user* and */dev* stands for *device,* not *development*.

Related Recipes

Recipe 7.6, "Renaming Long Names"

Recipe 10.4, "Removing Cleverness from Code"

7.2 Renaming and Breaking Helpers and Utils

Problem

You have a class with the name `Helper` and it has noncohesive and unclear behavior.

Solution

Rename the class to a more accurate name and break down the responsibilities.

Discussion

You can find helpers in many frameworks and code samples. This is another ambiguous and empty name that usually violates the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants") and the *bijection* (as defined in Chapter 2) with the real world. To address this problem, you need to find a suitable name. If the helper is a library, break all the services into different methods.

Here is an example of a helper class:

```
export default class UserHelpers {
  static getFullName(user) {
    return `${user.firstName} ${user.lastName}`;
  }

  static getCategory(userPoints) {
    return userPoints > 70 ? 'A' : 'B';
  }
}

// Notice static methods
import UserHelpers from './UserHelpers';

const alice = {
  firstName: 'Alice',
  lastName: 'Gray',
  points: 78,
};

const fullName = UserHelpers.getFullName(alice);
const category = UserHelpers.getCategory(alice);
```

You can use better names and split the responsibilities:

```
class UserScore {
  // This is an anemic class and should have a better protocol

  constructor(name, lastname, points) {
    this._name = name;
    this._lastname = lastname;
    this._points = points;
  }
  name() {
    return this._name;
  }
  lastname() {
    return this._lastname;
  }
  points() {
    return this._points;
  }
}
```

```
class FullNameFormatter {
  constructor(userscore) {
    this._userscore = userscore;

 }
  fullname() {
    return `${this._userscore.name()} ${this._userscore.lastname()}`;
  }
}

class CategoryCalculator{
  constructor(userscore1) {
     this._userscore = userscore1;
 }
  display() {
    return this._userscore.points() > 70 ? 'A' : 'B';
  }
}

let alice = new UserScore('Alice', 'Gray', 78);

const fullName = new FullNameFormatter(alice).fullname();
const category = new CategoryCalculator(alice).display();
```

Another option is to make the former helper stateless for reuse like this example:

```
class UserScore {
  // This is an anemic class and should have a better protocol

  constructor(name, lastname, points) {
    this._name = name;
    this._lastname = lastname;
    this._points = points;
  }
  name() {
    return this._name;
  }
  lastname() {
    return this._lastname;
  }
  points() {
    return this._points;
  }
}

class FullNameFormatter {
  fullname(userscore) {
    return `${userscore.name()} ${userscore.lastname()}`;
  }
}

class CategoryCalculator {
  display(userscore) {
    return userscore.points() > 70 ? 'A' : 'B';
  }
}

let alice = new UserScore('Alice', 'Gray', 78);

const fullName = new FullNameFormatter().fullname(alice);
const category = new CategoryCalculator().display(alice);
```

You can see in Figure 7-2 a `NumberHelper` that has more responsibilities and behavior than its real-world counterpart.

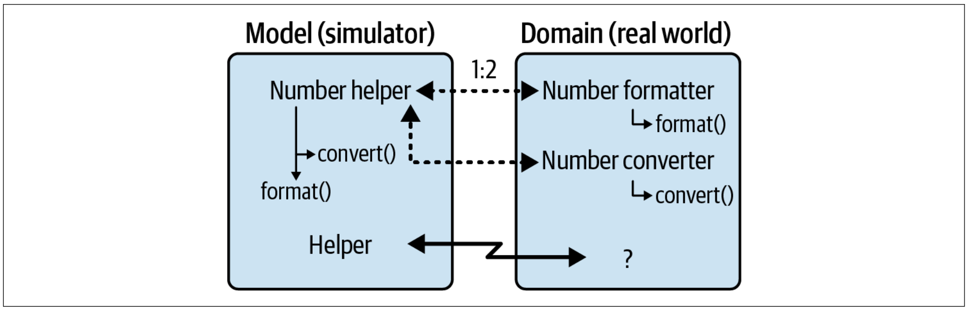


Figure 7-2. You cannot map the `NumberHelper` to a single real-world entity

Related Recipes

Recipe 6.5, "Changing Misplaced Responsibilities"

Recipe 7.7, "Renaming Abstract Names"

Recipe 11.5, "Removing Excess Methods"

Recipe 18.2, "Reifying Static Functions"

Recipe 23.2, "Reifying Anonymous Functions"

7.3 Renaming MyObjects

Problem

You have variables with the prefix "my."

Solution

Rename variables with "my" as a prefix.

Discussion

Objects prefixed with “my” lack context and violate bijection. You should change the name to a role-suggesting name. Several old tutorials use the word “my” as a lazy name. This is vague and leads to context mistakes like this example:

```
MainWindow myWindow = Application.Current.MainWindow as MainWindow;
```

Here’s an improved version with the role of a sales window:

```
MainWindow salesWindow = Application.Current.MainWindow as MainWindow;

/*
Since the window is instantiated, you are currently working
with a specialized window playing a special role
*/
```

7.4 Renaming Result Variables

Problem

You name the result of a function, method call, or computation with the ambiguous name “result.”

Solution

Always use role-suggesting good names. “Result” is always a very bad choice.

Discussion

Find the semantics of the result. If you don’t know how to name it, just name the variable with the same name as the last function call. Here you can see a result variable assigned after a method call:

```
var result;
result = lastBlockchainBlock();

// Many function calls
addBlockAfter(result);
```

Here’s a better approach with a role name:

```
var lastBlockchainBlock;
lastBlockchainBlock = findLastBlockchainBlock();

// Many function calls
// you should refactor them to minimize space
// between variable definition and usage
addBlockAfter(lastBlockchainBlock);
```

"Result" is an example of a generic and meaningless name, and a rename refactoring is cheap and safe. If you encounter this code follow the Boy Scout rule:

Boy Scout Rule

Uncle Bob's *Boy Scout rule* suggests leaving code better than you found it, just like leaving a campsite cleaner than you found it as a Boy Scout. The rule encourages developers to make small, incremental improvements to the codebase every time they touch it, instead of creating a mess of technical debt (see Chapter 21, "Technical Debt") that will be difficult to clean up later; it also favors changing things that are not completely fine. This contradicts the "If it ain't broke, don't fix it" principle.

`theResult` is a similar problem, as you can see in another example:

```
var result;
result = getSomeResult();

var theResult;
theResult = getSomeResult();
```

Applying the same recipe you get:

```
var averageSalary;
averageSalary = calculateAverageSalary();

var averageSalaryWithRaises;
averageSalaryWithRaises = calculateAverageSalary();
```

"If It Ain't Broke, Don't Fix It" Principle

The *"If it ain't broke, don't fix it"* principle is a common expression in software development that states that if a software system is working well, there is no need to make any changes or improvements to it. The principle goes back to the times when software didn't have automated tests, so making any change would probably break existing functionality. Real-world users usually tolerate defects in new features, but they become very angry when something that previously worked no longer functions as expected.

Related Recipes

Recipe 7.7, "Renaming Abstract Names"

See Also

Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin

7.5 Renaming Variables Named After Types

Problem

You have names including types, but names should always indicate role.

Solution

Remove the type since it is accidental and not present on the bijection.

Discussion

You should always design for change, hiding implementation details coupled with accidental implementation. To do so, rename your variables according to their role. See the following example where `regex` is the new instance you create:

```
public bool CheckIfPasswordIsValid(string textToCheck)
{
  Regex regex = new Regex(@"[a-z]{2,7}[1-9]{3,4}")
  var bool = regex.IsMatch(textToCheck);
  return bool;
}
```

Here's what it looks like after you give the variable `regex` a meaningful name:

```
public bool CheckIfStringHas3To7LowercaseCharsFollowedBy3or4Numbers(
  string password)
{
  Regex stringHas3To7LowercaseCharsFollowedBy3or4Numbers =
    new Regex(@"[a-z]{2,7}[1-9]{3,4}")
  var hasMatch =
    stringHas3To7LowercaseCharsFollowedBy3or4Numbers.IsMatch(password);
  return hasMatch;
}
```

Names should be long enough but not longer (review Recipe 7.6, "Renaming Long Names", up next). You can also ensure this semantic rule by instructing your linters to warn you against using names related to existing classes, types, or reserved words since they are too implementational. The first name you might come across may be related to an accidental point of view. It takes time to build a theory on the models you are building using the MAPPER, as defined in Chapter 2. Once you get there, you must rename your variables.

Related Recipes

Recipe 7.6, "Renaming Long Names"

Recipe 7.7, "Renaming Abstract Names"

Recipe 7.9, "Removing Class Names from Attributes"

7.6 Renaming Long Names

Problem

You have very long and redundant names.

Solution

Names should be long and descriptive but not too long. Shorten your names but don't use custom abbreviations.

Discussion

Long names can negatively impact readability and increase cognitive load. A rule of thumb is to use names related to the MAPPER. If abbreviations happen in the real world (for example, URL, HTTL, SSN) they are perfectly fine since they are not ambiguous when you are working on a specific domain.

See the following long and redundant name example:

```
PlanetarySystem.PlanetarySystemCentralStarCatalogEntry
```

Here's a shorter and more concise name:

```
PlanetarySystem.CentralStarCatalogEntry
```

You can train your linters to warn you about names that are too long. Remember there are no hard rules on name length, just heuristics, and heuristics are contextual.

Cognitive Load

Cognitive load is the amount of mental effort and resources required to process information and complete a task. It is the burden on a person's working memory as they try to process, understand, and remember information all at once.

Related Recipes

Recipe 7.1, "Expanding Abbreviations"

See Also

"Long and Short of Naming" by Agile Otter

7.7 Renaming Abstract Names

Problem

Your names are too abstract.

Solution

Replace abstract names with concrete ones using the real-world MAPPER.

Discussion

Names should have real-world meanings. When you are naming entities, you need to map abstract names to real-world concepts. These generalizations appear later in the process after you have modeled many concrete concepts. Usually, these domain names exist but are harder to determine. On the contrary, using pseudo-abstract names like *abstract*, *base*, *generic*, *helper*, etc. is a bad practice.

Here are some abstract examples:

```
final class MeetingsCollection {}
final class AccountsComposite {}
final class NotesArray {}
final class LogCollector {}

abstract class AbstractTransportation {}
```

Here are better, more concrete names for each one mapping to real-world concepts:

```
final class Schedule {}
final class Portfolio {}
final class NoteBook {}
final class Journal {}
final class Vehicle {}
```

You can set up your own policies and rules warning about certain words like *base, abstract, helper, manager, object,* etc. Finding names is the last thing you should do on your designs. Unless you have a clear business understanding, good names emerge at the end, after having defined behavior and protocol boundaries.

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 7.14, "Removing "Impl" Prefix/Suffix from Class Names"

Recipe 12.5, "Removing Design Pattern Abuses"

7.8 Correcting Spelling Mistakes

Problem

You have typos and spelling mistakes on your names.

Solution

Take care of your names. Use automated spell checkers.

Discussion

Readability is always important and bad spelling makes it harder to search terms in code. Note that polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic") is based on methods with the exact same name. The following example includes a typo:

```
comboFeededBySupplyer = supplyer.providers();
```

Here's what it looks like you correct it:

```
comboFedBySupplier = supplier.providers();
```

Pay close attention to your names because you will probably be the person reading your own code in a few months.

Related Recipes

Recipe 9.1, "Following Code Standards"

7.9 Removing Class Names from Attributes

Problem

You have variables that include a class name.

Solution

Don't prefix your attributes with your class name.

Discussion

Redundancy in names is a bad smell. This is a very simple recipe since you should not read attributes in isolation and names are contextual. In this code snippet, the class `Employee` has attributes prefixed with `emp`:

```
public class Employee {
   String empName = "John";
   int empId = 5;
   int empAge = 32;
}
```

After you remove them, redundancy is no longer present and the code is more compact:

```
public class Employee {
   String name;
   int id; // Ids are another smell
   int age; // Storing the age is another code smell
}
```

When the full name is included in the prefix, your linters can warn you. As always, be sure to name after the behavior, not type or data.

Related Recipes

Recipe 7.3, "Renaming MyObjects"

Recipe 7.5, "Renaming Variables Named After Types"

Recipe 7.10, "Removing the First Letter from Classes and Interfaces"

7.10 Removing the First Letter from Classes and Interfaces

Problem

You use a letter to prefix your classes indicating abstract, interface, etc.

Solution

Don't prefix or suffix your classes. Always use complete real-world concepts following the MAPPER.

Discussion

This practice is very common in some languages but hurts readability and creates cognitive load when you try to map the concept you see in the code. It also presents accidental implementation details. As always, you should name your objects after what they do, not what they are. Some languages have cultural conventions related to data types, abstract classes, or interfaces, and these names load your models with cognitive translations that are hard to follow, breaking the KISS principle (see Recipe 6.2, "Removing Empty Lines"). In C#, it's common practice to put "I" in the name of an interface because, without it, you can't tell whether it is an interface or a class.

Here's an example of the engine interface, abstract car, and implemented car:

```
public interface IEngine
{
    void Start();
}

public class ACar {}
```

If you stick to bijection names, it appears as follows:

```
public interface Engine
{
    void Start();
}

public class Vehicle {}
public class Car {}
```

With a thesaurus, you can reference unconventional terms that do not exist in reality.

Related Recipes

Recipe 7.9, "Removing Class Names from Attributes"

Recipe 7.14, "Removing "Impl" Prefix/Suffix from Class Names"

7.11 Renaming Basic / Do Functions

Problem

You have many variants of the same actions like `sort`, `doSort`, `basicSort`, `doBasicSort`, `primitiveSort`, `superBasicPrimitiveSort`.

Solution

Remove the function wrappers because they cause confusion and look like hacks.

Discussion

Using wrappers hurts readability and creates coupling between methods. It can make the real entry point hard to find. Which method would you call? If you need to wrap behavior, you can use good object wrappers like dynamic decorators.

Here's a `Calculator` class with many entry points:

```
final class Calculator {

    private $cachedResults;
```

```
        function computeSomething() {
            if (isSet($this->cachedResults)) {
                return $this->cachedResults;
            }
            $this->cachedResults = $this->logAndComputeSomething();
        }

        private function logAndComputeSomething() {
            $this->logProcessStart();
            $result = $this->basicComputeSomething();
            $this->logProcessEnd();
            return $result;
        }

        private function basicComputeSomething() {
            // Do real work here
        }
    }
```

This snippet uses objects instead of methods:

```
    final class Calculator {
        function computeSomething() {
            // Do real work here since I am the compute method
        }
    }

    // Clean and cohesive class, single responsibility

    final class CalculatorDecoratorCache {

        private $cachedResults;
        private $decorated;

        function computeSomething() {
            if (isset($this->cachedResults)) {
                return $this->cachedResults;
            }
            $this->cachedResults = $this->decorated->computeSomething();
        }
    }

    final class CalculatorDecoratorLogger {

        private $decorated;

        function computeSomething() {
            $this->logProcessStart();
            $result = $this->decorated->computeSomething();
            $this->logProcessEnd();
            return $result;
        }
    }
```

You can instruct your static linters to find wrapping methods if they follow conventions like *doXXX()*, *basicXXX()*, etc.

Decorator Pattern

The *decorator design pattern* allows you to dynamically add behavior to an individual object without affecting the behavior of other objects from the same class.

7.12 Converting Plural Classes to Singular

Problem

You have class names that use plural terms.

Solution

Rename your classes with singular terms. Classes represent concepts, and concepts are singular.

Discussion

Naming things requires additional effort, and you need to agree on certain rules across your system. Here is a small plural class name example:

```
class Users
```

You should simply rename it:

```
class User
```

Now, you can build one user, then another one, and yet another and put them all together in a collection holding users.

7.13 Removing "Collection" from Names

Problem

You have the word "collection" in your name.

Solution

Don't use "collection" in your name. It is too abstract for concrete concepts.

Discussion

Naming is very important and you need to deal a lot with collections. Collections are amazing as they don't need nulls to model absence. An empty collection is polymorphic with a full collection and you will avoid *nulls* and *ifs*. Using "collection" as part of your name hurts readability and is abstraction abuse. Look for good names in the MAPPER.

Here's a short example of a variable named `customerCollection`:

```
for (var customer in customerCollection) {
    // iterate with current customer
}

for (var currentCustomer in customerCollection) {
    // iterate with current customer
}
```

And here's a very small rename:

```
for (var customer in customers) {
    // iterate with current customer
}
```

All linters can detect bad naming like this, but they can also lead to false positives so you must be cautious. You need to care for all your clean code, variables, classes, and functions since accurate names are essential to understanding your code.

Related Recipes

Recipe 12.6, "Replacing Business Collections"

7.14 Removing "Impl" Prefix/Suffix from Class Names

Problem

You have classes with the prefix/suffix "Impl."

Solution

Name your classes after real-world concepts.

Discussion

Some languages include idioms and common usages that are contrary to good model naming, but you should pick your names carefully. It is nice to see a class implementing interfaces. It is nicer to understand what it does. Here you have an interface and a class realizing that interface:

```
public interface Address extends ChangeAware, Serializable {
    String getStreet();
}

// Wrong Name - There is no concept 'AddressImpl' in the real world
public class AddressImpl implements Address {
    private String street;
    private String houseNumber;
    private City city;
    // ..
}
```

In this simpler solution you just have the `Address`:

```
// Simple
public class Address {
    private String street;
    private String houseNumber;
    private City city;
    // ..
}

// OR
// Both are real-world names
public class Address implements ContactLocation {
    private String street;
    private String houseNumber;
    private City city;
    // ..
}
```

Remember to pick class names according to the essential bijection; don't follow the accidental implementation, and do not add "I" to your interfaces nor "Impl" to your implementations.

Related Recipes

Recipe 7.5, "Renaming Variables Named After Types"

Recipe 7.7, "Renaming Abstract Names"

7.15 Renaming Arguments According to Role

Problem

You have method arguments that lack declarative names.

Solution

Name your arguments according to the role and not the accidental position.

Discussion

When writing methods, you sometimes don't stop to find decent names and seldom refactor to adopt intention-revealing names. See the argument names in the example:

```
class Calculator:
  def subtract(self, first, second):
    return first - second
```

Applying the role suggestion rule you have clear contextual names:

```
class Calculator:
  def subtract(self, minuend, subtrahend):
    return minuend - subtrahend
```

Here's an example using the unit testing framework:

```
$this->assertEquals(one, another);
```

Instead, rename the arguments following the recipe:

```
$this->assertEquals(expectedValue, actualValue);
```

This is more important when the argument definition is far from its usage. A role name is more contextual and helpful.

Related Recipes

Recipe 7.5, "Renaming Variables Named After Types"

7.16 Removing Redundant Parameter Names

Problem

You have repeated names for your method's arguments.

Solution

Don't repeat your parameters' names. Names should be contextual and local.

Discussion

This is a code duplication problem. Your arguments should have a contextual name and shouldn't include the class you are creating. This might look like a small and innocent recipe, but it is important since these kinds of constructors present anemic properties. If you include the class name, you might be creating the objects based on properties instead of their behavior. When using names, you might miss that words are contextual and need to be read as a whole sentence. Parameters should be short and have contextual names.

Take a look at this redundant example:

```
class Employee
  def initialize(
    @employee_first_name : String,
    @employee_last_name : String,
    @employee_birthdate : Time)
  end
end
```

When you remove the repeated part from the name you get more synthetic names:

```
class Employee
  def initialize(
    @first_name : String,
    @last_name : String,
    @birthdate : Time)
  end
end
```

Related Recipes

Recipe 7.9, "Removing Class Names from Attributes"

Recipe 9.5, "Unifying Parameter Order"

7.17 Removing Gratuitous Context from Names

Problem

You prefix or suffix your classes with a global identifier.

Solution

Don't prefix or suffix your names with irrelevant information. Remove this information to honor the MAPPER and make code search easier.

Discussion

Class prefixing was a widespread practice used decades ago to claim ownership. Now you know clean names are more important. Gratuitous context refers to the unnecessary inclusion of additional information or data in code or user interfaces that do not contribute to the functionality or usability of the software.

Here is an example using a gratuitous WEBB prefix:

```
struct WEBBExoplanet {
    name: String,
    mass: f64,
    radius: f64,
    distance: f64,
    orbital_period: f64,
}

struct WEBBGalaxy {
    name: String,
    classification: String,
    distance: f64,
    age: f64,
}
```

Here's what it looks like after you remove the gratuitous prefix:

```
struct Exoplanet {
    name: String,
    mass: f64,
    radius: f64,
    distance: f64,
    orbital_period: f64,
}

struct Galaxy {
    name: String,
    classification: String,
    distance: f64,
    age: f64,
}
```

Applying this recipe is easy if you have renaming tools in your IDE. Remember that names should always be contextual.

Related Recipes

Recipe 7.9, "Removing Class Names from Attributes"

Recipe 7.10, "Removing the First Letter from Classes and Interfaces"

Recipe 7.14, "Removing "Impl" Prefix/Suffix from Class Names"

7.18 Avoiding Data Naming

Problem

Use entity domain names to model entity domain objects.

Solution

Don't name your variables "data."

Discussion

Bad naming hurts readability. You should always use role-suggesting names and find those names in the bijection. Using names with "data" favors the anemic treatment of objects, and you should think about domain-specific and role-suggesting names. Here you check if data exists:

```
if (!dataExists()) {
  return '<div>Loading Data...</div>';
}
```

And here you check if you found people:

```
if (!peopleFound()) {
  return '<div>Loading People...</div>';
}
```

You can check for this substring in your code and warn your developers. Data is everywhere if you see the world as only data. You can never see the data you manipulate. You can only infer it through behavior. You don't know the current temperature. You observe the thermometer pointing at 35 degrees. Your variables should reflect the domain and role they are fulfilling.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 7.5, "Renaming Variables Named After Types"

CHAPTER 8

Comments

The proper use of comments is to compensate for our failure to express ourself in code.
—Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*

8.0 Introduction

Back in the days of assembly language programming, the distance between what you intended to say as a programmer and how the computer worked was huge. Every few lines (sometimes on every line), you needed a little story to help you understand what the next few instructions meant. Today, comments are often a failure to choose good names. You will only need them to describe very important design decisions. They are dead code since they don't compile or run. Comments tend to diverge from the code they once described. They become floating islands of irrelevance and misdirection in the code. Clean code needs almost no comments at all. You can find some criteria on how to use them in the following recipes.

Assembly Language

Assembly language is a low-level programming language to write software programs for specific computer architectures. It is a human-readable language imperative code that is designed to be easily translated into machine language, which is the language that computers can understand.

8.1 Removing Commented Code

Problem

You have commented code.

Solution

Don't leave commented code. Use any source version control system and then safely remove the commented code.

Discussion

Before the 2000s, version control systems were uncommon and automated tests were not an established practice. To debug and test small changes, programmers used to comment on some pieces of code, but today this is a sign of sloppiness. Instead, you can utilize several tools to find previous versions and changes like `git bisect`.

Using Recipe 8.5, "Converting Comments to Function Names", you can extract the code and comment just the function call. Once you are sure you don't need it, you can remove the commented function and potentially refactor the code by removing unused methods. As a final step, after all of your tests pass, remove the comments following clean code practices.

In the following example, there are commented lines:

```
function arabicToRoman(num) {
  var decimal = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1];
  var roman = ['M', 'CM', 'D', 'CD', 'C', 'XC',
               'L', 'XL', 'X', 'IX', 'V', 'IV', 'I'];
  var result = '';

  for(var i = 0; i < decimal.length; i++) {
    // print(i)
    while(num >= decimal[i]) {
      result += roman[i];
      num -= decimal[i];
    }
  }
  // if (result > 0 return ' ' += result)

 return result;
}
```

When your tests pass, you can safely remove them:

```
function arabicToRoman(arabicNumber) {
  var decimal = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1];
  var roman = ['M', 'CM', 'D', 'CD', 'C', 'XC',
               'L', 'XL', 'X', 'IX', 'V', 'IV', 'I'];
  var romanString = '';

  for(var i = 0; i < decimal.length; i++) {
    while(arabicNumber >= decimal[i]) {
      romanString += roman[i];
      num -= decimal[i];
    }
  }

  return romanString;
}
```

Detecting when to use this recipe is hard. Some commercial linters and machine learning analyzers can detect or parse comments and guide you to remove them.

git bisect

Git is a version control system for software development. It helps you track changes to the code, collaborate with others, and revert to previous versions if necessary. Git stores the entire version history of every file. It also can manage multiple developers working on the same codebase.

`git bisect` is a command that helps you locate the commit that introduced a particular change in the code. The process starts by specifying a "good" commit that is known to not contain the defect and a "bad" commit that is known to contain the change. By iterating you can find the commit to blame and quickly locate the root cause.

Related Recipes

Recipe 8.2, "Removing Obsolete Comments"

Recipe 8.3, "Removing Logical Comments"

Recipe 8.5, "Converting Comments to Function Names"

Recipe 8.6, "Removing Comments Inside Methods"

8.2 Removing Obsolete Comments

Problem

You have comments that are no longer accurate.

Solution

Remove the obsolete comment.

Discussion

Comments add almost no value to your code and you need to restrict them only to very important design decisions. Since most people change the logic in the code and forget to update comments, they will likely become obsolete. Think carefully before adding a comment. Once it is in the codebase, it is beyond your control and can start to mislead you at any time. As Ron Jeffries says, “Code never lies; comments sometimes do.” You can remove the comment or replace it with tests (see Recipe 8.7, “Replacing Comments with Tests”).

In this code sample, the author left a comment showing some needed change:

```
void Widget::displayPlugin(Unit* unit)
{
 // TODO the Plugin will be modified soon, so I don’t implement this right now

 if (!isVisible) {
    // hide all widgets
    return;
 }
}
```

You should remove the comments and don’t leave any *ToDos* or *FixMes* (see Recipe 21.4, “Preventing and Removing ToDos and FixMes”):

```
void Widget::displayPlugin(Unit* unit)
{
  if (!isVisible) {
    return;
  }
}
```

As an exception to this recipe, you should not remove comments related to critical design decisions that are not possible to explain using this chapter’s recipes, for example, an important decision on performance, security, etc., not related to what your code actually does.

Related Recipes

Recipe 8.1, “Removing Commented Code”

Recipe 8.3, “Removing Logical Comments”

Recipe 8.5, “Converting Comments to Function Names”

Recipe 8.7, “Replacing Comments with Tests”

8.3 Removing Logical Comments

Problem

Your code has logical comments like a `true` or `false` in an `if` condition.

Solution

Don't change code semantics to skip code. Remove the dead conditions.

Discussion

Logical comments hurt readability, are not intention-revealing, and show sloppiness. You should rely on your source control system to make temporary changes. Changing code with a temporary hack is an awful developer practice because you might *forget* some and leave them forever.

The following example adds a `false` condition to avoid the `doStuff()` function for faster debugging:

```
if (cart.items() > 11 && user.isRetail())  {
  doStuff();
}
doMore();
// Production code
```

Here's what it looks like after adding a `false`:

```
// the false acts to temporary skip the if condition
if (false && cart.items() > 11 && user.isRetail())  {
  doStuff();
}
doMore();

if (true || (cart.items() > 11 && user.isRetail()))  {
// Same hack to force the condition
// code after the true is never evaluated
```

Instead of this, you should cover the two cases with different unit tests for proper debugging:

```
if (cart.items() > 11 && user.isRetail())  {
  doStuff();
}
doMore();
// Production code

// If you need to force or skip the condition
// you can do it with a covering test forcing a
// real-world scenario and not the code

testLargeCartItems()
testUserIsRetail()
```

Separation of concerns is extremely important, and business logic and hacks should always be treated separately.

Separation of Concerns

The *separation of concerns* concept aims to divide a software system into distinct, self-contained parts, with each part addressing a specific aspect or concern of the overall system. The goal is to create a modular and maintainable design that promotes code reusability, scalability, and ease of understanding by breaking it down into smaller, more manageable parts, allowing developers to focus on one concern at a time.

Related Recipes

Recipe 8.1, "Removing Commented Code"

8.4 Removing Getter Comments

Problem

You have getters with trivial comments.

Solution

Don't use getters. Don't comment on getters or other trivial functions.

Discussion

This recipe deals with the double problem of having getters as well as trivial comments. A few decades ago, people used to comment on every method. Even trivial ones. Here is an example of a getter comment on the `getPrice()` function:

```
contract Property {
    int private price;

    function getPrice() public view returns(int) {
           /* returns the Price */

        return price;
    }
}
```

Here's what it looks like after removing the getter comments:

```
contract Property {
    int private _price;

    function price() public view returns(int) {
        return _price;
    }
}
```

As an exception, if your function needs some comment and accidentally is a getter, you can add a nontrivial comment (preferably if it is related to a design decision).

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.8, "Removing Getters"

Recipe 8.5, "Converting Comments to Function Names"

8.5 Converting Comments to Function Names

Problem

You have code with lots of comments. Comments are coupled with implementation and hardly maintained.

Solution

Extract logic to a function with a name derived from the comment.

Discussion

If you have comments describing what a function should do, the best solution is to encode that comment into the function name. Make it intention-revealing. Review the following function with a bad name and a good describing comment:

```
final class ChatBotConnectionHelper {
    // ChatBotConnectionHelper is used
    // to create connection strings to Bot Platform
    // Use this class with getString() function
    // to get connection string to platform

    function getString() {
        //Get connection string from Chatbot
    }
}
```

After you apply the recipe you will have intention-revealing class and function names that need no comments:

```
final class ChatBotConnectionSequenceGenerator {

    function connectionSequence() {
    }
}
```

As a rule of thumb you can measure with your linters to detect comments and check the ratio of comments/lines of code against a predefined threshold (ideally near 1).

Related Recipes

Recipe 8.6, "Removing Comments Inside Methods"

8.6 Removing Comments Inside Methods

Problem

You have comments inside a method.

Solution

Don't add comments inside your methods. Extract them and leave declarative comments for very complex design decisions only.

Discussion

Comments inside methods split large actions into smaller ones. You should apply Recipe 10.7, "Extracting a Method to an Object", and rename the extracted methods with the comment description. Here is a long method separated by comments:

```
function recoverFromGrief() {
    // Denial stage
    absorbTheBadNews();
    setNumbAsProtectiveState();
    startToRiseEmotions();
    feelSorrow();

    // Anger stage
    maskRealEffects();
    directAngerToOtherPeople();
    blameOthers();
    getIrrational();

    // Bargaining stage
    feelVulnerable();
    regret();
    askWhyToMyself();
    dreamOfAlternativeWhatIfScenarios();
    postponeSadness();

    // Depression stage
    stayQuiet();
    getOverwhelmed();
    beConfused();
```

```
        // Acceptance stage
        acceptWhatHappened();
        lookToTheFuture();
        reconstructAndWalktrough();
    }
```

After you break it, it is more readable:

```
    function recoverFromGrief() {
        denialStage();
        angerStage();
        bargainingStage();
        depressionStage();
        acceptanceStage();
    }

    function denialStage() {
        absorbTheBadNews();
        setNumbAsProtectiveState();
        startToRiseEmotions();
        feelSorrow();
    }

    function angerStage() {
        maskRealEffects();
        directAngerToOtherPeople();
        blameOthers();
        getIrrational();
    }

    function bargainingStage() {
        feelVulnerable();
        regret();
        askWhyToMyself();
        dreamOfAlternativeWhatIfScenarios();
        postponeSadness();
    }

    function depressionStage() {
        stayQuiet();
        getOverwhelmed();
        beConfused();
    }

    function acceptanceStage() {
        acceptWhatHappened();
        lookToTheFuture();
        reconstructAndWalktrough();
    }
```

Comments serve a valuable purpose in documenting nonobvious design choices and you should not place them within the body of a function.

Related Recipes

Recipe 6.2, "Removing Empty Lines"

Recipe 6.7, "Documenting Design Decisions"

Recipe 8.5, "Converting Comments to Function Names"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 11.1, "Breaking Too Long Methods"

8.7 Replacing Comments with Tests

Problem

You have comments describing what a function does (and maybe how it achieves it) and instead of static and outdated descriptions, you want to have dynamic and well-maintained documentation.

Solution

Take your comment, compact it, and name your functions. Test it and remove the comments.

Discussion

Comments are seldom maintained and are harder to read than tests. Sometimes they even contain irrelevant implementation information. You can take the comment of the method explaining what the function does, rename the method with the comment description (the what), create tests to verify the comments, and omit irrelevant implementation details.

Here is a trivial example of a function describing what it does and how it accomplishes it:

```
def multiply(a, b):
    # This function multiplies two numbers and returns the result
    # If one of the numbers is zero, the result will be zero
    # If the numbers are both positive, the result will be positive
    # If the numbers are both negative, the result will be positive
    # The multiplication is done by invoking a primitive
    return a * b

# This code has a comment that explains what the function does.
# Instead of relying on this comment to understand the behavior of the code,
# you can write some unit tests that verify the behavior of the function.
```

After you remove the comment and create the test cases:

```
def multiply(first_multiplier, second_multiplier):
    return first_multiplier * second_multiplier

class TestMultiply(unittest.TestCase):
    def test_multiply_both_possitive_outcome_is_possitive(self):
        result = multiply(2, 3)
        self.assertEqual(result, 6)
    def test_multiply_both_negative_outcome_is_positive(self):
        result = multiply(-2, -4)
        self.assertEqual(result, 8)
    def test_multiply_first_is_zero_outcome_is_zero(self):
        result = multiply(0, -4)
        self.assertEqual(result, 0)
    def test_multiply_second_is_zero_outcome_is_zero(self):
        result = multiply(3, 0)
        self.assertEqual(result, 0)
    def test_multiply_both_are_zero_outcome_is_zero(self):
        result = multiply(0, 0)
        self.assertEqual(result, 0)

# You define a test function called test_multiply,
# which calls the multiply function with different arguments
# and verifies that the result is correct using the assertEqual method.

# 1. Take the comment of the method explaining what the function does.
# 2. Rename the method with the comment description (the what).
# 3. Create tests to verify the comments.
# 4. Omit irrelevant implementation details
```

You can rewrite the comment and compact it, but not always in an algorithmic way. This is not a safe refactor but it increases coverage. As an exception, you can't test private methods (see Recipe 20.1, "Testing Private Methods"). In the unlikely event that you need to replace a comment on a private method, you should test it indirectly or extract it into another object using Recipe 10.7, "Extracting a Method to an Object". As always, you can leave comments reflecting important design decisions.

Related Recipes

Recipe 8.2, "Removing Obsolete Comments"

Recipe 8.4, "Removing Getter Comments"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 20.1, "Testing Private Methods"

CHAPTER 9

Standards

The nice thing about standards is that you have so many to choose from. Furthermore, if you do not like any of them, you can just wait for next year's model.

—Andrew S. Tanenbaum, *Computer Networks, Fourth Edition*

9.0 Introduction

In large organizations, it is important to have code conventions to ensure that different teams and developers are working with a common set of rules and best practices. This can help to ensure that code is consistent and easy to understand, which can make it easier to work with and maintain, and also helps to improve the overall quality of the codebase. By enforcing a set of coding standards, organizations can ensure that developers are following best practices and writing code that is more likely to be reliable, scalable, and maintainable.

9.1 Following Code Standards

Problem

You work on a large codebase with many other developers. You need to read all the code with the same structure and conventions but there are many mixed standards.

Solution

Follow the same standards across the organization and enforce them (automatically if you can).

Discussion

Working on a solo project is easy unless you go back to it after some months. Working with many other developers requires some agreements. Following common code standards favors maintainability and readability and helps code reviewers. Most modern languages have common code standards like PSR2 in PHP, and most modern IDEs enforce them automatically.

In this example, you can see mixed code standards:

```
public class MY_Account {
    // This class name has a different case and underscores

    private Statement privStatement;
    // Attributes have visibility prefixes

    private Amount currentbalance = amountOf(0);

    public SetAccount(Statement statement) {
        this.statement = statement;
    }
    // Setters and getters are not normalized

    public GiveAccount(Statement statement)
    { this.statement = statement; }
    // Indentation is not uniform
    // Curly brace opens after function definition

    public void deposit(Amount value, Date date) {
        recordTransaction(
         value, date
        );
        // Some variables are named after type and not role
        // Parentheses are inconsistent
    }

    public void extraction(Amount value, Date date) {
        recordTransaction(value.negative(), date);
        // The opposite of *deposit* should be withdrawal
    }

    public void voidPrintStatement(PrintStream printer)
    {
    statement.printToPrinter(printer);
    // Name is redundant
    }

    private void privRecordTransactionAfterEnteredthabalance(
        Amount value, Date date) {

        Transaction transaction = new Transaction(value, date);
        Amount balanceAfterTransaction =
            transaction.balanceAfterTransaction(balance);

        balance = balanceAfterTransaction;

        statement.addANewLineContainingTransation(
            transaction, balanceAfterTransaction);
        // Naming is not uniform
        // Wrapped lines are not consistent
    }
}
```

Here's what it looks like if you follow (any arbitrary) common code standards:

```
public class Account {

    private Statement statement;

    private Amount balance = amountOf(0);

    public Account(Statement statement) {
        this.statement = statement;
    }

    public void deposit(Amount value, Date date) {
        recordTransaction(value, date);
    }

    public void withdrawal(Amount value, Date date) {
        recordTransaction(value.negative(), date);
    }

    public void printStatement(PrintStream printer) {
        statement.printOn(printer);
    }

    private void recordTransaction(Amount value, Date date) {
        Transaction transaction = new Transaction(value, date);
        Amount balanceAfterTransaction =
            transaction.balanceAfterTransaction(balance);
        balance = balanceAfterTransaction;
        statement.addLineContaining(transaction, balanceAfterTransaction);
    }
}
```

Linters and IDEs should test coding standards before a merge request is approved, and you can add your own rules for naming conventions related to any software elements like objects, classes, interfaces, modules, etc. Well-written clean code always follows standards with regard to naming conventions, formatting, and code style. Standards are helpful because they make things clear and deterministic for those who read your code, including yourself.

Automatic code formatting by a parser or compiler is the way machines provide feedback on how they interpret your instructions, prevents disagreements, and follows the fail fast principle. Code styling is automatic and mandatory in large organizations to enforce collective ownership.

Collective Ownership

Collective ownership states that all members of a development team have the ability to make changes to any part of the codebase, regardless of who originally wrote it. It is intended to promote a sense of shared responsibility, making code more manageable and easier to improve.

Related Recipes

Recipe 7.8, "Correcting Spelling Mistakes"

Recipe 10.4, "Removing Cleverness from Code"

9.2 Standardizing Indentations

Problem

You have code mixing tabs and spaces for indentation.

Solution

Don't mix indentation styles. Choose one and stick to it.

Discussion

There are various opinions on which style is better, but you can make your own decision. The most important thing here is code consistency. You can enforce them with code standard tests. Here is an example with mixed styles:

```
function add(x, y) {
   // --->..return x + y;
   return x + y;
}

function main() {
   // --->var x = 5,
   // --->....y = 7;
    var x = 5,
        y = 7;
}
```

Here's what it looks like when you standardize it:

```
function add(x, y) {
    // --->return x + y;
    return x + y;
}
```

Some languages like Python consider indent as part of the syntax. In these languages, indentation is not accidental since it changes code semantics. Some IDEs automatically convert one convention to the other.

Related Recipes

Recipe 9.1, "Following Code Standards"

9.3 Unifying Case Conventions

Problem

You have a codebase maintained by a lot of different people worldwide using different case conventions.

Solution

Don't mix different case conventions; choose one of them and enforce it.

Discussion

When different people make software together they might have personal or cultural differences as some prefer camel case, others snake_case, MACRO_CASE, and many others. The code should be straightforward and readable. There are also standard language conventions for cases, such as camelCase for Java or snake_case for Python.

Here are several case conventions mixed together in a JSON file:

```
{
    "id": 2,
    "userId": 666,
    "accountNumber": "12345-12345-12345",
    "UPDATED_AT": "2022-01-07T02:23:41.305Z",
    "created_at": "2019-01-07T02:23:41.305Z",
    "deleted at": "2022-01-07T02:23:41.305Z"
}
```

Here's what it looks like if you choose just one of them:

```
{
    "id": 2,
    "userId": 666,
    "accountNumber": "12345-12345-12345",
    "updatedAt": "2022-01-07T02:23:41.305Z",
    "createdAt": "2019-01-07T02:23:41.305Z",
    "deletedAt": "2022-01-07T02:23:41.305Z"
  // This doesn't mean THIS standard is the right one
}
```

You can tell your linters about your company's broad naming standards and enforce them. Whenever new people arrive at the organization, an automated test should politely ask to change the code. As valid exceptions, whenever you need to interact with out-of-your-scope code, you should use the client's standards, not yours.

Related Recipes

Recipe 9.1, "Following Code Standards"

9.4 Writing Code in English

Problem

You have code using your local language (not English) because business names are harder to translate.

Solution

Stick to English. Translate business names to English as well.

Discussion

All programming languages are written in English. Except for a few failed experiments during the '90s, all modern languages use English for their primitives and their frameworks. If you wanted to read or write in medieval Europe, you had to learn Latin. This is true for programming languages today with English and if you mix English with non-English names you can break polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic"), add cognitive load, make syntactic mistakes, break the bijection (as defined in Chapter 2), and more. Today, most IDEs and linters have translation tools or thesauri, and you can search for English translations of foreign words.

This example mixes English with Spanish:

```
const elements = new Set();
elements.add(1);
elements.add(1);

// This is the standard set
// Sets do not store duplicates
echo elements.size() yields 1

// You defined a multiset in Spanish
// because you are extending the domain
var moreElements = new MultiConjunto();

// 'multiconjunto' is the Spanish word for 'multiset'
// 'agregar' is the Spanish word for 'add'
moreElements.agregar('hello');
moreElements.agregar('hello');
echo moreElements.size() // yields 2 // Since it is a multiset

// elements and moreElements are NOT polymorphic
// You cannot exchange their accidental implementation

class Person {
  constructor() {
    this.visitedCities = new Set();
  }

  visitCity(city) {
    this.visitedCities.add(city);
    // Breaks if you change the set (expecting 'add()')
    // with a MultiConjunto (expecting 'agregar()')
  }
}
```

This is what it looks like written entirely in the English language:

```
const elements = new Set();
elements.add(1);
elements.add(1);

// This is the standard set
echo elements.size() // yields 1

// You define a multiset in English
var moreElements = new MultiSet();

moreElements.add('hello');
moreElements.add('hello');
echo moreElements.size() // yields 2 // Since it is a multiset
// elements and moreElements are polymorphic
// You can use either one in Person class. Even in runtime
```

9.5 Unifying Parameter Order

Problem

You have inconsistencies with the parameters you use.

Solution

Don't confuse your readers. Keep a consistent order.

Discussion

Code reads like prose. You need to read all methods in the same order. You can also use named parameters if your language supports them. In the following example, the two methods seem to be similar:

```
function giveFirstDoseOfVaccine(person, vaccine) { }

function giveSecondDoseOfVaccine(vaccine, person) { }

giveFirstDoseOfVaccine(jane, flu);
giveSecondDoseOfVaccine(jane, flu);
// Unnoticed mistake since you changed the parameters' order
```

Here's what it looks like if you are consistent with the parameter's sorting:

```
function giveFirstDoseOfVaccine(person, vaccine) { }

function giveSecondDoseOfVaccine(person, vaccine) { }

giveFirstDoseOfVaccine(jane, flu);
giveSecondDoseOfVaccine(jane, flu);
```

Here's what it looks like if you use named arguments:

```
function giveFirstDoseOfVaccine(person, vaccine) { }

giveFirstDoseOfVaccine(person=jane, vaccine=flu);
// equivalent to giveFirstDoseOfVaccine( vaccine=flu, person=jane);
giveSecondDoseOfVaccine(person=jane, vaccine=flu);
// equivalent to giveSecondDoseOfVaccine( vaccine=flu, person=jane);
```

Named Parameters

Named parameters are a feature of many programming languages that allow a programmer to specify the value of a parameter by providing its name rather than its position in the list of parameters. They are also known as keyword arguments.

Related Recipes

Recipe 7.16, "Removing Redundant Parameter Names"

Recipe 11.2, "Reducing Excess Arguments"

9.6 Fixing Broken Windows

Problem

You are changing some part of the code but find something else out of place.

Solution

Follow the Boy Scout rule (see Recipe 7.4, "Renaming Result Variables") and leave the code cleaner than you found it. If you find a mess, clean it up regardless of who might have made it. If you find a problem, fix it.

Discussion

As a programmer, you read code many more times than you write it. Take ownership of code you read that contains errors and leave it better. If you encounter code like this while making other changes:

```
      int mult(int a,int other)
       { int prod
         prod= 0;
         for(int i=0;i<other ;i++)
           prod+= a ;
            return prod;
       }

   // Formatting, naming, assignment and standards inconsistent
```

You should change it by applying many recipes:

```
   int multiply(int firstMultiplier, int secondMultiplier) {
     int product = 0;
     for(int index=0; index<secondMultiplier; index++) {
       product += firstMultiplier;
     }
     return product;
   }

   // or just multiply them :)
```

Don't be afraid to make changes, always strive for good test coverage to ensure no business functionality is impacted, and remember that developing software is a team activity, so you need to find consensus while making these kinds of changes.

Related Recipes

Recipe 9.2, “Standardizing Indentations”

Recipe 9.3, “Unifying Case Conventions”

Recipe 21.4, “Preventing and Removing ToDos and FixMes”

CHAPTER 10

Complexity

Object-oriented programming increases the value of these metrics by managing this complexity. The most effective tool available for dealing with complexity is abstraction. Many types of abstraction can be used, but encapsulation is the main form of abstraction by which complexity is managed in object-oriented programming.

—Rebecca Wirfs-Brock and Brian Wilkinson, "Object-Oriented Design: A Responsibility-Driven Approach"

10.0 Introduction

According to David Farley, if you want to be an excellent software engineer, you need to be an expert at learning, and your only duty is to keep accidental complexity at the lowest possible levels. Complexity is present in every large software system and is often the main source of problems. One of the core differences between a young software developer and a more experienced one is how they manage accidental complexity and keep it at a minimum.

10.1 Removing Repeated Code

Problem

You have duplicate behavior in your code. Duplicated behavior is not the same as duplicated code since code is not text.

Solution

You need to find the missing abstraction and move the repeated behavior there.

Discussion

Duplicated code hurts maintainability and violates the *don't repeat yourself* principle. It also increases maintenance costs, and changes can become time-consuming and error-prone. If the code has a defect, it might be present in multiple places. Duplicated code also prevents reusability since there are missing abstractions. You need to consider these drawbacks before using copy-and-paste commands.

Here is some duplicated code for a text replacer used on a `WordProcessor` and an `Obfuscator`:

```
class WordProcessor {

    function replaceText(string $patternToFind, string $textToReplace) {
        $this->text = '<<<' .
            str_replace($patternToFind, $textToReplace, $this->text) . '>>>';
    }
}

final class Obfuscator {

    function obfuscate(string $patternToFind, string $textToReplace) {
        $this->text =
            strlower(str_ireplace($patternToFind, $textToReplace, $this->text));
    }
}
```

Here's what it looks like when you encapsulate the text replacer logic into a new abstraction:

```
final class TextReplacer {
    function replace(
        string $patternToFind,
        string $textToReplace,
        string $subject,
        string $replaceFunctionName,
        $postProcessClosure) {
        return $postProcessClosure(
            $replaceFunctionName($patternToFind, $textToReplace, $subject));
    }
}

// Lots of tests on text replacer so you can gain confidence.

final class WordProcessor {
    function replaceText(string $patternToFind, string $textToReplace) {
        $this->text = (new TextReplacer())->replace(
            $patternToFind,
            $textToReplace,
            $this->text,
            'str_replace', fn($text) => '<<<' . $text . '>>>');
    }
}

final class Obfuscator {
    function obfuscate(string $patternToFind, string $textToReplace) {
        $this->text = (new TextReplacer())->replace(
            $patternToFind,
            $textToReplace,
            $this->text,
            'str_ireplace', fn($text) => strlower($text));
    }
}
```

Linters can find repeated code but are not very good at finding similar patterns. Maybe soon machine learning will help you find such abstractions automatically. With your refactoring tools, you need to refactor such code, trusting your tests as a safety net.

Copy-and-Paste Programming

Copy-and-paste programming is a technique where you copy existing code and paste it into another location, rather than writing new code. If you heavily utilize copy and paste, your code is less maintainable.

Related Recipes

Recipe 19.3, "Breaking Subclassification for Code Reuse"

10.2 Removing Settings/Configs and Feature Toggles

Problem

Your code relies on global configurations, settings, or feature toggles.

Solution

Identify, track, and remove feature toggles and customizations once they are mature. Reify configuration into small objects.

Discussion

Feature Flags

A *feature flag* (also known as a *feature toggle* or *feature switch*) allows you to enable or disable a specific feature or functionality at runtime, without requiring a full new deployment. This allows you to release new features to a subset of users or environments while keeping them hidden from others to perform A/B testing and early betas or canary releases.

Changing system behavior in a control board is a customer's dream. And a software engineer's nightmare. Settings bring global coupling, if pollution, and an explosion of testing scenarios. You can manage your configuration by creating polymorphic objects and injecting them externally. You should configure your objects so they can behave in different ways, and you should achieve that with explicit behavioral objects. A system with 300 boolean configurations has more test combinations (2^{300}) than the number of atoms in the universe (10^{80}).

Here is an example where a setting injected globally affects the way you retrieve the objects:

```
class VerySpecificAndSmallObjectDealingWithPersistency {
  retrieveData() {
    if (GlobalSettingsSingleton.getInstance().valueAt('RetrievDataDirectly')) {
      // Notice the unnoticed typo in 'RetrievDataDirectly'
      this.retrieveDataThisWay();
    }
    else {
      this.retrieveDataThisOtherWay();
    }
  }
}
```

You can explicitly use the strategy design pattern (see Recipe 14.4, "Replacing Switch/Case/Elseif Statements") and test it after you break the global coupling:

```
class VerySpecificAndSmallObjectDealingWithPersistency {
  constructor(retrievalStrategy) {
    this.retrievalStrategy = retrievalStrategy;
  }
  retrieveData() {
    this.retrievalStrategy.retrieveData();
  }
}
// You get rid of the if condition by using a polymorphic strategy
```

This is an architectural pattern, so you should control it or directly avoid it with design policies. As notable exceptions, sometimes you use feature toggling as a safeguard mechanism. This is acceptable in a legacy system, but these toggles should be very short-lived (a few weeks) in a CI/CD system.

A/B Testing

A/B testing compares two different versions of released software to determine which one is better for the final users.

Related Recipes

Recipe 14.16, "Reifying Hardcoded Business Conditions"

Recipe 17.3, "Breaking God Objects"

10.3 Changing State as Properties

Problem

You make state changes by modifying some internal attributes.

Solution

Model the status of your object as a set inclusion in a similar way to the real-world metaphor in the MAPPER.

Discussion

This recipe is counterintuitive unless you think of it under the mutability umbrella. You should model states as a mathematical set inclusion since a state is always *accidental;* you need to extract it and take it away from the object. Every state diagram of your object lifecycle is an opportunity to apply this recipe. Software engineers face a very difficult challenge in finding good models that survive their entire lifetime, as such models are rare.

Here's `Order` with an attribute modeling a possible state:

```
public abstract class OrderState {}

public class OrderStatePending extends OrderState {}
// This is a polymorphic hierarchy with different behavior
// An enum is not enough to model state

public class Order {
    public Order(LinkedList<int> items) {
        LinkedList<int> items = items;
        OrderState state = new OrderStatePending();
    }

    public function changeState(OrderState newState) {
        OrderState state = newState;
    }

    public function confirm() {
        state.Confirm(this);
    }

}
```

Here's what it looks like if you take the state out of the `Order` and manage collection grouping by state:

```
class Order {

    public Order(LinkedList<int> items) {
        items = items;
    }
}

class OrderProcessor {
    public static void main(String args[]) {

    LinkedList<int> elements = new LinkedList<int>();
    elements.add(1);
    elements.add(2);

    Order sampleOrder = new Order(elements);

    Collection<Order> pendingOrders = new LinkedList<Order>();
    Collection<Order> confirmedOrders = new LinkedList<Order>();

    pendingOrders.add(sampleOrder);

    pendingOrders.remove(sampleOrder);
    confirmedOrders.add(sampleOrder);
    }
}
```

In Figure 10-1, you can see that *Order 1* belongs to the pending orders collection both in the real world and the model. Making it a state would break the bijection. Confirmed orders are empty right now.

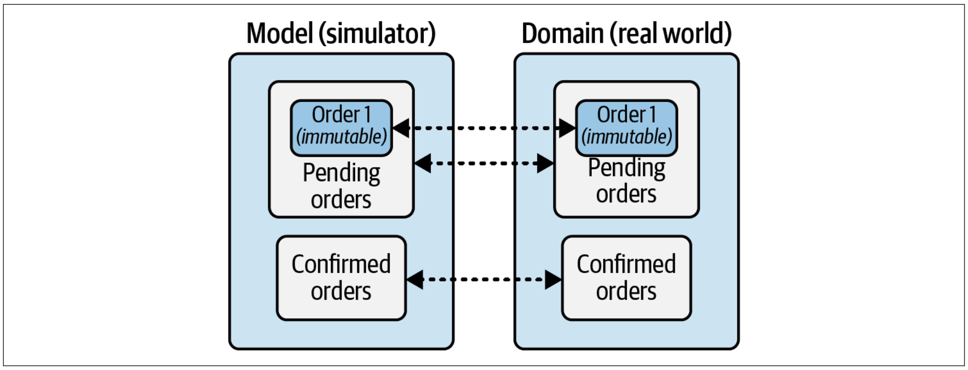


Figure 10-1. Orders belong to the same sets in the model and the real world

If you want to be extreme, you should consider *every* setter to be a potential state change. There are no silver bullets (see Recipe 4.1, "Creating Small Objects") preventing you from overdesigning. For example, changing the color of a visual component should be a counterexample. You should be aware and very cautious.

Overdesigning

Overdesigning is the practice of adding unnecessary accidental complexity to a software application. This can happen when you focus too much on making the software as feature rich as possible, rather than keeping it simple and focused on the core functionality.

Related Recipes

Recipe 3.3, "Removing Setters from Objects"

Recipe 16.2, "Removing Premature Optimization"

10.4 Removing Cleverness from Code

Problem

You find code difficult to read and tricky, full of names without semantics. Sometimes the code uses a language's accidental complexity.

Solution

Remove cleverness and tricks. Be humble and don't act as if you are too smart. Clean code asks for readability and simplicity over micro hacks.

Discussion

Cleverness is the opposite of readability and maintainability. Clever code full of premature optimization is hard to maintain and often has quality problems. Here is an algorithm to get the prime factors of a number:

```
function primeFactors(n){
  var f = [], i = 0, d = 2;

  for (i = 0; n >= 2; ) {
     if(n % d == 0){
       f[i++]=(d);
       n /= d;
    }
    else{
      d++;
    }
  }
  return f;
}
```

You should cover it with tests to ensure you don't break the code. Then make small refactors and renames, using recipes from this book to leave code cleaner. Follow the Boy Scout rule: remove the cleverness to make the code better than when you first encountered it:

```
function primeFactors(numberToFactor) {
  var factors = [],
  divisor = 2,
  remainder = numberToFactor;

  while(remainder>=2) {
    if(remainder % divisor === 0){
       factors.push(divisor);
       remainder = remainder / divisor;
    }
    else {
      divisor++;
    }
  }
  return factors;
}
```

A notable exception where you should exercise cleverness is optimized code for low-level operations since performance is more important than readability in such scenarios. You can write the code without optimizing it, then cover it with automated tests to ensure it works as expected. After you have enough test coverage, you can improve it, even to the point of sacrificing some readability. This is also an opportunity to use the test-driven development (see Recipe 4.8, "Removing Unnecessary Properties") technique on existing systems.

Related Recipes

Recipe 6.8, "Replacing Magic Numbers with Constants"

Recipe 6.15, "Avoiding Magic Corrections"

Recipe 16.2, "Removing Premature Optimization"

10.5 Breaking Multiple Promises

Problem

You have independent promises and you need to wait for all of them to complete.

Solution

Don't block yourself in an ordered way. Wait for all promises at once.

Discussion

While studying operating systems, you likely learned about semaphores, which are useful for waiting until all conditions are met, regardless of their ordering.

Semaphores

A *semaphore* is a synchronization object that helps manage access to shared resources and coordinate communication between concurrent processes or threads.

Here's an example with serial promises:

```
async fetchLongTask() { }
async fetchAnotherLongTask() { }

async fetchAll() {
  let result1 = await this.fetchLongTask();
  let result2 = await this.fetchAnotherLongTask();
  // But they can run in parallel !!
}
```

Here's what it looks like when you wait for them in parallel:

```
async fetchLongTask() { }
async fetchAnotherLongTask() { }

async fetchAll() {
  let [result1, result2] =
      await Promise.all([this.fetchLongTask(), this.fetchAnotherLongTask()]);
```

```
        // You wait until ALL are done
}
```

You can tell your linters to find certain patterns related to waiting for promises and always work to be as close as possible to real-world business rules. If the rule states you need to wait for *all* operations, you should not force a particular order.

Promises

A *promise* is a special object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

10.6 Breaking Long Chains of Collaborations

Problem

You have long chains of method calls.

Solution

Making long method chains generates coupling and ripple effects. Any change along the chain breaks the code. The solution is to send messages only to your acquaintances.

Discussion

If you have long method chains, the coupling propagates from the first to the last call. You also break encapsulation and violate Demeter's law (see Recipe 3.8, "Removing Getters") and the "Tell, don't ask" principle (see Recipe 3.3, "Removing Setters from Objects"). To solve this problem, you can create intermediate methods and higher-level messages.

The following example asks for the dog's feet to move:

```
class Dog {
  constructor(feet) {
     this.feet = feet;
  }
  getFeet() {
    return this.feet;
  }
}

class Foot {
   move() { }
}
```

```
feet = [new Foot(), new Foot(), new Foot(), new Foot()];
dog = new Dog(feet);

for (var foot of dog.getFeet()) {// incursion = 2
  foot.move();
}
// Equivalent to dog.getFeet()[0].move(); dog.getFeet()[1].move() ...
```

Here's what it looks like when you delegate the responsibility to the dog to achieve its goal:

```
class Dog {
   constructor(feet) {
     this.feet = feet;
  }
  walk() {
    // This is encapsulated on how the dog walks
    for (var foot of this.feet) {
      foot.move();
    }
  }
}

class Foot {
  move() { }
}

feet = [new Foot(), new Foot(), new Foot(), new Foot()];
dog = new Dog(feet);
dog.walk();
```

Avoid successive message calls. Try to hide the intermediate collaborations and create new protocols instead.

Related Recipes

Recipe 17.9, "Removing the Middleman"

10.7 Extracting a Method to an Object

Problem

You have a long algorithmic method. You want to understand it, test it, and reuse some parts.

Solution

Move it inside an object and break it into smaller parts.

Discussion

You find long methods hard to debug and test, especially if they have protected visibility. Algorithms exist in the real world and deserve their own objects. You need to create an object to represent an invocation of the method, move the big method to the new object, and convert the temporary variables of the method into private attributes. Finally, remove parameters from method invocation by also converting them to private attributes.

A method object is suitable when you are using several *extract methods*, passing a partial state among them as parts of an algorithm. A strong indicator of a method-object opportunity is when computations are not cohesively related to the host method. You can also reify anonymous functions with more atomic, cohesive, and testable method objects.

Imagine a big balance calculation method:

```
class BlockchainAccount {
  // ...
  public double balance() {
    string address;
    // Very long untestable method
  }
}
```

Here's what it looks like when you reify and refactor it:

```
class BlockchainAccount {
  // ...
  public double balance() {
    return new BalanceCalculator(this).netValue();
  }
}

// 1. Create an object to represent an invocation of the method
// 2. Move the big method to the new object
// 3. Convert the temporary variables of the method into private attributes
// 4. Break the big method in the new object by using the Extract Method
// 5. Remove parameters from method invocation
// by also converting them to private attributes

class BalanceCalculator {
  private string address;
  private BlockchainAccount account;

  public BalanceCalculator(BlockchainAccount account) {
    this.account = account;
  }

  public double netValue() {
    this.findStartingBlock();
    //...
    this computeTransactions();
  }
}
```

Some IDEs have tools to extract a function into a method object. You can make the changes automatically in a safe way and extract the logic into a new component, unit-test it, reuse it, exchange it, etc.

Related Recipes

Recipe 11.1, "Breaking Too Long Methods"

Recipe 11.2, "Reducing Excess Arguments"

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 14.13, "Extracting from Long Ternaries"

Recipe 20.1, "Testing Private Methods"

Recipe 23.2, "Reifying Anonymous Functions"

See Also

Original method object definition in Chapter 3 of *Smalltalk Best Practice Patterns* by Kent Beck

"Method Object" on C2 Wiki

10.8 Looking After Array Constructors

Problem

You use `Array` creation in JavaScript with `new Array()`.

Solution

Be very careful with JavaScript `Array`s and avoid `new Array()` since it is not homogenous or predictable.

Discussion

`new Array()` violates the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants") in JavaScript as this language has a lot of magic tricks. A language should be intuitive, homogeneous, predictable, and simple, but JavaScript, Python, PHP, and many others are not. You should use these languages as simply, clearly, and predictably as possible.

Here is a counterintuitive example when you create an array with one argument (the number 5):

```
const arrayWithFixedLength = new Array(3);

console.log(arrayWithFixedLength); // [ <3 empty items> ]
console.log(arrayWithFixedLength[0]); // Undefined
console.log(arrayWithFixedLength[1]); // Undefined
console.log(arrayWithFixedLength[2]); // Undefined
console.log(arrayWithFixedLength[3]); // Undefined too
// But should be Index out of range
console.log(arrayWithFixedLength.length); // 3
```

And this is what happens when you create it with two arguments:

```
const arrayWithTwoElements = new Array(3, 1);

console.log(arrayWithTwoElements); // [ 3, 1 ]
console.log(arrayWithTwoElements[0]); // 3
console.log(arrayWithTwoElements[1]); // 1
console.log(arrayWithTwoElements[2]); // Undefined
console.log(arrayWithTwoElements[5]); // Undefined (should be out of range)
console.log(arrayWithTwoElements.length); // 2

const arrayWithTwoElementsLiteral = [3,1];

console.log(arrayWithTwoElementsLiteral); // [ 3, 1 ]
console.log(arrayWithTwoElementsLiteral[0]); // 3
console.log(arrayWithTwoElementsLiteral[1]); // 1
console.log(arrayWithTwoElementsLiteral[2]); // Undefined
console.log(arrayWithTwoElementsLiteral[5]); // Undefined
console.log(arrayWithTwoElementsLiteral.length); // 2
```

The best solution is to avoid `new Array()` creation and use the syntactic constructor `[]`. Many "modern" languages are full of hacks intended to make life easier for programmers, but they are actually a source of potential undiscovered defects.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 13.3, "Using Stricter Parameters"

Recipe 24.2, "Dealing with Truthy Values"

10.9 Removing Poltergeist Objects

Problem

You have an object that appears and disappears mysteriously.

Solution

Add the necessary indirection layers, but no more.

Discussion

When adding intermediate objects, you add accidental complexity and damage readability. You can remove intermediate volatile objects if they add no business value to your solution, following the YAGNI principle (see Chapter 12, "YAGNI").

Poltergeist Objects

A *poltergeist* is a short-lived object used to perform initialization or to invoke methods in another, more permanent class.

Here you have a driver you create and discard to move a car:

```
public class Driver
{
    private Car car;

    public Driver(Car car)
    {
        this.car = car;
    }

    public void DriveCar()
    {
        car.Drive();
    }
}

Car porsche = new Car();
Driver homer = new Driver(porsche);
homer.DriveCar();
```

You can remove it, as shown here:

```
// You don't need the driver
Car porsche = new Car();
porsche.driveCar();
```

Don't add accidental complexity to the essential complexity you already have, and remove the middleman objects if they are not needed.

Related Recipes

Recipe 16.6, "Removing Anchor Boats"

Recipe 17.9, "Removing the Middleman"

CHAPTER 11

Bloaters

The purpose of software engineering is to control complexity, not to create it.

—Pamela Zave in *Programming Pearls, 2nd Edition* by Jon Bentley

11.0 Introduction

Bloaters are unavoidable when your code grows and many people collaborate. They often don't cause performance problems, but they damage maintainability and testability, preventing good software from evolving. The code becomes unnecessarily large, complex, and difficult to maintain, often due to the inclusion of unnecessary features, poor design choices, or excessive repetition. Code gets bloated in small steps and then you notice you have a big mess. You don't write long methods, but maybe you add small portions and a teammate adds some more, and so on. This is a kind of technical debt (see Chapter 21, "Technical Debt") that is easier to reduce with state-of-the-art automated tools.

11.1 Breaking Too Long Methods

Problem

You have a method with too many lines of code.

Solution

Extract the long method into smaller pieces. Break complex algorithms into parts. You can also unit-test these parts.

Discussion

Long methods have low cohesion and high coupling. They are difficult to debug and have low reusability. You can use this recipe to break structured libraries and helpers into smaller behaviors (see Recipe 7.2, "Renaming and Breaking Helpers and Utils"). The amount of lines depends on the programming language, but 8 to 10 lines should be enough for most of them.

Here's a long method:

```
function setUpChessBoard() {

    $this->placeOnBoard($this->whiteTower);
    $this->placeOnBoard($this->whiteKnight);
    // A lot of lines
    // .....
    $this->placeOnBoard($this->blackTower);
}
```

You can break it into parts as follows:

```
function setUpChessBoard() {
    $this->placeWhitePieces();
    $this->placeBlackPieces();
}
```

Now you can unit-test each one of them; you should be careful not to couple the tests to implementation details. All linters can measure and warn you when methods are larger than a predefined threshold.

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 8.6, "Removing Comments Inside Methods"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 14.13, "Extracting from Long Ternaries"

See Also

"Long Method" on Refactoring Guru

11.2 Reducing Excess Arguments

Problem

You have a method that needs too many arguments.

Solution

Don't pass more than three arguments to your method. Group related arguments together into parameter objects. You can bind them together.

Discussion

Methods with too many arguments have low maintainability, low reuse, and high coupling. You need to group them to find cohesive relations among arguments or just create a small object with the arguments' context. When you create such a context, you will enforce relations between parameters on creation, following the fail fast principle.

You should also avoid "basic" types, such as strings, arrays, integers, etc., and think about small objects (see Recipe 4.1, "Creating Small Objects"). You need to relate arguments and group them. Always favor real-world mappings. Figure out how the counterparts of the arguments in the real world group into cohesive objects. If a function gets too many arguments, some of them might be related to class construction.

This example invokes the `print` method with a lot of arguments:

```
public class Printer {
  void print(
           String documentToPrint,
           String paperSize,
           String orientation,
           boolean grayscales,
           int pageFrom,
           int pageTo,
           int copies,
           float marginLeft,
           float marginRight,
           float marginTop,
           float marginBottom
        ) {
    }
}
```

Instead, group some of the arguments together to avoid primitive obsession:

```
final public class PaperSize { }
final public class Document { }
final public class PrintMargins { }
final public class PrintRange { }
final public class ColorConfiguration { }
final public class PrintOrientation { }
// Class definition with methods and properties omitted for simplicity

final public class PrintSetup {
    public PrintSetup(
           PaperSize papersize,
           PrintOrientation orientation,
           ColorConfiguration color,
           PrintRange range,
           int copiesCount,
           PrintMargins margins
           ) {}
}

final public class Printer {
  void print(
         Document documentToPrint,
         PrintSetup setup
        ) {
    }
}
```

Most linters warn when the arguments list is too large, so you can apply this recipe if necessary.

Related Recipes

Recipe 3.7, "Completing Empty Constructors"

Recipe 9.5, "Unifying Parameter Order"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 11.6, "Breaking Too Many Attributes"

11.3 Reducing Excess Variables

Problem

Your code has too many variables declared and active.

Solution

Break the scope and let the variables be as local as possible.

Discussion

If you narrow your variable scope, you will have better readability and will reuse smaller pieces of the code. You will also find opportunities to remove unused variables. Your code could get dirty when programming and test cases may fail. With good coverage, you can iterate and narrow scopes while refactoring and reducing methods using Recipe 10.7, "Extracting a Method to an Object". Scopes are more evident in smaller contexts.

The following example has a lot of active variables at once:

```
function retrieveImagesFrom(array $imageUrls) {
  foreach ($imageUrls as $index => $imageFilename) {
    $imageName = $imageNames[$index];
    $fullImageName = $this->directory() . "\\" . $imageFilename;
    if (!file_exists($fullImageName)) {
      if (str_starts_with($imageFilename, 'https://cdn.example.com/')) {
          $url = $imageFilename;
          // This variable duplication is not really necessary
          // when you scope variables
          $save_to = "\\tmp"."\\".basename($imageFilename);
          $ch = curl_init ($url);
          curl_setopt($ch, CURLOPT_HEADER, 0);
          curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
          $raw = curl_exec($ch);
          curl_close ($ch);
          if(file_exists($saveTo)){
               unlink($saveTo);
          }
          $fp = fopen($saveTo,'x');
          fwrite($fp, $raw);
          fclose($fp);
          $sha1 = sha1_file($saveTo);
          $found = false;
          $files = array_diff(scandir($this->directory()), array('.', '..'));
          foreach ($files as $file){
              if ($sha1 == sha1_file($this->directory()."\\".$file)) {
                  $images[$imageName]['remote'] = $imageFilename;
                  $images[$imageName]['local'] = $file;
                  $imageFilename = $file;
                  $found = true;
                  // Iteration keeps going on even after you find it
              }
          }
          if (!$found){
            throw new \Exception('Image not found');
         }
        // Debugging at this point your context is polluted with variables
        // from previous executions no longer needed
        // for example: the curl handler
  }
```

Here's what it looks like after you narrow some of the scopes:

```
function retrieveImagesFrom(string imageUrls) {
  foreach ($imageUrls as $index => $imageFilename) {
    $imageName = $imageNames[$index];
    $fullImageName = $this->directory() . "\\" . $imageFilename;
    if (!file_exists($fullImageName)) {
        if ($this->isRemoteFileName($imageFilename)) {
            $temporaryFilename = $this->temporaryLocalPlaceFor($imageFilename);
            $this->retrieveFileAndSaveIt($imageFilename, $temporaryFilename);
            $localFileSha1 = sha1_file($temporaryFilename);
            list($found, $images, $imageFilename) =
              $this->tryToFindFile(
                $localFileSha1, $imageFilename, $images, $imageName);
            if (!$found) {
                throw new Exception('File not found locally ('.$imageFilename
                + ') Need to retrieve it and store it');
            }
        } else {
            throw new \Exception('Image does not exist on directory ' .
              $fullImageName);
        }
    }
```

Most linters can suggest avoiding the use of long methods, and this warning also hints that you should break and scope your variables. You should use Recipe 10.7, "Extracting a Method to an Object", in small, baby steps.

Baby Steps

Baby steps refers to an iterative and incremental approach where you make small, manageable tasks or changes during the development process. The concept of baby steps is rooted in the Agile development methodology.

Related Recipes

Recipe 6.1, "Narrowing Reused Variables"

Recipe 11.1, "Breaking Too Long Methods"

Recipe 14.2, "Renaming Flag Variables for Events"

11.4 Removing Excessive Parentheses

Problem

You have an expression with too many parentheses.

Solution

Use as few parentheses as possible without changing the code semantics.

Discussion

You read code from left to right (at least in Western culture), and parentheses often break this flow, adding cognitive complexity. You write code once and read it many more times, so readability is king. Here's a formula with too many parentheses used to calculate the Schwarzschild radius, which is a measure of the size of a nonrotating black hole:

```
schwarzschild = ((((2 * GRAVITATION_CONSTANT)) * mass) / ((LIGHT_SPEED ** 2)))
```

Here's what it looks like after you remove your extra parentheses:

```
schwarzschild = (2 * GRAVITATION_CONSTANT * mass) / (LIGHT_SPEED ** 2)
```

You can compact it further:

```
schwarzschild = 2 * GRAVITATION_CONSTANT * mass / (LIGHT_SPEED ** 2)
```

By following the order of operations in mathematics, you know that multiplication and division take precedence over addition and subtraction, so the multiplication of 2, `GRAVITATION_CONSTANT`, and `mass` can be performed first, followed by the division by `(LIGHT_SPEED ** 2)`, but it is less readable than the previous example. On some complex formulas, you can add extra parentheses for term readability. As with many recipes, it is always a trade-off.

Related Recipes

Recipe 6.8, "Replacing Magic Numbers with Constants"

11.5 Removing Excess Methods

Problem

You have too many methods in your class.

Solution

Break the class into more cohesive small pieces, and don't add any accidental protocol to your classes.

Discussion

Engineers tend to put a protocol in the first class they see fit. That's not a problem; you just need to refactor it after your tests cover the functionality. In this example the helper class has a lot of incohesive methods:

```
public class MyHelperClass {
  public void print() { }
  public void format() { }
  // ... many methods more

  // ... even more methods
  public void persist() { }
  public void solveFermiParadox() { }
}
```

You can break them into relevant abstractions using the MAPPER:

```
public class Printer {
  public void print() { }
}

public class DateToStringFormatter {
  public void format() { }
}

public class Database {
  public void persist() { }
}

public class RadioTelescope {
  public void solveFermiParadox() { }
}
```

Most linters count methods and can warn you so you can refactor the classes. Splitting them is a good practice to favor small and reusable objects.

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 11.6, "Breaking Too Many Attributes"

Recipe 11.7, "Reducing Import Lists "

Recipe 17.4, "Breaking Divergent Change"

Recipe 17.15, "Refactoring Data Clumps"

See Also

"Large Class" on Refactoring Guru

11.6 Breaking Too Many Attributes

Problem

You have a class that defines objects with lots of attributes.

Solution

Break the class into cohesive parts. Find methods related to attributes, then cluster these methods and break the object related to those clusters. In the end, find real objects related to these new objects and replace existing references.

Discussion

Here you can see a spreadsheet with too many attributes:

```
class ExcelSheet {
  String filename;
  String fileEncoding;
  String documentOwner;
  String documentReadPassword;
  String documentWritePassword;
  DateTime creationTime;
  DateTime updateTime;
  String revisionVersion;
  String revisionOwner;
  List previousVersions;
  String documentLanguage;
  List cells;
  List cellNames;
  List geometricShapes;
}
```

Here's what it looks like after breaking it into parts:

```
class ExcelSheet {
  FileProperties fileProperties;
  SecurityProperties securityProperties;
  DocumentDatingProperties datingProperties;
  RevisionProperties revisionProperties;
  LanguageProperties languageProperties;
  DocumentContent content;
}

// Object has less attributes
// They are not only grouped for testability
// New objects are more cohesive, more testable,
// have fewer conflicts, and are more reusable
// FileProperties/SecurityProperties can be reused for other documents
// Rules and preconditions on fileProperties will be moved to this object
// so ExcelSheet constructor will be cleaner
```

Most linters warn you when you declare too many attributes. Setting a good warning threshold should be easy since bloated objects know too much and are very difficult to change due to cohesion. Developers change these objects a lot, so they create merge conflicts and are a common source of problems.

Related Recipes

Recipe 11.2, "Reducing Excess Arguments"

Recipe 11.5, "Removing Excess Methods"

Recipe 17.3, "Breaking God Objects"

Recipe 17.4, "Breaking Divergent Change"

11.7 Reducing Import Lists

Problem

Your class relies on too many others; it will be coupled and fragile. A long import list is a good indicator of this problem.

Solution

Don't import too much in the same file; break the dependencies and coupling.

Discussion

You can break the class and hide the intermediate accidental implementation. Here's a very long import list:

```
import java.util.LinkedList;
import java.persistence;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.NoSuchElementException
import java.util.Queue;
import org.fermi.common.util.ClassUtil;
import org.fermi.Data;
// You rely on too many libraries

public class Demo {
   public static void main(String[] args) {

   }
}
```

Here's a simplified solution:

```java
import org.fermi.domainModel;
import org.fermi.workflow;

// You rely on few libraries
// and you hide their implementation
// Maybe transitive imports are the same
// but you don't break encapsulation

public class Demo {
   public static void main(String[] args) {

   }
}
```

You can set up a warning threshold on your linters. You'll also need to think about dependencies when building your solutions to minimize ripple effects. Most modern IDEs will raise warnings for unused imports.

Related Recipes

Recipe 11.5, "Removing Excess Methods"

Recipe 17.4, "Breaking Divergent Change"

Recipe 17.14, "Changing Coupling to Classes"

Recipe 25.3, "Removing Package Dependencies"

11.8 Breaking "And" Functions

Problem

You have functions that perform more than one task.

Solution

Unless you need atomicity, do not perform more than one task per function and break the composite functions.

Discussion

If you find a function with "and" as part of the name and don't need atomicity, you should split it because making two things in the same scope causes coupled code. The code is harder to test and read. You can extract and break the method. Doing more than one thing at once creates coupling and violates the single-responsibility principle (see Recipe 4.7, "Reifying String Validations"). It also hurts testability.

Here is a function performing two tasks:

```
def fetch_and_display_personnel():
  data = # ...

  for person in data:
    print(person)
```

Here's what it looks like after you break it:

```
def fetch_personnel():
  return # ...

def display_personnel(data):
  for person in data:
    print(person)
```

Here is another example:

```
calculatePrimeFactorsRemoveDuplicatesAndPrintThem()

// Three responsibilities
```

Here's what it looks like after you split it into three parts:

```
calculatePrimeFactors();

removeDuplicates();

printNumbers();

// Three different methods
// You can test them and reuse them
```

Functions with "and" in their name are good candidates for breaking. However, you need to check them carefully since there might be false positives. You should avoid doing more than needed, and your functions should be both minimal and atomic. When making methods, it is very important to utilize the rubber duck method to determine if you are making things correctly.

Rubber Duck Debugging

Rubber duck debugging is the concept of explaining your code line by line as if you were teaching a rubber duck how to program. By verbalizing and describing each step of your code, you may discover errors or logical inconsistencies that you may have missed before.

Related Recipes

Recipe 11.1, "Breaking Too Long Methods"

11.9 Breaking Fat Interfaces

Problem

You have interfaces declaring too much protocol.

Solution

Split your interfaces.

Discussion

The term "fat interface" emphasizes that the interface is overloaded with methods, including those that may not be necessary or used by all clients. The interface violates the principle of segregating interfaces into smaller, more focused contracts.

Interface Segregation Principle

The *interface segregation principle* states that objects should not be forced to depend on interfaces they do not use. It is better to have many small, specialized interfaces rather than one large, monolithic interface.

In the following example, you override some behavior:

```
interface Animal {
  void eat();
  void sleep();
  void makeSound();
  // This protocol should be common to all animals
}

class Dog implements Animal {
  public void eat() { }
  public void sleep() { }
  public void makeSound() { }
}

class Fish implements Animal
  public void eat() { }
  public void sleep() {
    throw new UnsupportedOperationException("I do not sleep");}
  public void makeSound() {
    throw new UnsupportedOperationException("I cannot make sounds");
  }
}

class Bullfrog implements Animal
  public void eat() { }
```

```
    public void sleep() {
      throw new UnsupportedOperationException("I do not sleep");
    }
    public void makeSound() { }
  }
```

When you segregate the interfaces to more atomic ones:

```
interface Animal {
  void move();
  void reproduce();
}
// You can even break these two responsibilities

class Dog implements Animal {
  public void move() { }
  public void reproduce() { }
}

class Fish implements Animal {
  public void move() { }
  public void reproduce() { }
}

class Bullfrog implements Animal {
  public void move() { }
  public void reproduce() { }
}
```

You can check the size of the interface behavior and value the cohesion of the whole protocol. Favoring small, reusable code components promotes code and behavior reuse.

Related Recipes

Recipe 12.4, "Removing One-Use Interfaces"

Recipe 17.14, "Changing Coupling to Classes"

CHAPTER 12

YAGNI

Einstein repeatedly argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.

—Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering*

12.0 Introduction

YAGNI, which stands for "You Ain't Gonna Need It," advises developers to only implement features or functionality that are actually needed at the moment, rather than adding unnecessary features or functionality that may not be used in the future. The idea behind YAGNI is to minimize accidental complexity and keep the focus on the most important tasks at hand.

The YAGNI principle can be seen as a counterpoint to the common tendency in software development to overengineer solutions or add unnecessary features in anticipation of future needs or requirements. This can lead to unnecessary complexity, wasted time and effort, and inflated maintenance costs.

The YAGNI principle encourages developers to keep their focus on the immediate needs of the project and only add features or functionality that are necessary to meet those needs. This helps to keep the project simple and focused and allows developers to be more agile and responsive to changing requirements.

12.1 Removing Dead Code

Problem

You have code that is no longer used or needed.

Solution

Do not keep code "just in case I need it." Remove it.

Discussion

Dead code hurts maintainability and violates the KISS principle (see Recipe 6.2, "Removing Empty Lines") since if the code does not run, nobody maintains it. See the following example with gold-plated code:

```
class Robot {
  walk(){
    //...
    }
  serialize(){
    //...
  }
  persistOnDatabase(database){
    //...
  }
}
```

Gold Plating

Gold plating refers to the practice of adding unnecessary features or functionality to a product or project, beyond the minimum requirements or specifications. This can occur for a variety of reasons, such as a desire to impress the client or to make the product stand out in the market. However, gold plating can be detrimental to the project, as it can lead to cost and schedule overruns, and may not provide any real value to the end user.

Here's a simpler object with proper responsibilities:

```
class Robot {
  walk(){
    // ...
    }
}
```

Test coverage tools can find dead code (uncovered) if you have a great suite of tests. But beware that coverage has problems with metaprogramming (see Chapter 23, "Metaprogramming"). When used, it's very difficult to find references to the code. Remove dead code for simplicity. If you are uncertain of your code, you can temporarily disable it using feature toggles. Removing code is always more rewarding than adding it, and you can always find it in Git history (see Recipe 8.1, "Removing Commented Code").

Related Recipes

Recipe 16.6, "Removing Anchor Boats"

Recipe 23.1, "Removing Metaprogramming Usage"

12.2 Using Code Instead of Diagrams

Problem

You use diagrams to document how software should work.

Solution

Use code and tests as self-living documentation.

Discussion

Most diagrams focus only on structure (accidental) and not behavior (essential). You should only use them to communicate ideas with other humans. Trust your tests. They are alive and well-maintained.

Figure 12-1 presents a sample Unified Modeling Language (UML) diagram. While the diagram is helpful, it is more important that you understand the code and tests instead since the diagram may become obsolete during development; tests cannot lie if you run them continuously.

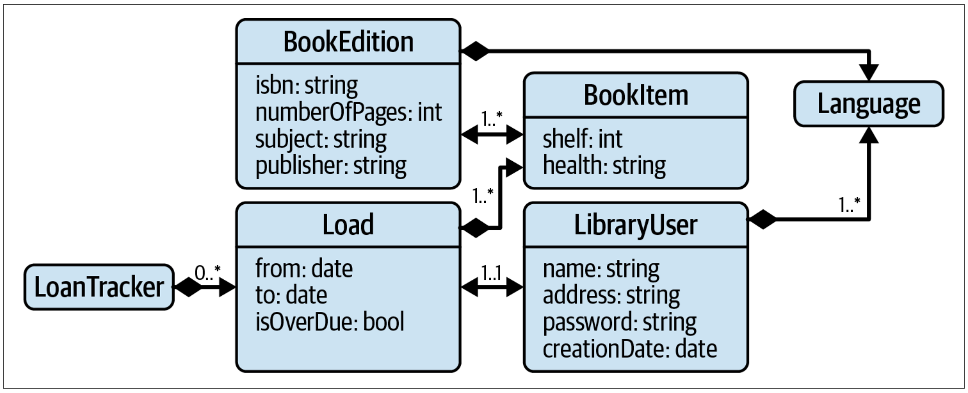


Figure 12-1. A simple UML diagram representing a library

Here is simplified running code from the diagram modeling part of the Library domain:

```
final class BookItem {
    function numberOfPages() { }
    function language(): Language { }
    function book(): Book { }
    function edition(): BookEdition { }
    // Loan and overdues are not book items responsibility
}

final class LoanTracker {
    function loan(
        BookItem $bookCopy,
        LibraryUser $reader,
        DatePeriod $loanDates) {
        // DatePeriod is better than anemic $fromDate and $toDate
    }
}

final class LoanTrackerTests extends TestCase {
    // Lots of maintained tests telling you how the system really works
}
```

Remove all code annotations and forbid them in a well-known policy. Software design is a contact sport, and you need to prototype and learn from your running models. Sheets and JPEGs are static and don't run; they live in a utopian world where everything works smoothly. Some high-level architecture diagrams are useful to understand the whole picture and communicate certain concepts with others.

UML Diagrams

UML (Unified Modeling Language) diagrams are standard visual representations describing the structure and behavior of a software system or application with a common set of symbols and notations. They were trendy in the '80s and '90s and closely related to the waterfall development model where the design is finished before you start the actual coding, as opposed to Agile methodologies. Many organizations still use UML today.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 12.5, "Removing Design Pattern Abuses"

See Also

"Computer-Aided Software Engineering" on Wikipedia

Waterfall Model

According to David Farley, the *waterfall model*, as applied to software development, is a staged, sequential approach to organizing work by breaking it down into a series of distinct phases with well-defined handovers between each phase. The idea is that you tackle each phase in turn, rather than iterate. This was the governing idea until Agile methodologies became more prominent in the '90s.

12.3 Refactoring Classes with One Subclass

Problem

You have a class with just one subclass.

Solution

Don't overgeneralize ahead of time since this is a speculative design; instead, work with the knowledge you've already acquired. Remove the abstract class until you get more concrete examples.

Discussion

In the past, experts often told engineers to design for change. Nowadays, you need to work with real evidence instead. Whenever you find a duplication you remove it, but not before. Here is an example with speculative design:

```
class Boss(object):
    def __init__(self, name):
        self.name = name

class GoodBoss(Boss):
    def __init__(self, name):
        super().__init__(name)

# This is actually a poor classification example
# Bosses should be immutable but can change their mood
# with constructive feedback
```

Here's what it looks like after you compact the hierarchy:

```
class Boss(object):
    def __init__(self, name):
        self.name = name

# Bosses are concrete and can change mood
```

Detection is very easy for linters since they can trace this error at compile time. Subclassing should never be your first option since you need to wait for abstractions to arise and not create them on a speculative basis. A more elegant solution would be to declare an interface if your language has this capability, as it is less coupled. As exceptions, some frameworks create an abstract class as a placeholder to build concrete models over them.

Related Recipes

Recipe 12.4, "Removing One-Use Interfaces"

Recipe 19.3, "Breaking Subclassification for Code Reuse"

Recipe 19.6, "Renaming Isolated Classes"

Recipe 19.7, "Making Concrete Classes Final"

Recipe 19.8, "Defining Class Inheritance Explicitly"

Recipe 19.9, "Migrating Empty Classes"

12.4 Removing One-Use Interfaces

Problem

You have an interface with just one realization.

Solution

Don't overgeneralize until you have more than one example to extract useful and cohesive protocols.

Discussion

Planning interfaces ahead of time and generalizing protocols is a sign of speculative design and overengineering. The following example is trying to determine how a vehicle should behave:

```
public interface Vehicle {
    public void start();
    public void stop();
}

public class Car implements Vehicle {
    public void start() {
        System.out.println("Running...");
    }
    public void stop() {
        System.out.println("Stopping...");
    }
}
// No more concrete vehicles??
```

Since there isn't enough evidence, stick to one realization instead:

```
public class Car {
    public void start() {
        System.out.println("Running...");
    }
    public void stop() {
        System.out.println("Stopping...");
    }
}

// Wait until you discover more concrete vehicles
```

There are a few exceptions to this recipe as this rule applies to business logic. Some frameworks define an interface as a protocol to be fulfilled. On your bijections, you need to model existing real-world protocols. Interfaces are the MAPPER (as defined in Chapter 2) analog to protocols. Also, dependency inversion protocols declare interfaces that are fulfilled with their realizations. Until that point, they can be empty. If your language defines an interface for test mocking, you should consider using Recipe 20.4, "Replacing Mocks with Real Objects". You always need to wait for abstractions and not be unnecessarily creative or speculative.

Dependency Inversion

Dependency inversion is a design principle that decouples higher-level objects from lower-level objects by inverting the traditional dependency relationship. Rather than having higher-level objects depend on lower-level objects directly, the principle suggests that both should depend on abstractions or interfaces. This allows for greater flexibility and modularity in the codebase, as changes to the implementation of a lower-level module do not necessarily require changes to the higher-level module.

Related Recipes

Recipe 7.14, "Removing "Impl" Prefix/Suffix from Class Names"

Recipe 12.3, "Refactoring Classes with One Subclass"

Recipe 20.4, "Replacing Mocks with Real Objects"

12.5 Removing Design Pattern Abuses

Problem

Your code has overdesign symptoms and abuses some design patterns.

Solution

Remove design patterns. Use simpler concepts. Use names based on real-world bijection concepts (essential) instead of implementation pattern names (accidental).

Discussion

The classes in Table 12-1 are named based on the implementation:

Table 12-1. Bad names using patterns

Bad example	Good example
`FileTreeComposite`	`FileSystem`
`DateTimeConverterAdapterSingleton`	`DateTimeFormatter`
`PermutationSorterStrategy`	`BubbleSort`
`NetworkPacketObserver`	`NetworkSniffer`
`AccountsComposite`	`Portfolio`

The five names from Table 12-1 belong to the real world; your mental model will map them 1:1 with familiar objects using the bijection. The hard part of removing patterns is to change the behavior of the objects to avoid the complexity added by a design pattern itself.

Related Recipes

Recipe 7.7, "Renaming Abstract Names"

Recipe 10.4, "Removing Cleverness from Code"

Recipe 12.2, "Using Code Instead of Diagrams"

Recipe 17.2, "Replacing Singletons"

12.6 Replacing Business Collections

Problem

You have specialized collections without additional behavior.

Solution

Don't create unnecessary abstractions. Use a standard library class from your language.

Discussion

Discovering abstractions on the MAPPER is a hard task. You should remove unneeded abstractions unless they add new behavior. This is a word dictionary:

```
Namespace Spelling;

final class Dictionary {

    private $words;
    function __construct(array $words) {
        $this->words = $words;
    }

    function wordCount(): int {
        return count($this->words);
    }

    function includesWord(string $subjectToSearch): bool {
        return in_array($subjectToSearch, $this->words);
    }
}

// This has protocol similar to an abstract data type dictionary
// And the tests

final class DictionaryTest extends TestCase {
    public function test01EmptyDictionaryHasNoWords() {
        $dictionary = new Dictionary([]);
        $this->assertEquals(0, $dictionary->wordCount());
    }

    public function test02SingleDictionaryReturns1AsCount() {
        $dictionary = new Dictionary(['happy']);
        $this->assertEquals(1, $dictionary->wordCount());
    }

    public function test03DictionaryDoesNotIncludeWord() {
        $dictionary = new Dictionary(['happy']);
        $this->assertFalse($dictionary->includesWord('sadly'));
    }

    public function test04DictionaryIncludesWord() {
        $dictionary = new Dictionary(['happy']);
        $this->assertTrue($dictionary->includesWord('happy'));
    }
}
```

You can use a standard class to achieve the same:

```
Namespace Spelling;

// final class Dictionary is no longer needed

// The tests use a standard class
// In PHP you use associative arrays
// Java and other languages have HashTables, Dictionaries, etc.

use PHPUnit\Framework\TestCase;

final class DictionaryTest extends TestCase {
    public function test01EmptyDictionaryHasNoWords() {
        $dictionary = [];
        $this->assertEquals(0, count($dictionary));
    }

    public function test02SingleDictionaryReturns1AsCount() {
        $dictionary = ['happy'];
        $this->assertEquals(1, count($dictionary));
    }

    public function test03DictionaryDoesNotIncludeWord() {
        $dictionary = ['happy'];
        $this->assertFalse(in_array('sadly', $dictionary));
    }

    public function test04DictionaryIncludesWord() {
        $dictionary = ['happy'];
        $this->assertTrue(in_array('happy', $dictionary));
    }
}
```

You can see this as a contradiction with the MAPPER concept, where you need to create objects that exist in the real world to find them in the bijection. The first P from MAPPER comes from Partial. You don't need to model all the real entities. Just the relevant ones.

As an exception, sometimes you need to optimize collections for performance reasons—only if you have enough strong evidence (see Chapter 16, "Premature Optimization"). You need to clean up code from time to time, and specialized collections are a good starting point.

Related Recipes

Recipe 13.5, "Avoiding Modifying Collections While Traversing"

CHAPTER 13

Fail Fast

There is an art to knowing where things should be checked and making sure that the program fails fast if you make a mistake. That kind of choosing is part of the art of simplification.

—Ward Cunningham

13.0 Introduction

The ability to fail fast is critical for clean code. You need to take action as soon as a business rule fails. Every silent failure is a missed improvement opportunity. To accurately debug a problem you need to find the root cause. And the root cause will give you a certain hint to trace and solve the failure. Fail fast systems are more robust than weaker systems where failures are swept under the rug and processing keeps going forward, even after the failure affected the correct result.

13.1 Refactoring Reassignment of Variables

Problem

You reuse variables with different scopes.

Solution

Don't reuse variable names. You break readability and refactor chances and gain nothing; this is a premature optimization that does not save memory. Narrow the scopes as much as possible.

Discussion

If you reuse a variable and extend its scope, automatic refactoring tools may break and your virtual machine might miss the chance to make optimizations. It is recommended that you define, utilize, and dispose of variables while keeping their lifecycle short. In this example, there are two unrelated purchases:

```
class Item:
  def taxesCharged(self):
    return 1;

lastPurchase = Item('Soda');
# Do something with the purchase

taxAmount = lastPurchase.taxesCharged();
# Lots of stuff related to the purchase
# You drink the soda

# You cannot extract method from below without passing
# useless lastPurchase as parameter

# a few hours later…
lastPurchase = Item('Whisky'); # You bought another drink

taxAmount += lastPurchase.taxesCharged();
```

Here's what it looks like if you narrow the scope:

```
class Item:
  def taxesCharged(self):
    return 1;

def buySupper():
  supperPurchase = Item('Soda');
  # Do something with the purchase

  # Lots of stuff related to the purchase
  # You drink the soda
  return supperPurchase;

def buyDrinks():
  # You can extract the method!

  # a few hours later..
  drinksPurchase = Item('Whisky');
  # I bought another drink

  return drinksPurchase;

taxAmount = buySupper().taxesCharged() + buyDrinks().taxesCharged();
```

Reusing variables is also referred to as a "noncontextual copy-and-paste" hint.

Related Recipes

Recipe 11.1, "Breaking Too Long Methods"

Virtual Machine Optimization

Most modern programming languages today run on *virtual machines* (VMs). They abstract hardware details and make many *optimizations* under the hood so you can focus on making code readable and avoid premature optimization (see Chapter 16, "Premature Optimization"). Writing clever performant code is almost never necessary since they solve many performance issues. In Chapter 16 you discover how to collect real evidence in order to determine if you need to optimize the code.

13.2 Enforcing Preconditions

Problem

You want to create more robust objects using business preconditions, postconditions, and invariants.

Solution

Turn on assertions both in development and production unless you have strong evidence that it's a significant performance problem.

Discussion

Object consistency is key to keeping up with the MAPPER (as defined in Chapter 2). You need to provide a warning as soon as your code breaks a software contract. It is easier to debug a problem as soon as it happens. As always, constructors are an excellent first line of defense.

Design by Contract

Object-Oriented Software Construction by Bertrand Meyer is a comprehensive guide to software development using the object-oriented paradigm. One of the key ideas of the book is the concept of "design by contract," which emphasizes the importance of creating clear and unambiguous contracts between software modules. A contract specifies the responsibilities and behavior that ensure that modules work correctly together and that software remains reliable and maintainable over time. When a contact is broken, the fail fast principle is honored and issues get noticed immediately.

Here is a familiar example of `Date` validation:

```
class Date:
  def __init__(self, day, month, year):
    self.day = day
    self.month = month
    self.year = year

  def setMonth(self, month):
    self.month = month

startDate = Date(3, 11, 2020)
# OK

startDate = Date(31, 11, 2020)
# Should fail, but it doesn't

startDate.setMonth(13)
# Should fail, but it doesn't
```

Here's what it looks like when you put the controls even before creating the `Date`:

```
class Date:
  def __init__(self, day, month, year):
    if month > 12:
        raise Exception("Month should not exceed 12")
    #
    # etc ...

    self._day = day
    self._month = month
    self._year = year

startDate = Date(3, 11, 2020)
# OK

startDate = Date(31, 11, 2020)
# fails

startDate.setMonth(13)
# fails since invariant makes object immutable
```

You should always be explicit about object integrity and turn on production assertions even if it brings minor performance penalties. Data and object corruption are harder to find, so failing fast is a blessing.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 25.1, "Sanitizing Inputs"

See Also

Object-Oriented Software Construction by Bertrand Meyer

Preconditions, Postconditions, and Invariants

A *precondition* is a condition that must be true before a function or method is called. It specifies the requirements that the inputs of the function or method must satisfy. An *invariant* is a condition that must hold true at all times during the execution of a program, regardless of any changes that may occur. It specifies a property of the program that should not change over time. Finally, a *postcondition* is tied to the time after the method is called. You can use them to ensure correctness, detect defects, or guide program design.

13.3 Using Stricter Parameters

Problem

You have magic functions that can receive a lot of different (and nonpolymorphic) arguments.

Solution

Create a clear contract. Expect just one protocol.

Discussion

A function signature is very important, and languages favoring magic castings are false friends. They promise easy fixes, but you will soon encounter yourself debugging unexpected values. Instead of creating a flexible function polluted with if conditions, you should always accept one kind of argument adhering to a single protocol.

Function Signature

A *function signature* specifies its name, parameter types, and return type if the language is strictly typed. It is used to distinguish one function from another and to ensure that function calls are made correctly.

In this example, you can receive several different and nonpolymorphic arguments:

```
function parseArguments($arguments) {
    $arguments = $arguments ?: null;
    // Always the billion-dollar mistake (null)
    if (is_empty($arguments)) {
        $this->arguments = http_build_query($_REQUEST);
        // Global coupling and side effects
    } elseif (is_array($arguments)) {
        $this->arguments = http_build_query($arguments);
    } elseif (!$arguments) { // null unmasked
        $this->arguments = null;
    } else {
        $this->arguments = (string)$arguments;
    }
}
```

Here is a canonical single solution:

```
function parseArguments(array $arguments) {
    $this->arguments = http_build_query($arguments);
}
```

Magic castings and flexibility have a price. They sweep the rubbish under the rug and violate the fail fast principle.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 15.1, "Creating Null Objects"

Recipe 24.2, "Dealing with Truthy Values"

13.4 Removing Default from Switches

Problem

You do not throw an exception when you actually should in switch statements.

Solution

Don't add a `default` clause to your `case` statements. Change it to an exception to be explicit and avoid speculative solutions.

Discussion

"Default" means "everything you don't know yet." You cannot foresee the future, so the default should be an exception for "unforeseen cases." When using cases, you often add a default case so it doesn't fail. But failing is always better than making decisions without evidence. Since `case` as well as `switch` are often a problem, you can avoid them using Recipe 14.4, "Replacing Switch/Case/Elseif Statements". In C and C++, the default case is optional. In Java, the default case is mandatory, and the compiler generates an error if it is omitted. In C#, the default case is optional, but the compiler generates a warning if it is omitted.

Here is a speculative default `case`:

```
switch (value) {
  case value1:
    // if value1 matches, the following will be executed...
    doSomething();
    break;
  case value2:
    // if value2 matches, the following will be executed...
    doSomethingElse();
    break;
  default:
    // if the value does not presently match the above values
    // or future values
    // the following will be executed
    doSomethingSpecial();
    break;
}
```

Here's what it looks like when you replace it with an exception:

```
switch (value) {
  case value1:
    // if value1 matches the following will be executed...
    doSomething();
    break;
  case value2:
    // if value2 matches the following will be executed...
    doSomethingElse();
    break;
  case value3:
  case value4:
    // You currently know these options exist
    doSomethingSpecial();
    break;
  default:
    // if value does not match the above values you need to make a decision
    throw new Exception('Unexpected case ' + value + ', need to consider it');
    break;
}
```

You can tell your linters to warn you on using default unless there's an exception, because writing robust code doesn't mean you need to make decisions without evidence. Since there are many valid usages, you need to be aware of false positives.

Related Recipes

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

13.5 Avoiding Modifying Collections While Traversing

Problem

You are changing a collection while traversing it.

Solution

Do not modify collections while traversing them since it can create inconsistencies for internal pointers.

Discussion

Some developers tend to over-optimize their solutions, assuming that copying collections is a costly operation. This is not true for small- and medium-sized collections, and programming languages iterate collections in many different ways (see Chapter 16, "Premature Optimization"). Modifying them during an iteration is generally not safe, and the consequences can appear far from the traversing code.

Here is an example with erratic consequences:

```
// Here you add elements to the collection...
Collection<Object> people = new ArrayList<>();

for (Object person : people) {
    if (condition(person)) {
        people.remove(person);
    }
}
// You iterate AND remove elements, elements,
// risking skipping other candidates for removal
```

Here's what it looks like when you change the code to a safer solution by copying the collection:

```
// Here you add elements to the collection...
Collection<Object> people = new ArrayList<>();

List<Object> iterationPeople = ImmutableList.copyOf(people);

for (Object person : iterationPeople) {
    if (condition(person)) {
        people.remove(person);
    }
}
// You iterate a copy and remove it from the original

people.removeIf(currentIndex -> currentIndex == 5);
// Or use language tools (if available)
```

This is something developers learn early on, but it still happens a lot in the industry and real-world software.

Related Recipes

Recipe 6.6, "Replacing Explicit Iterations"

Recipe 12.6, "Replacing Business Collections"

13.6 Redefining Hash and Equality

Problem

You implement hashing on your objects but do not redefine equality.

Solution

If you check for the *hash*, you should also check for *equality* for consistency.

Discussion

Hashing guarantees two objects are different when they have a different hash value but not that they are the same if they have the same hash value; this asymmetry can cause a bijection fault. You should always check for hash (fast) and then check for equality (slower).

Here is an example of `Person` comparison on large collections:

```
public class Person {

 public String name;
```

```
  @Override
  public boolean equals(Person anotherPerson) {
    return name.equals(anotherPerson.name);
  }

 @Override
  public int hashCode() {
    return (int)(Math.random()*256);
  }
 }
```

When using a hashmap, you can make a mistake and guess the object is not present in the collection. Here's what it looks like when you also redefine the hash:

```
public class Person {

public String name;
   // Having public attributes is another problem
   @Override
   public boolean equals(Person anotherPerson) {
     return name.equals(anotherPerson.name);
   }

  @Override
   public int hashCode() {
     return name.hashCode();
   }
}
```

Many linters have rules for hash and equality redefinition using static analysis and parse trees. With mutation testing (see Recipe 5.1, "Changing var to const"), you can seed different objects with the same hash and check your tests. Every performance improvement has its drawbacks, and that's the reason why you should always tune performance after your code works and has automated functional tests covering it.

Related Recipes

Recipe 14.15, "Changing Equal Comparison"

Recipe 16.7, "Extracting Caches from Domain Objects"

Hashing

Hashing refers to the process of mapping data of arbitrary size to a fixed-size value. The output of a hash function is called a hash value or hash code. You can use hash values as an index table in large collections; they work as a shortcut to find elements in a more performant way than iterating the elements sequentially.

13.7 Refactoring Without Functional Changes

Problem

You develop and refactor at the same time.

Solution

Don't change functionality and refactor at the same time.

Discussion

Mixing refactors and functional changes makes code reviews more difficult and creates possible merge conflicts. Sometimes you detect that refactoring is needed for further development; if this is the case, put your solution on hold. Work on the refactoring and continue with your solution after that.

Here is a simple refactoring and a change of functionality at the same time:

```
getFactorial(n) {
  return n * getFactorial(n);
}

// Rename and change

factorial(n) {
  return n * factorial(n-1);
}

// This is a very small example
// Things go worse when dealing with more code
```

By dividing the modifications, you can enhance their clarity:

```
getFactorial(n) {
  return n * getFactorial(n);
}
// Change

getFactorial(n) {
  return n * getFactorial(n-1);
}
// Run the tests

factorial(n) {
  return n * factorial(n-1);
}
// Rename
```

You can use a physical token as a reminder. Either you are in the refactoring stage or the developing stage.

CHAPTER 14
Ifs

Make the change easy (warning: this might be hard), then make the easy change.

—Kent Beck on Twitter

14.0 Introduction

Using GoTos was a well-established practice until Edsger Dijkstra wrote his incredible paper: "Go To Statement Considered Harmful." Nowadays nobody uses the GoTo instruction (see Recipe 18.3, "Replacing GoTo with Structured Code") and few programming languages still support it, because it creates spaghetti code, which is unmaintainable and error-prone. Structured programming solved the spaghetti code problem years ago.

Spaghetti Code

Spaghetti code is poorly structured code that is difficult to understand and maintain. The name "spaghetti" is used because the code is often tangled and interconnected in a way that resembles a plate of tangled spaghetti noodles. It contains redundant or duplicated code, as well as numerous conditional statements, jumps, and loops that can be difficult to follow.

The next evolution will be removing most if statements because ifs/cases and switches are GoTos disguised as structured flow. Both GoTos and ifs are present in low-level machine programming languages like Assembler.

Most if statements are coupled with *accidental decisions*. This coupling generates a ripple effect and makes code harder to maintain since accidental ifs are considered as harmful as GoTos because they violate the open/closed principle (see Recipe 14.3, "Reifying Boolean Variables"). Your designs will become less extensible. If statements open the door to even more serious problems, like `switches`, `cases`, `defaults`, `return`, `continue`, and `breaks`. They make your algorithms darker and force you to build *accidentally complex* solutions.

Structured Programming

Structured programming emphasizes the use of control flow constructs, such as loops and functions, to improve the clarity, maintainability, readability, and reliability of computer programs. You break down a program into smaller, manageable pieces, and then organize those pieces using structured control flow constructs.

14.1 Replacing Accidental Ifs with Polymorphism

Problem

You have accidental ifs within your code.

Solution

Replace them with polymorphic objects.

Discussion

Here's an *essential* if statement:

```
class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchXRatedMovie() {
    if (this.age < 18)
      throw new Error("You are not allowed to watch this movie");
    else
      this.watchMovie();
  }
  watchMovie() {
    // ..
  }
}

const jane = new MovieWatcher(12);

jane.watchXRatedMovie();
// Throws exception since Jane is too young to watch the movie
```

You need to decide whether to remove this if sentence or not, and you need to understand whether it represents a business rule (*essential*) or an implementation artifact (*accidental*). People outside the software world in the real world describe age constraints in natural language using ifs. Therefore, you need to honor the bijection, identify it as essential, and NOT replace it.

This example has an accidental if:

```
class Movie {
  constructor(rate) {
    this.rate = rate;
  }
}

class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchMovie(movie) {
    if ((this.age < 18) && (movie.rate === 'Adults Only'))
      throw new Error("You are not allowed to watch this movie");
    // if the exception is not raised you can watch the movie
    playMovie();
  }
}

const jane = new MovieWatcher(12);
const theExorcist = new Movie('Adults Only');

jane.watchMovie(theExorcist);
// Jane cannot watch the exorcist since she is 12
```

The movie rating if is not related to a real-world if but to accidental (and coupled) implementation. The problem is the design decision to model ratings with strings (see Chapter 3, "Anemic Models"). This is a classic *neither open-to-extension nor closed-to-modification* solution.

The problem worsens with new requirements:

```
class Movie {
  constructor(rate) {
    this.rate = rate;
  }
}

class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchMovie(movie) {
    // !!!!!!!!!!!!!!!!! IFS ARE POLLUTING HERE !!!!!!!!!!!!!!!!!!!!!!!!!!
    if ((this.age < 18) && (movie.rate === 'Adults Only'))
```

```
        throw new Error("You are not allowed to watch this movie");
      else if ((this.age < 13) && (movie.rate === 'PG 13'))
        throw new Error("You are not allowed to watch this movie");
      // !!!!!!!!!!!!!!!! IFS ARE POLLUTING HERE !!!!!!!!!!!!!!!!!!!!!!!!!!!

      playMovie();
    }
  }

  const theExorcist = new Movie('Adults Only');
  const gremlins = new Movie('PG 13');

  const jane = new MovieWatcher(12);

  jane.watchMovie(theExorcist);
  // Jane cannot watch the exorcist since she is 12
  jane.watchMovie(gremlins);
  // Jane cannot watch Gremlins since she is 12

  const joe = new MovieWatcher(16);

  joe.watchMovie(theExorcist);
  // Joe cannot watch The Exorcist since he is 16
  joe.watchMovie(gremlins);
  // Joe CAN watch Gremlins since he is 16
```

The code is polluted with ifs since new ratings will bring new ifs and a default statement is also missing. The strings used to represent ratings are not first-class objects. A typo will introduce hard-to-find errors and you are forced to add getters on `Movie` to make decisions.

This example creates a polymorphic hierarchy for every if condition (if it doesn't already exist), moves every if body to the former abstraction, and replaces the if calls with a single polymorphic method call:

```
// 1. Create a polymorphic hierarchy for every if condition
// (if it doesn't already exist)
class MovieRate {
  // If language permits this should be declared abstract
}

class PG13MovieRate extends MovieRate {
  //2. Move every *if body* to the former abstraction
  warnIfNotAllowed(age) {
    if (age < 13)
      throw new Error("You are not allowed to watch this movie");
  }
}

class AdultsOnlyMovieRate extends MovieRate {
  //2. Move every *if body* to the former abstraction
  warnIfNotAllowed(age) {
    if (age < 18)
      throw new Error("You are not allowed to watch this movie");
  }
}

class Movie {
  constructor(rate) {
    this.rate = rate;
  }
}

class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchMovie(movie) {
    // 3. Replace if calls by a polymorphic method call
    movie.rate.warnIfNotAllowed(this.age);
    // watch movie
  }
}

const theExorcist = new Movie(new AdultsOnlyMovieRate());
const gremlins = new Movie(new PG13MovieRate());

const jane = new MovieWatcher(12);

// jane.watchMovie(theExorcist);
// Jane cannot watch the exorcist since she is 12
// jane.watchMovie(gremlins);
// Jane cannot watch gremlins since she is 12

const joe = new MovieWatcher(16);

// joe.watchMovie(theExorcist);
// Joe cannot watch the exorcist since he is 16
joe.watchMovie(gremlins);
// Joe CAN watch gremlins since he is 16
```

This solution is better since the code is no longer polluted with ifs, and extending the model will be enough when you get new requirements and learn from the domain. If you need to create new ratings, you will address it with polymorphic new instances. Also, a default behavior is not needed since exceptions break flow. Many times a *null object* (see Recipe 15.1, "Creating Null Objects") will be enough. The ratings are now first-class objects; you don't have the chance for typo problems from the previous example. The essential ifs are still there (checking for an age) and the accidental ones are gone (rate limits).

To address the collaborator chain problem, you can break the code:

```
movie.rate.warnIfNotAllowed(this.age);

class Movie {
  constructor(rate) {
    this._rate = rate; // Rate is now private
  }
  warnIfNotAllowed(age) {
    this._rate.warnIfNotAllowed(age);
  }
}

class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchMovie(movie) {
    movie.warnIfNotAllowed(this.age);
    // watch movie
  }
}
```

The rating is private, so you don't break encapsulation and don't need getters. Here's what it looks like if you apply the recipe to essential ifs:

```
class Age {
}

class AgeLessThan13 extends Age {
  assertCanWatchPG13Movie() {
    throw new Error("You are not allowed to watch this movie");
  }
  assertCanWatchAdultMovie() {
    throw new Error("You are not allowed to watch this movie");
  }
}

class AgeBetween13And18 extends Age {
  assertCanWatchPG13Movie() {
    // No problem
  }
  assertCanWatchAdultMovie() {
    throw new Error("You are not allowed to watch this movie");
  }
}

class MovieRate {
  // If language permits this should be declared abstract
  // abstract assertCanWatch();
}

class PG13MovieRate extends MovieRate {
  // Move every *if body* to the former abstraction
  assertCanWatch(age) {
    age.assertCanWatchPG13Movie()
  }
}

class AdultsOnlyMovieRate extends MovieRate {
  // Move every *if body* to the former abstraction
  assertCanWatch(age) {
     age.assertCanWatchAdultMovie()
  }
}

class Movie {
  constructor(rate) {
    this._rate = rate; // Rate is now private
  }
  watchByMe(moviegoer) {
    this._rate.assertCanWatch(moviegoer.age);
  }
}

class MovieWatcher {
  constructor(age) {
    this.age = age;
  }
  watchMovie(movie) {
    movie.watchByMe(this);
  }
}

const theExorcist = new Movie(new AdultsOnlyMovieRate());
const gremlins = new Movie(new PG13MovieRate());

const jane = new MovieWatcher(new AgeLessThan13());

// jane.watchMovie(theExorcist);
// Jane cannot watch the exorcist since she is 12
// jane.watchMovie(gremlins);
// Jane cannot watch gremlins since she is 12

const joe = new MovieWatcher(new AgeBetween13And18());

// joe.watchMovie(theExorcist);
// Joe cannot watch the exorcist since he is 16
joe.watchMovie(gremlins);
// Joe CAN watch gremlins since he is 16
```

This code works but it has overdesign symptoms as classes representing ages are not related to real concepts on your model, thus violating the bijection principle. Also, the model is too complex because you will need new classes related to new age groups and age groups might not be disjoint.

To avoid the last design and set a clear boundary between *essential* and *accidental* ifs, you can use the following rule:

A good design rule is to create abstractions if the elements belong to the same domain (movies and ratings) and not create them if they cross domains (movies and ages).

This recipe recommends you avoid most if sentences. This might be hard for you, especially at the beginning, since the usage of conditionals is very rooted in the profession and you might be very comfortable with this practice. However, it's possible to remove all accidental ifs. This will make your models less coupled and more extensible. The null object pattern is a special case of this technique. You will be able to remove all nulls since null ifs are *always* accidental (see Chapter 15, "Null").

Polymorphic Hierarchy

In a *polymorphic hierarchy*, classes are organized in a hierarchical structure based on their "behaves-as-a" relationships. This allows for the creation of specialized classes that inherit behaviors from more general classes. In a polymorphic hierarchy, a base abstract class serves as the foundation and defines common behavior shared by multiple concrete subclasses. The subclasses inherit these characteristics from the superclass and can add their own behavior. Subclassification is one way to enforce polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic"). But it is a rigid one since you cannot change a superclass after compile time.

14.2 Renaming Flag Variables for Events

Problem

You have flag arguments (booleans) with vague names in your functions.

Solution

Rename flag variables to show what happened.

Discussion

Flags indicate what happened. Sometimes their name is too generic. Here is a flag example showing that something has happened:

```
function dummy() {

    $flag = true;

    while ($flag == true) {
        $result = checkSomething();
        if ($result) {
            $flag = false;
        }
    }
}
```

This solution is more declarative and intention-revealing:

```
function dummy()
{
    $atLeastOneElementWasFound = false;

    while (!$atLeastOneElementWasFound) {

        $elementSatisfies = checkSomething();
        if ($elementSatisfies) {
            $atLeastOneElementWasFound = true;
        }
    }
}
```

You should search the entire code for badly named flags. Flags are widespread on production code. You should restrict their usage and enforce clear and intention-revealing names.

Related Recipes

Recipe 6.4, "Removing Double Negatives"

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 14.3, "Reifying Boolean Variables"

14.3 Reifying Boolean Variables

Problem

You have code using boolean variables as flags, exposing accidental implementation and polluting the code with if conditions.

Boolean Flags

A *boolean flag* is a variable that can only be either true or false, representing the two possible states of a binary condition. Boolean flags are commonly used to control the flow of logic through conditional statements, loops, and other control structures.

Solution

Don't use boolean variables since they force you to write ifs (see Recipe 14.1, "Replacing Accidental Ifs with Polymorphism"). Create polymorphic states instead.

Discussion

Boolean variables break extensibility and the open-closed principle from SOLID (see Recipe 19.1, "Breaking Deep Inheritance"). They are difficult to compare in some languages that convert all values to truthy and falsy (see Recipe 24.2, "Dealing with Truthy Values"). If the boolean maps to a real-world boolean entity you should create it following the MAPPER as defined in Chapter 2. Otherwise, you can model it using the *state design pattern* to favor extensibility.

Open-Closed Principle

The *open-closed principle* is the "O" from SOLID (see Recipe 19.1, "Breaking Deep Inheritance"). It states that software classes should be open for extension but closed for modification. You should be able to extend the behavior without modifying the code. This principle encourages the use of abstract interfaces, inheritance, and polymorphism to allow new functionality to be added without changing existing code. The principle also promotes the separation of concerns (see Recipe 8.3, "Removing Logical Comments"), making it easier to develop, test, and deploy software components independently.

Here's an example with three boolean flags:

```
function processBatch(
    bool $useLogin,
    bool $deleteEntries,
    bool $beforeToday) {
    // ...
}
```

You can reify it into real-world concepts as follows:

```
function processBatch(
    LoginStrategy $login,
    DeletionPolicy $deletionPolicy,
    Date $cutoffDate) {
    // ...
}
```

Automatic detection can warn you about boolean usage, but this can yield false positives, and some languages have issues with boolean comparators. In languages with truthy and falsy values like JavaScript (see Recipe 24.2, "Dealing with Truthy Values"), booleans are a common error source, and you should take extra care when declaring something as a boolean. Flags are difficult to maintain and extend. Learn more about the domain and use polymorphism instead of ifs/switch/cases (see Recipe 14.1, "Replacing Accidental Ifs with Polymorphism").

State Design Pattern

The *state design pattern* allows an object to change its behavior when its internal state changes at runtime without changing its class. You define a set of valid state objects encapsulating each state's behavior within a separate class. The state objects expose a common interface that allows the context object to delegate its behavior to the appropriate state object. When the context object's state changes, it simply switches to the appropriate state object. It promotes loose coupling between the context object and its state objects and enables the context object to be more flexible and extensible, favoring the open-closed principle.

Related Recipes

Recipe 6.4, "Removing Double Negatives"

Recipe 14.2, "Renaming Flag Variables for Events"

See Also

"FlagArgument" by Martin Fowler

14.4 Replacing Switch/Case/Elseif Statements

Problem

You have control structures with switches and cases.

Solution

Replace them with polymorphic objects.

Discussion

Switches combine too many decisions together and break the open-closed principle (see Recipe 14.3, "Reifying Boolean Variables") since every new condition will change the main algorithm, favoring merge conflicts. They also cause duplicated code and very large methods. You should create hierarchies/compose objects following the open-closed principle using the state pattern to model transitions and the strategy pattern/method object to choose branches.

The following example converts several different audio formats to MP3:

```
class Mp3Converter {
  convertToMp3(source, mimeType) {
    if(mimeType.equals("audio/mpeg")) {
        this.convertMpegToMp3(source)
    } else if(mimeType.equals("audio/wav")) {
        this.convertWavToMp3(source)
    } else if(mimeType.equals("audio/ogg")) {
        this.convertOggToMp3(source)
    } else if(...) {
        // Lots of new else clauses
}
```

This is equivalent to the same problematic code:

```
class Mp3Converter {
  convertToMp3(source, mimeType) {
  switch (mimeType) {
    case "audio/mpeg":
      this.convertMpegToMp3(source);
      break;
    case "audio/wav":
      this.convertWavToMp3(source);
      break;
    case "audio/ogg":
      this.convertOggToMp3(source);
      break;
   default:
      throw new Error("Unsupported MIME type: " + mimeType);
  }
}
```

Having specialized converters is not a problem since you can add a new one without changing the code:

```
class Mp3Converter {
  convertToMp3(source, mimeType) {
    const foundConverter = this.registeredConverters.
        find(converter => converter.handles(mimeType));
        // Do not use metaprogramming to find and iterate converters
        // since this is another problem.
    if (!foundConverter) {
      throw new Error('No converter found for ' + mimeType);
    }
    foundConverter.convertToMp3(source);
  }
}
```

Since there are valid cases for if/else usage, you should not pull the plug and forbid these instructions. You can put a ratio of if statements/other statements as a warning instead.

Related Recipes

Recipe 10.7, "Extracting a Method to an Object"

Recipe 13.4, "Removing Default from Switches"

Recipe 14.5, "Replacing Hardcoded If Conditions with Collections"

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 15.1, "Creating Null Objects"

Strategy Design Pattern

The *strategy design pattern* defines a family of interchangeable algorithms, encapsulates each one, and makes them interchangeable at runtime. The pattern allows a client object to choose from a range of algorithms to use, based on the specific context or situation at runtime. It also promotes loose coupling between the client object and the strategies and makes it easier for you to extend or modify the behavior of the client object without affecting its implementation.

14.5 Replacing Hardcoded If Conditions with Collections

Problem

You have hardcoded if conditions.

Solution

Create a collection to map your conditions.

Discussion

Hardcoded if conditions break testability. You can replace all ifs with a dynamic condition or polymorphism. Here is an example mapping internet domains with country names:

```
private string FindCountryName (string internetCode)
{
  if (internetCode == "de")
    return "Germany";
  else if(internetCode == "fr")
    return "France";
  else if(internetCode == "ar")
    return "Argentina";
    // lots of else clauses
  else
    return "Suffix not Valid";
}
```

You can create two collections to map them or a single one:

```
private string[] country_names = {"Germany", "France", "Argentina"}; // lots more
private string[] Internet_code_suffixes= {"de", "fr", "ar" }; // more
// You also can do inline initialization here

private Dictionary<string, string> Internet_codes =
 new Dictionary<string, string>();

// There are more efficient ways for collection iteration
// This pseudocode is only for illustration
int currentIndex = 0;
foreach (var suffix in Internet_code_suffixes) {
  Internet_codes.Add(suffix, Internet_codes[currentIndex]);
  currentIndex++;
}

private string FindCountryName(string internetCode) {
  return Internet_codes[internetCode];
}
```

In the past, hardcoding was not an option. With modern methodologies, you learn by hardcoding, and then you generalize and refactor your solutions.

Related Recipes

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 14.16, "Reifying Hardcoded Business Conditions"

14.6 Changing Boolean to Short-Circuit Conditions

Problem

You have a full boolean evaluation but you know the result before finishing it.

Solution

Be lazy when evaluating boolean conditions. Use a short circuit if your language allows it.

Discussion

Boolean truth tables are great for mathematics, but you need to be cautious since short-circuit evaluation sometimes has side effects and performance issues. Short-circuit evaluation helps you avoid building invalid full evaluations, such as full evaluation even when the value is already defined (for example using an OR condition where the first part is true).

Here is an invalid full evaluation using logical AND (&):

```
if (isOpen(file) & size(contents(file)) > 0)
  // It performs a full evaluation since it is the bitwise AND
  // will fail since you cannot retrieve contents
  // from a file that is not open
```

This version uses short-circuit evaluation:

```
if (isOpen(file) && size(contents(file)) > 0)
  // Short-circuit evaluation
  // If the file is not open it will not try to get the contents
```

As an exception, you shouldn't use a short circuit as an if alternative. If the operands have side effects—since most programming languages support short circuits and many of them have it as the only option—you need to favor this kind of expression.

Related Recipes

Recipe 14.9, "Avoiding Short-Circuit Hacks"

Recipe 14.12, "Changing Comparison Against Booleans"

Recipe 24.2, "Dealing with Truthy Values"

14.7 Adding Implicit Else

Problem

You have an if statement without an else.

Solution

Be explicit, even with the else guard, and put it next to your if condition.

Discussion

Code with explicit else guards is more readable and has a smaller cognitive load. It can also help you to detect unforeseen conditions and to favor the fail fast principle (see Chapter 13, "Fail Fast"). If you do an early return on an if sentence, you can omit the else part and then remove the if and use polymorphism. That is when you miss the real cases.

Here is a function with an implicit else:

```
function carBrandImplicit(model) {
  if (model === 'A4') {
    return 'Audi';
  }
  return 'Mercedes-Benz';
}
```

Here's what it looks like when you make it explicit:

```
function carBrandExplicit(model) {
  if (model === 'A4') {
    return 'Audi';
  }
  if (model === 'AMG') {
    return 'Mercedes-Benz';
  }

  // Fail Fast
  throw new Exception('Model not found');
}
```

You can also rewrite them and perform mutation testing (see Recipe 5.1, "Changing var to const"). This recipe evokes a lot of public debate and strong opinions. Listen to all the opinions and then evaluate all the pros and cons.

Related Recipes

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 14.10, "Rewriting Nested Arrow Code"

14.8 Rewriting Conditional Arrow Code

Problem

You have nested boolean conditions resembling a stair or arrow.

Solution

Avoid checking for boolean expressions and returning an explicit boolean. Replace it with a formula value.

Discussion

Stairs code, or arrow code, is hard to read. The scope is also hard to match from the beginning to the end. This type of code can also be classified as ninja code, which is more common in low-level languages. When dealing with boolean formulas, it is more readable to show a business boolean formula than a stair of boolean checks followed by returning an explicit true/false.

Here's an example of arrow code:

```
def is_platypus(self):
    if self.is_mammal():
        if self.has_fur():
            if self.has_beak():
                if self.has_tail():
                    if self.can_swim():
                        return True
    return False

# This is also wrong since it is polluted
# with IFs and not readable by a biologist
def is_platypus(self):
    if not self.is_mammal():
        return False
    if not self.has_fur():
        return False
    if not self.has_beak():
        return False
    if not self.has_tail():
        return False
    if not self.can_swim():
        return False
    return True
```

Here's what it looks like when you rewrite the condition:

```
def is_platypus(self):
    return self.is_mammal() &&
        self.has_fur() &&
            self.has_beak() &&
                self.has_tail() &&
                    self.can_swim()

# You can even group conditions according to animal taxonomies
```

Based on syntax trees, you can safely refactor the code removing the explicit boolean value. Beware of returning booleans. After the return, you will need an if statement that you might remove using the applicable recipe.

Ninja Code

Ninja code (also known as clever code or smart code) refers to code that is cleverly written, but difficult to understand or maintain. It is often created by experienced programmers who enjoy using advanced programming techniques or specific language features to write more efficient and prematurely optimized code. While ninja code can be impressive and may run faster than other code, it can be difficult to read and comprehend, leading to problems with maintainability, scalability, and future development. Ninja code is the opposite of clean code.

Related Recipes

Recipe 14.2, "Renaming Flag Variables for Events"

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 14.12, "Changing Comparison Against Booleans"

Recipe 22.4, "Rewriting Nested Try/Catches"

Recipe 22.6, "Rewriting Exception Arrow Code"

Recipe 24.2, "Dealing with Truthy Values"

14.9 Avoiding Short-Circuit Hacks

Problem

You use boolean evaluation as a readability shortcut for a second condition that is only valid if the first one passed.

Solution

Don't use boolean comparison for side-effect functions. Rewrite it as an if.

Discussion

Clever programmers like to write hacky and obscure code even when there is no strong evidence to support this "improvement." Writing dependent conditions as boolean is a sign of premature optimization (see Chapter 16) and damages readability.

The following example combines a condition and a consequence of the first condition:

```
userIsValid() && logUserIn();

// This expression is short-circuited
// Does not value second statement
// unless the first one is true

functionDefinedOrNot && functionDefinedOrNot();

// In some languages undefined works as a false
// If functionDefinedOrNot is not defined, this does
// not raise an error or run
```

You can change both cases using more declarative ifs:

```
if (userIsValid()) {
    logUserIn();
}

if(typeof functionDefinedOrNot == 'function') {
    functionDefinedOrNot();
}
// Checking for a typeOf is not a good solution
```

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 14.6, "Changing Boolean to Short-Circuit Conditions"

Recipe 15.2, "Removing Optional Chaining"

14.10 Rewriting Nested Arrow Code

Problem

You have nested ifs and elses that are very hard to read and test.

Solution

Avoid nested ifs and try to avoid all accidental ifs.

Discussion

In procedural code, it is very common to see complex nested ifs. This solution is more related to scripting than object-oriented programming. Here's an example of arrow or stairs code:

```
if (actualIndex < totalItems)
    {
      if (product[actualIndex].Name.Contains("arrow"))
      {
        do
        {
          if (product[actualIndex].price == null)
          {
            // handle no price
          }
          else
          {
            if (!(product[actualIndex].priceIsCurrent()))
            {
              // add price
            }
            else
            {
              if (hasDiscount)
              {
                // handle discount
              }
              else
              {
                // etc
              }
            }
          }
          actualIndex++;
        }
        while (actualIndex < totalCount && totalPrice < wallet.money);
      }
      else
        actualIndex++;
    }
    return actualIndex;
  }
```

You can refactor it as follows:

```
foreach (products as currentProduct) {
  addPriceIfDefined(currentProduct)
}
addPriceIfDefined()
{
  // You only add the price if it follows the above rules
}
```

Since many linters can parse syntax trees, you can check at compile time for nesting levels.

Related Recipes

Recipe 6.13, "Avoiding Callback Hell"

Recipe 11.1, "Breaking Too Long Methods"

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 14.8, "Rewriting Conditional Arrow Code"

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 14.18, "Rewriting Nested Ternaries"

Recipe 22.6, "Rewriting Exception Arrow Code"

See Also

"Flattening Arrow Code" at the Coding Horror blog

14.11 Preventing Return Boolean Values for Condition Checks

Problem

You have code checking (and probability casting) for a boolean condition.

Solution

Don't return explicit booleans. Most boolean usages don't map to real-world booleans in the MAPPER. Rewrite it to a business condition.

Discussion

Most boolean returns are coupled to accidental solutions and don't actually correspond to a real-world boolean. You need to write high-level code and solutions to favor the bijection rule and many booleans don't. You can also return a boolean proposition instead of checking a negation and create answers with a business logic formula, not an algorithm. When dealing with boolean formulas, it is more readable to show a business boolean formula than introduce a negated if clause. When dealing with low-level abstractions, it is common to find boolean returns. When you create complex and mature software, you should start to forget about this primitive obsession and care more about real-world rules and identities.

Here's a familiar and not-so-innocent example:

```
function canWeMoveOn() {
  if (work.hasPendingTasks())
    return false;
  else
    return true;
}
```

You can tie your business rule directly to map the real-world business condition as follows:

```
function canWeMoveOn() {
  return !work.hasPendingTasks();
}
```

This return is syntactically identical but more readable. Nevertheless, beware of returning booleans; after the return, you will need an if statement. For this, you can apply Recipe 14.4, "Replacing Switch/Case/Elseif Statements".

In this short example, you can identify if a number is even or odd:

```
boolean isEven(int num) {
    if(num % 2 == 0) {
       return true;
    } else {
       return false;
    }
}
```

By returning the exact implementation your code is cleaner:

```
boolean isEven(int numberToCheck) {
  // You decouple the what (to check for even or odd)
  // with how (the algorithm)
  return (numberToCheck % 2 == 0);
}
```

You can search code libraries for "return true" statements and try to replace them when possible.

Related Recipes

Recipe 6.4, "Removing Double Negatives"

Recipe 14.2, "Renaming Flag Variables for Events"

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 14.12, "Changing Comparison Against Booleans"

Recipe 14.17, “Removing Gratuitous Booleans”

Recipe 24.2, “Dealing with Truthy Values”

14.12 Changing Comparison Against Booleans

Problem

You compare with booleans and perform magic castings, then get unexpected results.

Solution

Don’t compare against a true value. Either you are true or false, or you shouldn’t compare.

Discussion

Comparing against a boolean constant hides magic castings in some languages that deal with truthy and falsy values (see Recipe 24.2, “Dealing with Truthy Values”). They don’t follow the principle of least surprise (see Recipe 5.6, “Freezing Mutable Constants”) nor the fail fast principle (see Chapter 13). You should never mix booleans with boolean castable objects since many languages cast values to boolean-crossing domains.

Here is an example in a bash script:

```
#!/bin/bash

if [ false ]; then
    echo "True"
else
    echo "False"
fi

# this evaluates to true since
# "false" is a non-empty string

if [ false ] = true; then
    echo "True"
else
    echo "False"
fi

# this also evaluates to true
```

To avoid this problem, you can use the explicit `false` constant:

```
#!/bin/bash

if false ; then
    echo "True"
else
    echo "False"
fi

# this evaluates to false
```

It is common practice to use many nonbooleans as booleans, but you should be very strict when using booleans. Many languages have problems with truthy values. Even the scripting languages.

Related Recipes

Recipe 24.2, "Dealing with Truthy Values"

14.13 Extracting from Long Ternaries

Problem

You have very long code in ternary conditions.

Solution

Don't use ternaries for code execution. You should read them as a math formula.

Discussion

Long ternary conditions are difficult to read and break code reuse and testability. You can apply Recipe 10.7, "Extracting a Method to an Object". When a ternary condition is used in code that contains multiple functions, it can be challenging to determine which function is being affected by the condition. This can make it harder to identify and fix defects, as well as to understand how the code works in general.

See the following example:

```
const invoice = isCreditCard ?
  prepareInvoice();
  fillItems();
  validateCreditCard();
  addCreditCardTax();
  fillCustomerDataWithCreditCard();
  createCreditCardInvoice()
:
  prepareInvoice();
```

```
    fillItems();
    addCashDiscount();
    createCashInvoice();

  // The intermediate results are not considered
  // The value of the invoice is the result of
  // the last execution
```

After you apply the extract method recipe, you can read it like a math formula:

```
const invoice = isCreditCard ?
                    createCreditCardInvoice() :
                    createCashInvoice();
```

This is more compact:

```
if (isCreditCard) {
  const invoice = createCreditCardInvoice();
} else {
  const invoice = createCashInvoice();
}
```

Even better with polymorphism:

```
const invoice = paymentMethod.createInvoice();
```

Linters can detect large code blocks. No matter where you have long lines of code, you can always refactor into higher-level functional and shorter methods.

Related Recipes

Recipe 10.7, "Extracting a Method to an Object"

Recipe 11.1, "Breaking Too Long Methods"

14.14 Converting Nonpolymorphic Functions to Polymorphic

Problem

You have methods that perform the same action but are not interchangeable.

Solution

Force polymorphism for extensibility.

Discussion

Polymorphism

Two objects are *polymorphic* regarding a set of methods if they have the same signature and perform the same action (maybe with a different implementation).

You can achieve partial polymorphism if two objects answer to the same method invocation in the same semantic way with possibly different implementations. With polymorphism, you favor extensibility, reduce coupling, and avoid lots of ifs.

In this example the names are not polymorphic even though they achieve the same behavior:

```
class Array {
    public function arraySort() {
    }
}

class List {
    public function listSort() {
    }
}

class Stack {
    public function stackSort() {
    }
}
```

After renaming the functions, you can achieve polymorphism:

```
interface Sortable {
   public function sort();
}

class Array implements Sortable {
   public function sort() {
      // Implementation of the sort() method for Array
   }
}

class List implements Sortable {
   public function sort() {
      // Implementation of the sort() method for List
   }
}

class Stack implements Sortable {
   public function sort() {
      // Implementation of the sort() method for Stack
```

```
    }
}
```

This is a semantic mistake. You could add a warning for similar method names on polymorphic classes. Naming is very important (see Chapter 7, "Naming"), and you need to name based on concepts and not based on accidental types.

Related Recipes

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

14.15 Changing Equal Comparison

Problem

You have code comparing attributes' equality.

Solution

Don't export and compare; hide the comparison in a single method.

Discussion

Attribute comparison is heavily used in code. You need to focus on behavior and responsibilities. It is an object's responsibility to compare with other objects. Premature optimizers (see Chapter 16) will tell you this is less performant, but you should ask them for real evidence and contrast the more maintainable solution without breaking encapsulation and creating duplicated code.

Here you can see an example of a large codebase wherein a business rule (compare case sensitive or insensitive) is duplicated in many places:

```
if (address.street == 'Broad Street') { }

if (location.street == 'Bourbon St') { }

// 24601 usages in a big system
// Comparisons are case sensitive
```

By delegating the comparison responsibility to a single place, you avoid all code duplication and allow the rule change at a single point:

```
if (address.isAtStreet('Broad Street') { }

if (location.isAtStreet('Bourbon St') { }
// 24601 usages in a big system

function isAtStreet(street) {
  // You can change comparisons to
```

```
    // case sensitive in just one place.
  }
```

You can detect attribute comparison using syntax trees. There can be good uses for primitive types, as with many other recipes. You need to put responsibilities in a single place, including comparison. If some of your business rules change, you need to change a *single point*.

Related Recipes

Recipe 4.2, "Reifying Primitive Data"

Recipe 13.6, "Redefining Hash and Equality"

Recipe 14.12, "Changing Comparison Against Booleans"

Recipe 17.8, "Preventing Feature Envy"

14.16 Reifying Hardcoded Business Conditions

Problem

Your code has hardcoded conditions.

Solution

Move hard business rules to configuration.

Discussion

When you learned to program, you likely concluded that hardcoding is always a bad idea. If you decide to use test-driven development (TDD) (see Recipe 4.8, "Removing Unnecessary Properties"), this is always the case because the methodology favors quick and simple solutions before speculative ones, but TDD also advises you to quickly remove these refactorings with the third optional step to generalize after obtaining strong evidence. If your code has unexplained hardcoded conditions, you need to research the reason and if it is a valid business rule, extract it with an intention-revealing function. If the condition is expressed using a global setting you can also remove it using the code in Recipe 10.2, "Removing Settings/Configs and Feature Toggles".

See the following real-world example favoring a customer with special rules:

```
if (currentExposure > 0.15 && customer != "Very Special Customer") {
  // Be extra careful not to liquidate
  liquidatePosition();
}
```

Here's a more declarative solution, forcing you to specify these customers:

```
customer.liquidatePositionIfNecessary(0.15);

  // This follows the "Tell, don't ask" principle
```

You can search for primary hardcoded conditions (related to primitive types), but you might have more false positives than actual problems. If you conduct code reviews, pay special attention to this kind of hardcoding.

Related Recipes

Recipe 10.2, "Removing Settings/Configs and Feature Toggles"

Recipe 14.5, "Replacing Hardcoded If Conditions with Collections"

14.17 Removing Gratuitous Booleans

Problem

You have a function containing several return statements and all return the same value.

Solution

Carefully check your boolean expressions and refactor them.

Discussion

Designing a function to always return a fixed value can lead to poor code readability and may hide potential defects. However, if this design choice is made, it should not have a negative impact on the functionality of the program. If this occurs consistently throughout the function's logic, it is likely to be an error.

Here is a simple example:

```
if a > 0 and True:
// This code was left on debugging and passed
// by mistake during code reviews
print("a is positive")
else:
print("a is not positive")
```

Here's what it looks like after you simplify it:

```
if a > 0:
    print("a is positive")
else:
    print("a is not positive")
```

Boolean expressions should be straightforward to read and understand.

Related Recipes

Recipe 14.11, "Preventing Return Boolean Values for Condition Checks"

Recipe 14.12, "Changing Comparison Against Booleans"

14.18 Rewriting Nested Ternaries

Problem

You have many nested ternary conditions.

Solution

Change nested ternaries to ifs with early returns.

Discussion

Nesting is always a problem because of the added complexity, and you can fix it with polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic") or early returns. Here is an example with five nested ternaries and a default:

```
const getUnits = secs => (
 secs <= 60       ? 'seconds' :
 secs <= 3600     ? 'minutes' :
 secs <= 86400    ? 'hours'   :
 secs <= 2592000  ? 'days'    :
 secs <= 31536000 ? 'months'  :
                      'years'
)
```

Using ifs makes the code more declarative:

```
const getUnits = secs => {
 if (secs <= 60) return 'seconds';
 if (secs <= 3_600) return 'minutes';
 if (secs <= 86_400) return 'hours';
 if (secs <= 2_592_000) return 'days';
 if (secs <= 31_536_000) return 'months';
 return 'years'
}

// This is using 'Numeric Separators' notation from JavaScript
// to favor readability.
// The underscores are ignored by the JavaScript engine
// and do not affect the value of the number.
```

This is even more human friendly and declarative:

```
const getUnits = secs => {
 if (secs <= 60) return 'seconds';
 if (secs <= 60 * 60) return 'minutes';
 if (secs <= 24 * 60 * 60) return 'hours';
 if (secs <= 30 * 24 * 60 * 60) return 'days';
 if (secs <= 12 * 30 * 24 * 60 * 60) return 'months';
 return 'years'
}

// You can read the premature optimization chapter to find out
// if this brings a considerable performance penalty
```

You can also use a map or polymorphic small objects (see Recipe 4.1, "Creating Small Objects"):

```
const timeUnits = {
  60: 'seconds',
  3_600: 'minutes',
  86_400: 'hours',
  2_592_000: 'days',
  31_536_000: 'months',
};

const getUnits = secs => {
  const unit = Object.entries(timeUnits)
    .find(([limit]) => secs <= limit)?.[1] || 'years';
  return unit;
}
```

Linters can detect this complexity using parsing trees since you need to deal with accidental complexity to improve code readability.

Related Recipes

Recipe 6.13, "Avoiding Callback Hell"

Recipe 14.5, "Replacing Hardcoded If Conditions with Collections"

CHAPTER 15

Null

I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

—Tony Hoare

15.0 Introduction

Most programmers use *null* heavily. It is comfortable, efficient, and fast. And yet software developers have suffered a bazillion problems related to its use. This chapter highlights the problems with null usage and how to solve them.

Null is a flag. It represents different situations depending on the context in which it is used and invoked. This yields the most serious error in software development: coupling a hidden decision in the contract between an object and who uses it. It also breaks the bijection principle by representing multiple elements of the domain with the same entity and forcing you to make contextual interpretations. All objects should be as specific as possible and have a single responsibility (see Recipe 4.7, "Reifying String Validations"), but the least cohesive object of any system is the wildcard: null, which is mapped to several different concepts in the real world.

15.1 Creating Null Objects

Problem

You use null.

Solution

Null is schizophrenic and doesn't exist in the real world. Null's creator and Turing Award–winner Tony Hoare regretted it, and programmers around the world suffered for it. You can replace null using null objects.

Discussion

Programmers use null as different flags. This can indicate different conditions such as the absence of a value, an undefined value, an error, and so on. Multiple semantics lead to coupling and errors. Null usage has a lot of problems, including coupling between the callers and the senders, mismatch between the callers and the senders, and if/switch/case pollution. Also, null is not polymorphic with real objects, which is how you get the null pointer exception error. Lastly, null does not exist in the real world. Thus, it violates the bijection principle.

Null Object Pattern

The *null object pattern* suggests creating a special object called a "null object" that behaves like a regular object but has almost no functionality. Its advantage is that you can safely call methods on the null object without having to check for null references with an if (See Chapter 14, "Ifs").

In this example, there is a coupon or null (no coupon):

```
class CartItem {
    constructor(price) {
        this.price = price;
    }
}

class DiscountCoupon {
    constructor(rate) {
        this.rate = rate;
    }
}

class Cart {
    constructor(selecteditems, discountCoupon) {
        this.items = selecteditems;
        this.discountCoupon = discountCoupon;
    }

    subtotal() {
        return this.items.reduce((previous, current) =>
            previous + current.price, 0);
    }
    total() {
        if (this.discountCoupon == null)
            return this.subtotal();
        else
            return this.subtotal() * (1 - this.discountCoupon.rate);
    }
}

cart = new Cart([
    new CartItem(1),
    new CartItem(2),
    new CartItem(7)
    ], new DiscountCoupon(0.15)]);
// 10 - 1.5 = 8.5

cart = new Cart([
    new CartItem(1),
    new CartItem(2),
    new CartItem(7)
    ], null);
// 10 - null  = 10
```

By introducing the null object pattern, you can avoid if checking (that you might sometimes miss):

```
class CartItem {
    constructor(price) {
        this.price = price;
    }
}

class DiscountCoupon {
    constructor(rate) {
        this.rate = rate;
    }

    discount(subtotal) {
        return subtotal * (1 - this.rate);
    }
}

class NullCoupon {
    discount(subtotal) {
        return subtotal;
    }
}

class Cart {
    constructor(selecteditems, discountCoupon) {
        this.items = selecteditems;
        this.discountCoupon = discountCoupon;
    }
    subtotal() {
        return this.items.reduce(
            (previous, current) => previous + current.price, 0);
    }

    total() {
        return this.discountCoupon.discount(this.subtotal());
    }
}

cart = new Cart([
    new CartItem(1),
    new CartItem(2),
    new CartItem(7)
    ], new DiscountCoupon(0.15));
// 10 - 1.5 = 8.5

cart = new Cart([
    new CartItem(1),
    new CartItem(2),
    new CartItem(7)
    ], new NullCoupon());
// 10 - nullObject = 10
```

Most linters can show null usage and warn you, and there are languages like Typescript that have no null at all. The Rust language has an "Option" type that represents either Some(value) or None. Kotlin has a "nullable type" system that allows developers to specify whether a value can be null or not. Null is also present in relational databases, representing many different things at the same time.

Null Pointer Exception

A *null pointer exception* is a common error that occurs when a program attempts to access or use a null pointer, which is a variable or object reference that points to no memory address or no object instance.

See Also

"Null References: The Billion Dollar Mistake" by Tony Hoare

15.2 Removing Optional Chaining

Problem

You have code sweeping a null under the rug during a function call cascade.

Solution

Avoid nulls and undefined values. If you avoid them, you will never need optionals.

Discussion

Optional chaining, optionals, coalescence, and many other solutions help you deal with sweeping the infamous nulls under the rug. There's no need to use them once your code is mature, robust, and has no more nulls.

Optional Chaining

Optional chaining allows you to access nested properties of an object without having to check for the existence of each property in the chain. Without it if you try to access a property of an object that does not exist, it will throw an error.

Here you can see the optional chaining operator:

```
const user = {
  name: 'Hacker'
};

if (user?.credentials?.notExpired) {
  user.login();
}

user.functionDefinedOrNot?.();

// Seems compact but it is hacky and has lots
// of potential NULLs and Undefined
```

As always you can make this explicit. This code is less compact but more declarative:

```
function login() {}

const user = {
  name: 'Hacker',
  credentials: { expired: false }
};

if (!user.credentials.expired) {
  login();
}

// Also compact
// User is a real user or a polymorphic NullUser
// Credentials are always defined.
// Can be an instance of InvalidCredentials
// Assuming you eliminated nulls from the code
```

```
if (user.functionDefinedOrNot !== undefined) {
    functionDefinedOrNot();
}

// This is also wrong.
// Explicit undefined checks are a similar problem
```

You can achieve a similar behavior using the Elvis operator (`?:`):

```
a ?: b
```

This is shorthand for:

```
if (a != null) a else b
```

For example, you can rewrite:

```
val shipTo = address?: "No address specified"
```

To:

```
val shipTo = if (address != null) address else "No address specified"
```

These are language features; you can detect them and remove them using Recipe 15.1, "Creating Null Objects". Many developers feel safe polluting their code with code dealing with nulls. This is safer than not treating nulls at all, but sooner or later you will miss some check. Nullish, truthy, and falsy values are always problematic and you need to aim higher and make cleaner code (see Recipe 24.2, "Dealing with Truthy Values").

The good: remove all nulls from your code. The bad: use optional chaining. The ugly: not treating nulls at all.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 14.6, "Changing Boolean to Short-Circuit Conditions"

Recipe 14.9, "Avoiding Short-Circuit Hacks"

Recipe 15.1, "Creating Null Objects"

Recipe 24.2, "Dealing with Truthy Values"

See Also

"Optional Chaining" on Mozilla.org

"Nullish Value" on Mozilla.org

15.3 Converting Optional Attributes to a Collection

Problem

You need to model an optional attribute.

Solution

Collections are polymorphic and a fantastic tool to model optionality. Model optional attributes with a collection.

Discussion

If you need to model something that might be missing, some fancy languages will provide optional, nullable, and many other incorrect solutions dealing with null. But notice that empty collections and nonempty collections are polymorphic.

Here is an example of an optional email:

```
class Person {
  constructor(name, email) {
    this.name = name;
    this.email = email;
  }

  email() {
    return this.email;
    // might be null
  }
}

// You cannot safely use person.email()
// You need to check for null explicitly
```

Here's what it looks like when you convert the email to a (possibly empty) collection of emails:

```
class Person {
  constructor(name, emails) {
    this.name = name;
    this.emails = emails;
    // emails should always be a collection.
    // even an empty one
    // You can check it here
    if (emails.length > 1) {
        throw new Error("Emails collection can have at most one element.");
   }
  }

  emails() {
    return this.emails;
  }
  // You can mutate the emails since they are not essential

  addEmail(email) {
    this.emails.push(email);
  }

  removeEmail(email) {
    const index = this.emails.indexOf(email);
    if (index !== -1) {
      this.emails.splice(index, 1);
    }
  }
}

// You can iterate the person.emails()
// in a loop without checking for null
```

You can detect nullable attributes and change them when necessary since this is a generalization of the null object pattern (see Recipe 15.1, "Creating Null Objects").

> You can check the new cardinality of the collection to ensure it meets the requirements (previously it was 0 or 1).

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 15.2, "Removing Optional Chaining"

Recipe 17.7, "Removing Optional Arguments"

15.4 Using Real Objects for Null

Problem

You need to create null objects that are real.

Solution

Don't abuse design patterns, including the null object pattern. Find real null objects on the bijection and create these objects instead.

Discussion

Abusing the null object pattern (see Recipe 15.1, "Creating Null Objects") produces empty classes, pollutes namespaces (see Recipe 18.4, "Removing Global Classes"), and creates duplicate behavior. You can create null objects instantiating real object classes. The null object pattern is a great alternative to nulls and ifs, and the structure of the pattern tells you to create a hierarchy. However, this is not necessary; instead, you need real objects to be polymorphic to null objects. Inheritance is not the only way to achieve polymorphism (see Recipe 14.4, "Replacing Switch/Case/Elseif Statements"). A simple solution is to create a real object that behaves like a null one. Table 15-1 lists some familiar null objects.

Table 15-1. Familiar null objects

Class	Null object
Number	0
String	""
Array	[]

This example creates a special `NullAddress`:

```
abstract class Address {
    public abstract String city();
    public abstract String state();
    public abstract String zipCode();
}

// Using inheritance for null objects is a mistake
// You should use interfaces (when available)
public class NullAddress extends Address {

    public NullAddress() { }

    public String city() {
        return Constants.EMPTY_STRING;
    }

    public String state() {
        return Constants.EMPTY_STRING;
    }

    public String zipCode() {
        return Constants.EMPTY_STRING;
    }

}

public class RealAddress extends Address {
    private String zipCode;
    private String city;
    private String state;

    public RealAddress(String city, String state, String zipCode) {
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
    }

    public String zipCode() {
        return zipCode;
    }

    public String city() {
        return city;
    }

    public String state() {
        return state;
    }

}
```

Here's what it looks like when you use a real instance of the `Address` object:

```
    // There are just "addresses"
    public class Address {

        private String zipCode;
        private String city;
        private String state;

        public Address(String city, String state, String zipCode) {
            // Looks anemic :(
            this.city = city;
            this.state = state;
            this.zipCode = zipCode;
        }

        public String zipCode() {
            return zipCode;
        }

        public String city() {
            return city;
        }

        public String state() {
            return state;
        }

    }
```

```
Address nullAddress = new Address(
    Constants.EMPTY_STRING,
    Constants.EMPTY_STRING,
    Constants.EMPTY_STRING);

// or

Address nullAddress = new Address("", "", "");

// This is the null object
// You should NOT assign it to a singleton, static, or global
// It behaves like a null object. That's enough
// No premature optimizations
```

Creating null object classes is sometimes an overdesign symptom. This is often the case when you can create and use a real object. This real object should never be `global`, `singleton` (see Recipe 17.2, "Replacing Singletons"), or `static`.

Related Recipes

Recipe 14.4, "Replacing Switch/Case/Elseif Statements"

Recipe 15.1, "Creating Null Objects"

Recipe 17.2, "Replacing Singletons"

Recipe 18.1, "Reifying Global Functions"

Recipe 18.2, "Reifying Static Functions"

Recipe 19.9, "Migrating Empty Classes"

15.5 Representing Unknown Locations Without Using Null

Problem

You use special values to indicate missing data. But you can make mistakes.

Solution

Don't use null values for real places.

Discussion

If you flag some value to indicate you are missing data, you will violate the fail fast principle and get unexpected results. You need to model the unknown location with polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic"). Null Island is a fictional place, located at the coordinates 0°N 0°E, at the intersection of the Prime Meridian and the equator in the Atlantic Ocean. The name “Null Island” comes from the fact that this location represents the point where a lot of GPS systems place any data that has missing or invalid location coordinates. In reality, there is no landmass at this location, and it is actually in the middle of the ocean. This point has become a popular reference for geographic information systems (GIS) and mapping software, as it serves as a way to filter out errors in location data.

See the following example using the zero values as a special case:

```kotlin
class Person(val name: String, val latitude: Double, val longitude: Double)
fun main() {
    val people = listOf(
        Person("Alice", 40.7128, -74.0060), // New York City
        Person("Bob", 51.5074, -0.1278), // London
        Person("Charlie", 48.8566, 2.3522), // Paris
        Person("Tony Hoare", 0.0, 0.0) // Null Island
    )

    for (person in people) {
        if (person.latitude == 0.0 && person.longitude == 0.0) {
            println("${person.name} lives on Null Island!")
        } else {
            println("${person.name} lives at " +
                    "(${person.latitude}, ${person.longitude}).")
        }
    }
}
```

When you explicitly model the unavailability of the data, you don’t know where Tony lives:

```kotlin
abstract class Location {
    abstract fun calculateDistance(other: Location): Double
    abstract fun ifKnownOrElse(knownAction: (Location) -> Unit,
        unknownAction: () -> Unit)
}

class EarthLocation(val latitude: Double, val longitude: Double) : Location() {
    override fun calculateDistance(other: Location): Double {
        val earthRadius = 6371.0
        val latDistance = Math.toRadians(
            latitude - (other as EarthLocation).latitude)
        val lngDistance = Math.toRadians(
            longitude - other.longitude)
        val a = sin(latDistance / 2) * sin(latDistance / 2) +
          cos(Math.toRadians(latitude)) *
          cos(Math.toRadians(other.latitude)) *
          sin(lngDistance / 2) * sin(lngDistance / 2)
        val c = 2 * atan2(sqrt(a), sqrt(1 - a))
        return earthRadius * c
}
```

```
    override fun ifKnownOrElse(knownAction:
      (Location) -> Unit, unknownAction: () -> Unit) {
        knownAction(this)
    }
}

class UnknownLocation : Location() {
    override fun calculateDistance(other: Location): Double {
        throw IllegalArgumentException(
            "Cannot calculate distance from an unknown location.")
    }

    override fun ifKnownOrElse(knownAction:
        (Location) -> Unit, unknownAction: () -> Unit) {
            unknownAction()
    }
}

class Person(val name: String, val location: Location)

fun main() {
    val people = listOf(
        Person("Alice", EarthLocation(40.7128, -74.0060)), // New York City
        Person("Bob", EarthLocation(51.5074, -0.1278)), // London
        Person("Charlie", EarthLocation(48.8566, 2.3522)), // Paris
        Person("Tony", UnknownLocation()) // Unknown location
    )
    val rio = EarthLocation(-22.9068, -43.1729) // Rio de Janeiro coordinates

    for (person in people) {
        person.location.ifKnownOrElse(
            { location -> println(person.name" is " +
                person.location.calculateDistance(rio) +
                    " kilometers { println("${person.name} "
                        + "is at an unknown location.") }
       )
    }
}
```

The function `ifKnownOrElse` is a monad solving the problem using polymorphism with a null object. Don't use null to represent real objects (see Recipe 15.4, "Using Real Objects for Null").

Monads

A *monad* provides a structured way to encapsulate and manipulate functions. It allows you to chain operations together, handling functions and their side effects in a consistent and predictable manner, for example when working with optional values.

Related Recipes

Recipe 15.1, “Creating Null Objects”

Recipe 15.4, “Using Real Objects for Null”

Recipe 17.5, “Converting 9999 Special Flag Values to Normal”

See Also

“Null Island” on Wikipedia

“Null Island” on Google Maps

CHAPTER 16
Premature Optimization

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

—Donald Knuth, "Structured Programming with go to Statements"

16.0 Introduction

Premature optimization is a big problem in the industry. Every developer is fascinated by computational complexity and the beauty of the fastest algorithms. Deciding where and when to make an optimization is one of the differences between a junior developer and a senior one. Optimizations are not free since they add a lot of accidental complexity and you must apply them very carefully. Complex solutions carry your model far away from the real world because they add many obscure layers between the model and the business objects, violating the bijection principle. They also damage readability. You should apply them only under strong factual evidence.

Computational Complexity

Computational complexity studies the resources required to solve computational problems. The most important are time and memory. It measures and compares the efficiency of algorithms and computational systems regarding these resources.

Modern artificial intelligence assistants help you optimize your code. You can become a technological centaur, leave the accidental optimization tools to artificial assistants, and focus on the essential domain problems.

16.1 Avoiding IDs on Objects

Problem

You have IDs, primary keys, and references that don't exist in the real world.

Solution

Remove the IDs and link directly to your objects.

Discussion

IDs are accidental and don't exist in the real world unless you need IDs to export your objects and create a global reference. Therefore, you don't need to link objects internally using IDs. They are not valid attributes for any object since references are always outside the object (no object should know their identification). Following the bijection principle you should avoid them and only use identifiers if you need to provide an external (accidental) reference. Common occurrences of external identifiers are databases, APIs, and serializations.

Primary Keys

In the context of databases, a *primary key* is a unique identifier for a specific record or row in a table. It serves as a way to uniquely identify each record in a table and allows for efficient searching and sorting of data. A primary key can be a single column or a combination of columns that, when combined, form a unique value for each record in the table. Typically, a primary key is created with a table and used as a reference by other tables in a database that have relations with that table.

If you need to declare a key, always use dark keys (something not related to consecutive small integers) like GUIDs, and if you are afraid of creating a big relation graph use proxies or lazy loading (searching for the related full object only when you need it). You need strong evidence of a real performance penalty to add this accidental complexity to your code.

Globally Unique Identifier

A GUID (*globally unique identifier*) is a unique identifier used in computer systems to map resources such as files, objects, or entities in a network. GUIDs are generated using algorithms that guarantee their uniqueness.

Here’s a school domain with `Teacher`, `School`, and `Student` entities:

```
class Teacher {
    static getByID(id) {
        // This is coupled to the database
        // Thus violating separation of concerns
    }

    constructor(id, fullName) {
        this.id = id;
        this.fullName = fullName;
    }
}

class School {
    static getByID(id) {
        // go to the coupled database
    }

    constructor(id, address) {
        this.id = id;
        this.address = address;
    }
}

class Student {
    constructor(id, firstName, lastName, teacherId, schoolId) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.teacherId = teacherId;
        this.schoolId = schoolId;
    }

    school() {
        return School.getById(this.schoolId);
    }

    teacher() {
        return Teacher.getById(this.teacherId);
    }
}
```

Here’s what it looks like when they’re referenced in real world:

```
class Teacher {
    constructor(fullName) {
        this.fullName = fullName;
    }
}

class School {
    constructor(address) {
        this.address = address;
    }
}

class Student {
    constructor(firstName, lastName, teacher, school) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.teacher = teacher;
        this.school = school;
    }
}
// The ids are no longer needed since they don't exist in the real world.
// If you need to expose a School to an external API or a database,
// another object (not school)
// will keep the mapping externalId<->school and so on
```

This is a design policy. You can enforce linters and business objects to warn you if you define an attribute or function including the sequence ID. IDs are not necessary for object-oriented software. You reference objects (essential) and never IDs (accidental). If you need to provide a reference out of your system's scope (APIs, interfaces, serializations) use dark and meaningless IDs like GUIDs. You can use the repository design pattern or something similar.

Repository Design Pattern

The *repository design pattern* provides an abstraction layer between the application's business logic and the data storage layer, allowing for a more flexible and maintainable architecture.

Related Recipes

Recipe 3.6, "Removing DTOs"

Recipe 16.2, "Removing Premature Optimization"

See Also

"Universally Unique Identifier" on Wikipedia

16.2 Removing Premature Optimization

Problem

You have speculative code optimized without empirical evidence.

Solution

Don't speculate about things that might not happen. Make optimizations using evidence from real-world scenarios.

Discussion

Premature optimization is a very ill-advised and common practice that makes the code more complex and difficult to maintain. It damages readability and testability and brings accidental coupling. You must optimize your code only after it has good coverage and a conclusive real-world scenario benchmark. If you need to decide between two implementations, you should choose the more readable one, even if the benchmark says it performs badly in a cycle with a thousand executions. Most likely, you will not call the function a thousand times. You can use the test-driven development technique (see Recipe 4.8, "Removing Unnecessary Properties") instead since it always favors the simplest solution.

Here is an example where caching is used for a situation where database performance is acceptable:

```
class Person {
    ancestors() {
        cachedResults =
            GlobalPeopleSingletonCache.getInstance().relativesCache(this.id);
        if (cachedResults != null) {
            return (cachedResults.hashFor(this.id)).getAllParents();
        }
        return database().getAllParents(this.id);
    }
}
```

Since you will not run this code a thousand times you can simplify it:

```
class Person {
  ancestors() {
     return this.mother.meAndAncestors().concat(this.father.meAndAncestors());
  }
  meAndAncestors() {
     return this.ancestors().push(this);
  }
}
```

This is an antipattern (see Recipe 5.5, "Removing Lazy Initialization"). It cannot be detected by mechanical tools (yet). You should defer performance decisions until functional models are mature enough. Donald Knuth created/compiled the best performing algorithms and data structures in his book *The Art of Computer Programming*. And showing great wisdom, he warned you not to abuse them.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

See Also

"Structured Programming with go to Statements" by Donald Knuth on ACM

"Premature Optimization" on C2 Wiki

16.3 Removing Bitwise Premature Optimizations

Problem

You have code micro-optimized with bitwise operators.

Solution

Don't use bitwise operators unless your business model is bitwise logic.

Bitwise Operators

Bitwise operators manipulate the individual bits of numbers. Your computer uses them to perform low-level logical operations between bits like AND, OR, and XOR. They work in the integer domain, which is different from the boolean domain.

Discussion

You don't have to confuse integers with booleans. They are completely different in the bijection. Sadly, many programming languages mix them for cleverness and premature optimization, breaking the natural barrier between accidental implementation and essential domain problems. This situation brings a lot of unforeseen problems with truthy and falsy values (see Recipe 24.2, "Dealing with Truthy Values") and also breaks maintainability. You should only optimize code based on evidence and always use the scientific method, benchmarking, and improving code only if it is really necessary. Bear the cost of changeability and maintainability when using bitwise operators. The following code works in many languages, but it is a hack:

```
const nowInSeconds = ~~(Date.now() / 1000)

// The double bitwise NOT operator ~~
// is a bitwise operation that performs a bitwise
// negation followed by a bitwise negation again.
// This operation effectively truncates any decimal places
// converting the result to an integer.
```

This is clearer:

```
const nowInSeconds = Math.floor(Date.now() / 1000)
```

As always, if your domain is real-time or mission-critical software, you can sacrifice readability for the trade-off, but if you find this code in a pull request or code review, you need to understand the reasoning. If it is not justified, you should do a rollback and change it to normal logic.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 16.2, "Removing Premature Optimization"

Recipe 16.5, "Changing Structural Optimizations"

Recipe 22.1, "Removing Empty Exception Blocks"

Recipe 24.2, "Dealing with Truthy Values"

16.4 Reducing Overgeneralization

Problem

You have code with premature overgeneralization.

Solution

Don't make generalizations beyond actual knowledge. You need to be an expert at learning and not at guessing the future.

Discussion

Overgeneralization is always a special case of bijection violation (as defined in Chapter 2) since you are modeling entities you haven't seen in the real world yet. Refactoring is not just looking at structural code; you need to refactor behavior and check if it really needs an abstraction.

This example makes an inference on something before having enough proof:

```
fn validate_size(value: i32) {
    validate_integer(value);
}

fn validate_years(value: i32) {
    validate_integer(value);
}

fn validate_integer(value: i32) {
    validate_type(value, :integer);
    validate_min_integer(value, 0);
}
```

This is a more compact solution:

```
fn validate_size(value: i32) {
    validate_type(value, Type::Integer);
    validate_min_integer(value, 0);
}

fn validate_years(value: i32) {
    validate_type(value, Type::Integer);
    validate_min_integer(value, 0);
}

// Duplication is accidental, therefore you should not abstract it
```

Software development is a thinking activity, and you have automated tools to help and assist you.

Related Recipes

Recipe 10.1, "Removing Repeated Code"

16.5 Changing Structural Optimizations

Problem

You have structural optimizations on time and space complexity based on nonrealistic scenarios.

Solution

Don't optimize data structures until you have a real-use scenario benchmark.

Discussion

Structural optimizations damage readability because they take you away from the bijection. If you need to make structural optimizations because you have strong evidence, you should still cover your scenarios with tests. Write readable (and possibly nonperformant) code. Do a real benchmark with real user data. (No, iterating your code 100,000 times might not be a real use case.) If you have conclusive data, you need to improve the benchmark's bottlenecks using the Pareto principle (see "Chapter 1" in Chapter 1). Attack the worst 20% of problems causing 80% of the bad performance.

In university and online courses, people often learn algorithms, data structures, and computational complexity before good design rules. You tend to overestimate the (possible) performance problems and underestimate code readability and software lifetime. Premature optimization often comes with no evidence that you are solving real problems, and you need to surgically improve your code when the facts tell you that you have a real issue.

See this problem in action with a real-world example:

```
for (k = 0; k < 3 * 3; ++k) {
     const i = Math.floor(k / 3);
     const j = k % 3;
     console.log(i + ' ' +  j);
}

// This cryptic piece of code iterates a
// two-dimensional array
// You don't have proof this will be useful
// in real contexts
```

Here's how to completely rewrite it:

```
for (outerIterator = 0; outerIterator< 3; outerIterator++) {
  for (innerIterator = 0; innerIterator< 3; innerIterator++) {
   console.log(outerIterator + ' ' +  innerIterator);
  }
 }

// This is a readable double for-loop
// 3 is a small number
// No performance issues (for now)
// You will wait for real evidence
```

If you are writing business code and not low-level code, you need to stop optimizing for machines and start optimizing for human readers and code maintainers. Also, avoid programming languages designed for premature optimization (Go, Rust, C++) and favor robust, higher level, and clean code choices.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 16.2, "Removing Premature Optimization"

16.6 Removing Anchor Boats

Problem

You have code just in case you need it later.

Solution

Don't leave code for future use.

Discussion

Anchor boats bring accidental complexity and coupling and are also an example of dead code. You need to remove dead code and leave only covered and actually tested code. Here's an anchor boat example:

```
final class DatabaseQueryOptimizer {

  public function selectWithCriteria($tableName, $criteria) {
    // Make some optimizations manipulating criteria
  }

  private function sqlParserOptimization(SQLSentence $sqlSentence)
    : SQLSentence {
    // Parse the SQL converting it to a string
    // and then working with their nodes as strings and lots of regex
    // This was a very costly operation overcoming real SQL benefits.
    // But since you made too much work we decide to keep the code.
  }
}
```

Here's what it looks like when you remove it:

```
final class DatabaseQueryOptimizer {

  public function selectWithCriteria($tableName, $criteria) {
    // Make some optimizations manipulating criteria
  }
}
```

Using some mutation testing (see Recipe 5.1, "Changing var to const") variants you can remove the dead code and see if some tests fail. You need to have good coverage to rely on this solution. Dead code is always a problem, and you can use modern development techniques like TDD (see Recipe 4.8, "Removing Unnecessary Properties") to ensure all code is alive.

Related Recipes

Recipe 10.9, "Removing Poltergeist Objects"

Recipe 12.1, “Removing Dead Code”

16.7 Extracting Caches from Domain Objects

Problem

Caches are fancy because they apparently solve performance problems magically, but they have a hidden cost.

Solution

Remove the cache until you have concrete evidence and are ready to pay the costs of using it.

Discussion

Cache

A *cache* temporarily stores frequently accessed objects for faster access. You can use it to improve the performance of software applications by reducing the number of accesses to expensive resources. By caching data in memory, software can avoid the overhead of accessing slower storage devices and instead retrieve objects directly from the cache.

Caches create a lot of coupling since they don’t exist in the real world and make changes more difficult. They break determinism and damage testability and maintainability. When you program a solution with a cache, your code becomes erratic unless you encapsulate the solution very carefully. Cache invalidation is a very difficult problem and most of the time underestimated by cache developers. If you have a conclusive benchmark and are willing to pay for some coupling, you can put an object in the middle. Be careful and add unit tests for all your invalidation scenarios. Experience shows you will face them in an incremental way. Look for a real-world cache metaphor in the MAPPER (as defined in Chapter 2); if you find it you can model it.

Here is an intrusive cache embedded in the Book object:

```
final class Book {

    private $cachedBooks;

    public function getBooksFromDatabaseByTitle(string $title) {
        if (!isset($this->cachedBooks[$title])) {
            $this->cachedBooks[$title] =
                $this->doGetBooksFromDatabaseByTitle($title);
        }
        return $this->cachedBooks[$title];
    }

    private function doGetBooksFromDatabaseByTitle(string $title) {
        return globalDatabase()->selectFrom('Books', 'WHERE TITLE = ' . $title);
    }
}
```

You can put the cache outside the Book domain object:

```
final class Book {
    // Just Book related Stuff
}

interface BookRetriever {
    public function bookByTitle(string $title);
}

final class DatabaseLibrarian implements BookRetriever {
    public function bookByTitle(string $title) {
        // Go to the database (hopefully not a global)
    }
}

final class HotSpotLibrarian implements BookRetriever {
    // You always look for real-life metaphors
    private $inbox;
    private $realRetriever;

    public function bookByTitle(string $title) {
        if ($this->inbox->includesTitle($title)) {
            // You are lucky. Someone has just returned the book copy.
            return $this->inbox->retrieveAndRemove($title);
        } else {
            return $this->realRetriever->bookByTitle($title);
        }
    }
}
```

This is a design smell and you can enforce it by a policy. Caches should be functional and intelligent; in this way, you can manage invalidation. General-purpose caches are suitable only for low-level objects like operating systems, files, and streams; you shouldn't cache domain objects with them.

Related Recipes

Recipe 13.6, "Redefining Hash and Equality"

Recipe 16.2, "Removing Premature Optimization"

16.8 Removing Callback Events Based on Implementation

Problem

You have coupled code between the trigger and the action in a callback.

Solution

Name your functions according to what happened at the event.

Discussion

Callbacks follow the observer design pattern, the intention of which is to decouple an event from the action. If you make this link directly, your code is less maintainable and more related to implementation. As a rule, you should name the events after "what happened," not "what you should do."

See the event in this example:

```
const Item = ({name, handlePageChange)} =>
  <li onClick={handlePageChange}>
    {name}
  </li>

// handlePageChange is coupled with what you decide to do
// instead of what really happened
//
// You cannot reuse this kind of callback
```

This is more concise and less coupled:

```
const Item = ({name, onItemSelected)} =>
  <li onClick={onItemSelected}>
    {name}
  </li>

// onItemSelected will be called just when an item was selected.
// Parent can decide what to do (or do nothing)
// You defer the decision
```

You can detect this problem and apply the recipe during peer code reviews. Names are very important. You should delay defining names coupled with implementation until the very last moment.

Observer Design Pattern

The *observer design pattern* defines a one-to-many dependency between objects; for example, when one object changes its state, all its dependent objects are notified and updated automatically, without a direct reference. You subscribe to published events and the modified object emits an alert without knowing who is subscribed.

Related Recipes

Recipe 17.13, "Removing Business Code from the User Interface"

16.9 Removing Queries from Constructors

Problem

You have methods that access a database in a constructor.

Solution

Constructors should construct (and probably initialize) objects. Decouple your persistence mechanism from your domain objects.

Discussion

Side effects are always a bad practice. Also, you should avoid coupling the database with your business objects since persistence is accidental and not present on the MAPPER. You need to decouple essential business logic from accidental persistence. In persistence classes, run queries in functions other than constructors/destructors. When dealing with legacy code, you will often find that the database is not correctly separated from business objects. Constructors should never have side effects because according to the single responsibility principle (see Recipe 4.7, "Reifying String Validations"), they should only build *valid* objects.

Here there is a `Person` constructor explicitly calling the database:

```
public class Person {
  int childrenCount;

  public Person(int id) {
    connection = new DatabaseConnection();
    childrenCount = connection.sqlCall(
        "SELECT COUNT(CHILDREN) FROM PERSON WHERE ID = " . id);
  }
}
```

Here's what it looks like after you decouple it:

```
public class Person {
  int childrenCount;

  public Person(int id, int childrenCount) {
    this.childrenCount = childrenCount;
    // You can assign the number in the constructor
    // Accidental database is decoupled
    // You can test the object
  }
}
```

Separation of concerns is key, and coupling is your main enemy when designing robust software (see Recipe 8.3, "Removing Logical Comments").

Related Recipes

Recipe 16.10, "Removing Code from Destructors"

16.10 Removing Code from Destructors

Problem

You have code that deallocates things in your destructors.

Solution

Don't use destructors. And don't write functional code there.

Discussion

Using code destructors creates coupling problems and yields unexpected results and, most likely, memory leaks. You should avoid them following the rule of zero and let the garbage collector work for you. A class destructor is a special method called when an object is destroyed or goes out of scope. In the past, virtual machines had no garbage collectors, and destructors were responsible for cleaning up any resources that the object had acquired during its lifetime, such as closing open files or releasing memory allocated on the heap. Nowadays, object destruction and resource release are automatic in most modern programming languages.

Rule of Zero

The *rule of zero* suggests you avoid writing code for things that the programming language or existing libraries can do on their own. If there is a behavior that can be implemented without writing any code, then you should rely on the existing code.

Here is an explicit destructor deallocating a file resource:

```
class File {
public:
    File(const std::string& filename) {
        file_ = fopen(filename.c_str(), "r");
    }
    ~File() {
        if (file_) {
            fclose(file_);
        }
    }

private:
    FILE* file_;
};
```

You can add a warning on the destructor about whether a `File` is still open:

```
class File {
public:
    File() : file_(nullptr) {}

    bool Open(const std::string& filename) {
        if (file_) {
            fclose(file_);
        }
        file_ = fopen(filename.c_str(), "r");
        return (file_ != nullptr);
    }

    bool IsOpen() const {
        return (file_ != nullptr);
    }

    void Close() {
        if (file_) {
            fclose(file_);
            file_ = nullptr;
        }
    }
    ~File() {
        // Instead of closing the file you throw an exception
        // if it is open (which is an invalid scenario)
        if (file_) {
            throw std::logic_error(
                "File is still open after reaching its destructor");
        }
    }

private:
    FILE* file_;
};
```

In many languages, there are specific interfaces to convey the intent of "closability," and that should be utilized if you really want to release resources, for example, `Closable` in Java or the `Disposable` interface in C#.

Linters can warn you when you write code in destructors. As notable exceptions, in very critical low-level code you cannot afford a garbage collector due to its small performance penalty and resource consumption. Another exception for garbage collectors is real-time systems because most automatic collectors execute at a random time and produce arbitrary delays on real-time behavior. In other cases, writing code in destructors is a symptom of premature optimization. You need to understand the lifecycle of your objects and manage the events accurately.

Garbage Collector

A *garbage collector* is used by programming languages to automatically manage memory allocation and deallocation. It works by identifying and then removing objects from memory that are no longer in use by the program, freeing up used memory.

Related Recipes

Recipe 16.9, "Removing Queries from Constructors"

CHAPTER 17

Coupling

Two parts of a software system are coupled if a change in one might cause a change in the other.

—Neal Ford et al., *Software Architecture: The Hard Parts* (O'Reilly 2021)

17.0 Introduction

Coupling is the degree of interdependence between your objects. Higher coupling means that changes in one object can have a significant impact on others, whereas low coupling means that objects are relatively independent and changes in one of them have little impact on others. High coupling can make it difficult to make changes to software without unintended consequences. Most of the work in a large software system lowers accidental coupling. Systems with high coupling are harder to understand and maintain, the interactions between objects are more complex, and changes cause ripple effects throughout the codebase. An entangled system with desirable emergent properties can be fascinating, while a badly coupled one can become a maintenance nightmare.

17.1 Making Hidden Assumptions Explicit

Problem

You have code with hidden assumptions not explicit in your solution and they affect your system's behavior.

Solution

Keep your code explicit.

Discussion

Software is about contracts, and ambiguous contracts are a nightmare. Hidden assumptions are underlying beliefs or expectations not explicitly stated in the code. They are still present and can impact the behavior of the software. Various factors can give rise to assumptions, such as incomplete requirements, incorrect presumptions about the user or environment, limitations of the programming language or tools, and bad accidental decisions.

Here you can see an (incorrect) hidden assumption on the measurement units:

```
tenCentimeters = 10
tenInches = 10

tenCentimeters + tenInches
# 20
# This error is based on the hidden assumption of a unit (any)
# and caused the Mars Climate Orbiter failure
```

If you make it explicit, you can handle an early exception and deal with the conversion:

```
class Unit:
    def __init__(self, name, symbol):
        self.name = name
        self.symbol = symbol

class Measure:
    def __init__(self, scalar, unit):
        self.scalar = scalar
        self.unit = unit

    def __str__(self):
        return f"{self.scalar} {self.unit.symbol}"

centimetersUnit = Unit("centimeters", "cm")
inchesUnit = Unit("inches", "in")

tenCentimeters = Measure(10, centimetersUnit)
tenInches = Measure(10, inchesUnit)

tenCentimeters + tenInches
# error until a conversion factor is introduced,
# in this case, the conversion is constant
# inches = centimeters / 2.54
```

Hidden assumptions can be difficult to identify and can lead to defects, security vulnerabilities, and usability issues. To mitigate these risks, you should be aware of assumptions and biases. Engage with users to understand their needs and expectations, and thoroughly test your software in various scenarios to uncover hidden assumptions and edge cases.

Related Recipes

Recipe 6.8, "Replacing Magic Numbers with Constants"

See Also

See the Mars Climate Orbiter disaster discussion in "The One and Only Software Design Principle" section of Chapter 2.

17.2 Replacing Singletons

Problem

You have singletons on your code.

Solution

Singletons can create a lot of problems, and most of the developer community considers singletons to be antipatterns. You can replace them with contextual unique objects.

Discussion

Singletons are a clear example of global coupling and premature optimization (see Chapter 16, "Premature Optimization"). They make testability more difficult, introduce tight coupling between classes, and have problems in multithreading environments. In the past, many developers used singletons as a global well-known point of access for database access, configuration, environment setting, and logging.

Here you can see a typical singleton definition where `God` is unique according to certain religions:

```
class God {
    private static $instance = null;

    private function __construct() {
    }

    public static function getInstance() {
        if (null === self::$instance) {
            self::$instance = new self();
        }

        return self::$instance;
    }
}
```

Instead, you can use a contextual object:

```
interface Religion {
    // Define common behavior for religions
}

final class God {
    // Different religions have different beliefs
}

final class PolytheisticReligion implements Religion {
    private $gods;

    public function __construct(Collection $gods) {
        $this->gods = $gods;
    }
}

final class MonotheisticReligion implements Religion {
    private $godInstance;

    public function __construct(God $onlyGod) {
        $this->godInstance = $onlyGod;
    }
}

// According to Christianity and some other religions,
// there's only one God.
// This does not hold for other religions.

$christianGod = new God();
$christianReligion = new MonotheisticReligion($christianGod);
// Under this context God is unique.
// You cannot create or change a new one.
// This is a scoped global.

$jupiter = new God();
$saturn = new God();
$mythologicalReligion = new PolytheisticReligion([$jupiter, $saturn]);

// Gods are unique (or not) according to context
// You can create test religions with or without unicity
// This is less coupled since you break the direct reference to the God class
// God class's single responsibility is to create gods. Not to manage them
```

Singletons are a design antipattern and you should avoid them by policy. You can add linter rules for patterns like `getInstance()` so new developers cannot infect code with this antipattern. Table 17-1 shows a summary of known issues with singletons.

Table 17-1. Some known problems with singletons

Problem	Description
Bijection violation	Singletons don't exist in the real world.
Tight coupling	They provide a hard-to-break global point of access.
Accidental implementation	It is too related to implementation and does not mimic real-world behavior.
Hard testability	It is hard to write unit tests when singletons are present.
Does not save memory	Modern garbage collectors deal better with volatile objects than a permanent one.
Breaks dependency injection	It is more difficult to decouple dependencies.
Instantiation contract violation	When you ask a class to create an instance, you expect a new one.
Fail fast violation	You shouldn't be able to create new instances, and they should fail instead of giving you an old instance.
Implementation coupling	You use `getInstance()` instead of the `new()` method.
More difficult to apply the test-driven development (TDD) technique	TDD deals with coupling and you need to circumvent it when creating new tests.
Unique concepts are contextual	See the previous code example. Uniqueness depends on the scope. And it never should be global.
Multithreaded environment problems	Many singletons are neither thread-safe nor reentrant and cause unexpected behavior.
Accumulate garbage state	Running multiple tests might bloat the singleton, and since it is not collected, the garbage remains.
Class single responsibility violation / separation of concerns	Class's single responsibility is to create instances, not to manage them.
Easy friend for entry point	Once you have a singleton, you can start polluting this convenient global reference with more objects.
Dependency hell	A class depends on a singleton object, and that singleton object depends on another singleton object, and so on.
Lack of flexibility	Once a singleton object is created, it cannot be replaced or modified.
Difficulty in lifecycle management	Managing the lifecycle of a singleton can be challenging and may lead to memory leaks or unnecessary resource utilization.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 12.5, "Removing Design Pattern Abuses"

17.3 Breaking God Objects

Problem

You have an object that knows too much or does too much.

Solution

Don't assign too many responsibilities to a single object.

Discussion

God Objects

God objects have an excessive amount of responsibilities or control over the entire system. These objects tend to be large and complex, with a significant amount of code and logic. They violate the single-responsibility principle (see Recipe 4.7, "Reifying String Validations") and the separation of concerns concept (see Recipe 8.3, "Removing Logical Comments"). God objects tend to become a bottleneck in the software architecture, making the system difficult to maintain, scale, and test.

God objects have cohesion and invite a lot of coupling, merge conflicts, and other maintenance problems. They also violate the bijection principle by having more responsibilities than real-world entities. You need to adhere to the single-responsibility principle and split responsibilities. Notable examples are old routine software libraries.

Here is an example of a God object:

```
class Soldier {
   run() {}
   fight() {}
   driveGeneral() {}
   clean() {}
   fire() {}
   bePromoted() {}
   serialize() {}
   display() {}
   persistOnDatabase() {}
   toXML() {}
   jsonDecode() {}

  // ...
  }
```

Instead, split it, leaving just the essential responsibilities:

```
class Soldier {
   run() {}
   fight() {}
   clean() {}
  }
```

As for the rest of the functions of `Soldier`, you can encapsulate each into a dedicated logic class. Linters can count methods and warn against a threshold for possible God objects. As a notable example, objects whose sole responsibility is to bring a point of entry, like façades, can be God objects. Libraries were fine in the '60s, but in object-oriented programming, you will distribute responsibilities among many objects.

Façade Pattern

The *façade design pattern* provides a simplified interface to a complex system or subsystem. It is used to hide the complexity of a system and provide a simpler interface for clients to use. It also behaves like a mediator between the client and the subsystem, shielding the client from the details of the subsystem's implementation.

Constant classes are a special case. These classes have low cohesion and high coupling, violating the single-responsibility principle (see Recipe 4.7, "Reifying String Validations"). Here you can see a class with too many unrelated constants:

```
public static class GlobalConstants
{
    public const int MaxPlayers = 10;
    public const string DefaultLanguage = "en-US";
    public const double Pi = 3.14159;
}
```

You can break it into smaller classes and then add the relevant behavior to each of them:

```
public static class GameConstants
{
    public const int MaxPlayers = 10;
}

public static class LanguageConstants
{
    public const string DefaultLanguage = "en-US";
}

public static class MathConstants
{
    public const double Pi = 3.14159;
}
```

You can tell your linters to warn you of too many constant definitions against a preset threshold as finding the correct responsibilities is one of your primary tasks when designing software.

Related Recipes

Recipe 6.8, "Replacing Magic Numbers with Constants"

Recipe 10.2, "Removing Settings/Configs and Feature Toggles"

Recipe 11.6, "Breaking Too Many Attributes"

Recipe 17.4, "Breaking Divergent Change"

17.4 Breaking Divergent Change

Problem

You change something in a class and then you also need to change something unrelated in the same class.

Solution

Classes should have just one responsibility and one reason to change. Split the divergent class.

Discussion

Divergent classes have low cohesion, high coupling, and code duplication. These classes violate the single-responsibility principle (see Recipe 4.7, "Reifying String Validations"). You create classes to fulfill responsibilities; if an object does too much, it might change in different directions and you should extract another class.

In the following example, the `Webpage` object has too many responsibilities:

```
class Webpage {
  renderHTML() {
    this.renderDocType();
    this.renderTitle();
    this.renderRssHeader();
    this.renderRssTitle();
    this.renderRssDescription();
    this.renderRssPubDate();
  }
  // RSS format might change
}
```

Here's what it looks like if you break up the responsibilities:

```
class Webpage {
  renderHTML() {
    this.renderDocType();
    this.renderTitle();
    (new RSSFeed()).render();
  }
  // HTML render can change
}

class RSSFeed {
  render() {
    this.renderDescription();
    this.renderTitle();
    this.renderPubDate();
    // ...
  }
  // RSS format might change
  // Might have unitary tests
  // etc.
}
```

You can automatically detect large classes or track changes. Classes must follow the single-responsibility principle (see Recipe 4.7, "Reifying String Validations") and have just one reason to change. If they evolve in different ways, they are doing too much.

Related Recipes

Recipe 11.5, "Removing Excess Methods"

Recipe 11.6, "Breaking Too Many Attributes"

Recipe 11.7, "Reducing Import Lists "

Recipe 17.3, "Breaking God Objects"

See Also

"Divergent Change" on Refactoring Guru

17.5 Converting 9999 Special Flag Values to Normal

Problem

You use constants like `Maxint` to flag invalid IDs, thinking you will never reach it.

Solution

Don't couple real identifiers with invalid ones.

Discussion

Invalid IDs represented with valid ones violate the bijection principle; you might reach the invalid ID sooner than you think. Don't use nulls for invalid IDs either since they cause coupling flags from the caller to functions. You need to model special cases with special polymorphic objects (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic") and avoid `9999`, `-1`, and `0` since they are valid domain objects and produce implementation coupling. In the early days of computing, data types were strict. Then they invented "the billion-dollar mistake" (null—see Chapter 15) and grew up and modeled special scenarios with special values.

The following example uses a special flag number (`9999`):

```
#define INVALID_VALUE 9999

int main(void)
{
    int id = get_value();
    if (id == INVALID_VALUE)
    {
        return EXIT_FAILURE;
        // id is a flag and also a valid domain value
    }
    return id;
}
int get_value()
{
  // something bad happened
  return INVALID_VALUE;
}
// returns EXIT_FAILURE (1)
```

Here's what it looks like when you remove the need for a special valid value (`9999`) and replace it with an invalid value (`-1`):

```
int main(void)
{
   int id = get_value();
   if (id < 0)
   {
      printf("Error: Failed to obtain value\n");
      return EXIT_FAILURE;
   }
   return id;
}
int get_value()
{
   // something bad happened
   return -1;  // Return a negative value to indicate error
}
```

Even better if your language supports exceptions (see Chapter 22, "Exceptions"):

```
// No INVALID_VALUE defined

int main(void)
{
   try {
      int id = get_value();
      return id;
   } catch (const char* error) {
        printf("%s\n", error);
        return EXIT_FAILURE;
   }
}

int get_value()
{
  // something bad happened
   throw "Error: Failed to obtain value";
}

// returns EXIT_FAILURE (1)
```

External identifiers are usually mapped to numbers or strings. But if the external identifier is missing, don't try to use a number or string as (an invalid) reference.

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 25.2, "Changing Sequential IDs"

Recipe 15.5, "Representing Unknown Locations Without Using Null"

17.6 Removing Shotgun Surgery

Problem

A single functional change leads to multiple code changes in your software.

Solution

Isolate the changes and follow the don't repeat yourself (DRY) principle (see Recipe 4.7, "Reifying String Validations").

Shotgun Surgery

Shotgun surgery describes a situation where a single change in the codebase requires multiple changes in different parts across the system. It happens when changes to one part of the codebase affect many other parts of the system. It's analogous to firing a shotgun: a single blast can hit multiple targets at once, just as a single code change can affect multiple parts of the system.

Discussion

If you have poor responsibility assignments or code duplication, you may need to make too many modifications in your model when some real-world behavior changes. This is a sign of a bad bijection, as defined in Chapter 2. It usually happens when you have copy-pasted code across the system.

Imagine you need to make some changes to the following code:

```
final class SocialNetwork {

    function postStatus(string $newStatus) {
        if (!$user->isLogged()) {
            throw new Exception('User is not logged');
        }
        // ...
    }

    function uploadProfilePicture(Picture $newPicture) {
        if (!$user->isLogged()) {
            throw new Exception('User is not logged');
        }
        // ...
    }

    function sendMessage(User $recipient, Message $messageSend) {
        if (!$user->isLogged()) {
            throw new Exception('User is not logged');
        }
        // ...
    }
}
```

Here's what it looks like when you extract the logic to a single place:

```
final class SocialNetwork {

    function postStatus(string $newStatus) {
        $this->assertUserIsLogged();
        // ...
    }

    function uploadProfilePicture(Picture $newPicture) {
        $this->assertUserIsLogged();
        // ...
    }

    function sendMessage(User $recipient, Message $messageSend) {
        $this->assertUserIsLogged();
        // ...
    }

    function assertUserIsLogged() {
        if (!$this->user->isLogged()) {
            throw new Exception('User is not logged');
            // This is just a simplification
            // Operations should be defined as objects with preconditions etc.
        }
    }
}
```

Some modern linters as well as generative machine learning tools can detect repeated patterns (not just repeated code), and also, while performing your code reviews, you can easily detect this problem and ask for a refactoring. Adding a new feature should be straightforward if your model maps 1:1 to the real world and your responsibilities are in the correct places. You should be alert for small changes spanning several classes.

17.7 Removing Optional Arguments

Problem

You have optional arguments in your functions.

Solution

Optional arguments generate a hidden coupling in the name of more compact code.

Discussion

When you have optional arguments, you couple the calling method to accidental optional values, generating unexpected results, side effects, and ripple effects. In languages with optional arguments but limited to basic types, you need to set a flag and add an accidental if. As a solution, you need to make arguments explicit and use *named parameters* if your language supports them.

Here there's an optional validation policy:

```
final class Poll {

    function _construct(
        array $questions,
        bool $annonymousAllowed = false,
        $validationPolicy = 'Normal') {

        if ($validationPolicy == 'Normal') {
          $validationPolicy = new NormalValidationPolicy();
        }
        // ...
    }
}

// Valid
new Poll([]);
new Poll([], true);
new Poll([], true , new NormalValidationPolicy());
new Poll([], , new StrictValidationPolicy());
```

When you make the argument explicit there are no hidden assumptions:

```
final class Poll {

    function _construct(
        array $questions,
        AnonyomousStrategy $anonymousStrategy,
        ValidationPolicy $validationPolicy) {
        // ...
    }
}

// invalid
new Poll([]);
new Poll([], new AnonyomousInvalidStrategy());
new Poll([], , new StrictValidationPolicy());

// Valid
new Poll([], new AnonyomousInvalidStrategy(), new StrictValidationPolicy());
```

Detection is easy if the language supports optional arguments. You always need to be explicit and favor readability over shorter (and more coupled) function calls.

Related Recipes

Recipe 17.10, "Moving Default Arguments to the End"

Recipe 21.3, "Removing Warning/Strict Off"

17.8 Preventing Feature Envy

Problem

You have one object using too many methods of another object.

Solution

Break the dependency and refactor the behavior.

Feature Envy

Feature envy happens when an object is more interested in the behavior of another object than its own by excessively using another object's methods.

Discussion

Feature envy creates significant dependency and coupling, damaging code reuse and testability. It is usually a symptom of bad responsibility assignments. You need to find the responsibilities using MAPPER and move the methods to the appropriate class.

This candidate defines how to print its address:

```
class Candidate {
   void printJobAddress(Job job) {
     System.out.println("This is your position address");
     System.out.println(job.address().street());
     System.out.println(job.address().city());
     System.out.println(job.address().ZipCode());
   }
}
```

The printing responsibility belongs to the related job:

```
class Job {

  void printAddress() {
     System.out.println("This is your job position address");
     System.out.println(this.address().street());
     System.out.println(this.address().city());
     System.out.println(this.address().ZipCode());
     // You might even move this responsibility directly to the address!
     // Some address information is relevant to a job for package tracking
   }
}

class Candidate {
  void printJobAddress(Job job) {
    job.printAddress();
    }
  }
```

Here is another example where the area of the rectangle is computed with an external formula:

```
function area(rectangle) {
   return rectangle.width * rectangle.height;
   // Notice you are sending consecutive messages to
   // the same object and doing calculations
}
```

Here's what it looks like using separation of concerns:

```
class Rectangle {
    constructor(width, height) {
         this.height = height;
         this.width = width;
    }
    area() {
        return this.width * this.height;
    }
}
```

Some linters can detect a sequential pattern of collaborations with another object.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 6.5, "Changing Misplaced Responsibilities"

Recipe 14.15, "Changing Equal Comparison"

Recipe 17.16, "Breaking Inappropriate Intimacy"

17.9 Removing the Middleman

Problem

You have a *middleman* object creating unnecessary indirection.

Solution

Remove the intermediary.

Discussion

Middleman objects break Demeter's law (see Recipe 3.8, "Removing Getters"), adding complexity and unnecessary indirection and creating empty classes. You should remove them. Here is an example where the client has an address (which in turn has a zip code). The `Address` class is an anemic model and its only responsibility is to return the zip code:

```
public class Client {
    Address address;
    public ZipCode zipCode() {
        return address.zipCode();
    }
}

public class Address {
   // Middleman
    private ZipCode zipCode;

    public ZipCode zipCode() {
        return new ZipCode('CA90210');
    }
}

public class Application {
   ZipCode zipCode = client.zipCode();
}
```

Here the client exposes the address:

```
public class Client {
    public ZipCode zipCode() {
        // Can also store it
        return new ZipCode('CA90210');
    }
}

public class Application {
   ZipCode zipCode = client.zipCode();
}
```

Similar to its opposite (Recipe 10.6, "Breaking Long Chains of Collaborations"), you can detect this smell using parsing trees.

Related Recipes

Recipe 10.6, "Breaking Long Chains of Collaborations"

Recipe 10.9, "Removing Poltergeist Objects"

Recipe 19.9, "Migrating Empty Classes"

See Also

"Remove Middle Man" on Refactoring.com

"Middle Man" on C2 Wiki

"Remove Middleman" on JetBrains

17.10 Moving Default Arguments to the End

Problem

You have default arguments in the middle of the arguments list.

Solution

Function signatures should not be error-prone. Try not to use default arguments (see Recipe 17.7, "Removing Optional Arguments"), but if you really need them, don't use optional arguments before mandatory ones.

Discussion

Default arguments might fail unexpectedly, violating the fail fast principle. They also hurt readability since you need to think a lot about parameter sorting. You need to move your optional arguments last or remove them using the related Recipe 17.7, "Removing Optional Arguments".

Here you can see the optional color before the model:

```
function buildCar($color = "red", $model) {
  //...
}
// First argument with optional argument

buildCar("Volvo");
// Runtime error: Too few arguments to function buildCar()
```

By moving it to the last position you avoid the problem:

```
function buildCar($model, $color = "Red", ){...}

buildCar("Volvo");
// Works as expected

def functionWithLastOptional(a, b, c='foo'):
    print(a)
    print(b)
    print(c)

functionWithLastOptional(1, 2) // Prints 1, 2, foo

def functionWithMiddleOptional(a, b='foo', c):
    print(a)
    print(b)
    print(c)
```

```
functionWithMiddleOptional(1, 2)
# SyntaxError: non-default argument follows default argument
```

Many linters can enforce this rule since you can derive it from the function signature. Also, many compilers directly forbid it. Try to be strict when defining functions to avoid coupling between the caller and the optional value defined by the method.

Related Recipes

Recipe 9.5, "Unifying Parameter Order"

Recipe 17.7, "Removing Optional Arguments"

See Also

"Method Arguments with Default Values Should Be Last" on Sonar Source

17.11 Avoiding the Ripple Effect

Problem

You make small changes to your code and notice too many unexpected problems.

Solution

If small changes have a big impact, you need to decouple your system.

Discussion

The ripple effect is a problem you can address using many of the recipes in this book. To avoid it, you always need to decouple the changes covering the existing functionality with tests and then refactor and isolate what is changing.

Here is a popular object coupled to an accidental implementation to get the current time:

```
class Time {
   constructor(hour, minute, seconds) {
     this.hour = hour;
     this.minute = minute;
     this.seconds = seconds;
    }
    now() {
      // call operating system
    }
}

// Adding a time zone will have a big ripple effect
// Changing now() to consider time zone will also create the effect
```

Here's what it looks like when you remove the now method:

```
class Time {
   constructor(hour, minute, seconds, timezone) {
     this.hour = hour;
     this.minute = minute;
     this.seconds = seconds;
     this.timezone = timezone;
  }
  // Removed now() since it is invalid without context
}

class RelativeClock {
   constructor(timezone) {
     this.timezone = timezone;
   }
   now(timezone) {
     var localSystemTime = this.localSystemTime();
     var localSystemTimezone = this.localSystemTimezone();
     // Do some math translating time zones
     // ...
     return new Time(..., timezone);
   }
}
```

It is not easy to detect problems before they happen. Mutation testing (see Recipe 5.1, "Changing var to const") and root cause analysis of single points of failure may help. There are multiple strategies to deal with legacy and coupled systems. You should deal with this problem before it explodes in your face.

Single Point of Failure

A *single point of failure* refers to a component or part of a system that, if it were to fail, would cause the entire system to fail or become unavailable. The system is dependent on this component or part, and without it, nothing can function properly. Good designs try to have redundant components to avoid this ripple effect.

Related Recipes

Recipe 10.6, "Breaking Long Chains of Collaborations"

17.12 Removing Accidental Methods on Business Objects

Problem

You have persistence, serialization, displaying, importing, logging, or exporting code in your domain objects.

Solution

Remove all accidental methods. They belong to different domains, so they should be in different objects.

Discussion

Coupling objects to accidental problems makes software less maintainable and less readable, hence you should not mix accidental and essential behavior. You need to decouple business objects. Separate accidental concerns: move persistence, formatting, and serialization to special objects and keep essential protocol using the bijection.

Here you have a `car` bloated with accidental protocol. Cars and accidents should always be apart:

```
class car:

    def __init__(self, company, color, engine):
        self._company = company
        self._color = color
        self._engine = engine

    def goTo(self, coordinate):
        self.move(coordinate)

    def startEngine(self):
        ## code to start engine
        self.engine.start()

    def display(self):
        ## displaying is accidental
        print ('This is a', self._color, self.company)

    def to_json(self):
        ## serializing is accidental
        return "json"

    def update_on_database(self):
        ## persistence is accidental
        Database.update(this)

    def get_id(self):
        ## identifiers are accidental
        return self.id;

    def from_row(self, row):
        ## persistence is accidental
        return Database.convertFromRow(row);

    def forkCar(self):
        ## concurrency is accidental
        ConcurrencySemaphoreSingleton.get_instance().fork_cr(this)
```

Here's what it looks like when you strip the methods:

```
class car:

    def __init__(self,company,color,engine):
        self._company = company
        self._color = color
        self._engine = engine

    def goTo(self, coordinate):
        self.move(coordinate)

    def startEngine(self):
        ## code to start engine
        self._engine.start()
```

It is difficult (but not impossible) to create linting rules based on naming that provides hints for suspicious names. As exceptions, some frameworks force you to inject dirty code into objects, for example, identifiers. You might be very used to seeing polluted business objects. This is normal. You need to reconsider the consequences and coupling of these designs.

17.13 Removing Business Code from the User Interface

Problem

You have input validations on your user interface.

Solution

You should always create proper objects in your backends and move the validations to your domain objects.

Discussion

UIs are accidental. Validating business rules on the UI brings security problems and potential code duplication. This kind of coupling damages the testability and extensibility of APIs, microservices, etc., making business objects anemic and mutable. Code duplication is a warning for premature optimization. Building a system with UI validations might evolve into an API or external component consumption; you need to validate objects on the backend and send good validation messages to client components.

Here is an example of validations on the UI:

```
<script type="text/javascript">

function checkForm(form)
{
  if(form.username.value == "") {
    alert("Error: Username cannot be blank!");
    form.username.focus();
    return false;
  }
  re = /^\w+$/;
  if(!re.test(form.username.value)) {
    alert("Error: Username must contain only letters,"
      + " numbers and underscores!");
    form.username.focus();
    return false;
  }

  if(form.pwd1.value != "" && form.pwd1.value == form.pwd2.value) {
    if(form.pwd1.value.length < 8) {
      alert("Error: Password must contain at least eight characters!");
      form.pwd1.focus();
      return false;
    }
    if(form.pwd1.value == form.username.value) {
      alert("Error: Password must be different from Username!");
      form.pwd1.focus();
      return false;
    }
    re = /[0-9]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least one number (0-9)!");
      form.pwd1.focus();
      return false;
    }
    re = /[a-z]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least"
            + " one lowercase letter (a-z)!");
      form.pwd1.focus();
      return false;
    }
    re = /[A-Z]/;
    if(!re.test(form.pwd1.value)) {
      alert("Error: password must contain at least"
            + " one uppercase letter (A-Z)!");
      form.pwd1.focus();
      return false;
    }
  } else {
    alert("Error: Please check that you've entered"
          +" and confirmed your password!");
    form.pwd1.focus();
```

```
      return false;
    }

    alert("You entered a valid password: " + form.pwd1.value);
    return true;
  }

  </script>

  <form ... onsubmit="return checkForm(this);">
  <p>Username: <input type="text" name="username"></p>
  <p>Password: <input type="password" name="pwd1"></p>
  <p>Confirm Password: <input type="password" name="pwd2"></p>
  <p><input type="submit"></p>
  </form>
```

Here's what it looks like after you move the validation code to your business objects:

```
  <script type="text/javascript">

    // Send a post to a backend
    // Backend has domain rules
    // Backend has test coverage and rich models
    // It is more difficult to inject code in a backend
    // Validations will evolve on your backend
    // Business rules and validations are shared with every consumer
    // UI / REST / Tests / Microservices ... etc. etc.
    // No duplicated code
    function checkForm(form)
    {
      const url = "https://<hostname/login";
      const data = { };

      const other_params = {
          headers : { "content-type" : "application/json; charset=UTF-8" },
          body : JSON.stringify(data),
          method : "POST",
          mode : "cors"
      };

      fetch(url, other_params)
          .then(function(response) {
              if (response.ok) {
                  return response.json();
              } else {
                  throw new Error("Could not reach the API: " +
                      response.statusText);
              }
          }).then(function(data) {
              document.getElementById("message").innerHTML = data.encoded;
          }).catch(function(error) {
              document.getElementById("message").innerHTML = error.message;
          });
      return true;
    }

  </script>
```

As notable exceptions, if you have strong evidence of severe performance bottlenecks you need to automatically duplicate your business logic on the frontend; you cannot just skip the backend part. You should not do this manually because you will forget to do it. Use TDD (see Recipe 4.8, "Removing Unnecessary Properties") where you put all your business logic behavior on your domain objects.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.6, "Removing DTOs"

Recipe 6.13, "Avoiding Callback Hell"

Recipe 6.14, "Generating Good Error Messages"

Recipe 16.8, "Removing Callback Events Based on Implementation"

17.14 Changing Coupling to Classes

Problem

You have global classes and you use them as entry points.

Solution

Break the coupling and use objects like interfaces instead of classes as they are more easily replaceable.

Discussion

Global classes create coupling and prevent extensibility; they are also hard to mock (see Recipe 20.4, "Replacing Mocks with Real Objects"). You can use interfaces or traits (if available) and dependency inversion (see Recipe 12.4, "Removing One-Use Interfaces") to favor loose coupling.

Here you have coupling:

```
public class MyCollection {
     public bool HasNext { get; set;} // implementation details
     public object Next(); // implementation details
}

public class MyDomainObject sum(MyCollection anObjectThatCanBeIterated) {
 // Tight coupling
}

// We cannot fake or mock this method
// since it always expects an instance of MyCollection
```

Here's what it looks like when you mock it:

```
public interface Iterator {
     public bool HasNext { get; set;}
     public object Next();
}

public Iterator Reverse(Iterator iterator) {
    var list = new List<int>();
    while (iterator.HasNext) {
       list.Insert(0, iterator.Next());
    }
    return new ListIterator(list);
}

public class MyCollection implements Iterator {
     public bool HasNext { get; set;} // Implementation details
     public object Next(); // Implementation details
}

public class myDomainObject {
     public int sum(Iterator anObjectThatCanBeIterated) {
     // Loose coupling
    }
}

// Can use any Iterator (even a mocked one as long as it adheres to the protocol)
```

You can use almost any linter to find references to classes. You should not abuse it since many legitimate uses might raise false positives. Dependencies to interfaces make a system less coupled and thus more extensible and testable. Interfaces change less often than concrete implementations. Some objects implement many interfaces, and declaring which part depends on which interface makes the coupling more granular and the object more cohesive.

Related Recipes

Recipe 20.4, "Replacing Mocks with Real Objects"

Loose Coupling

Loose coupling aims to minimize the interdependence of different objects within a system. They have minimal knowledge about each other, and changes made to one component do not affect other components in the system, preventing a ripple effect.

17.15 Refactoring Data Clumps

Problem

You have some objects that are always together.

Solution

Make cohesive primitive objects; their parts will always travel together.

Discussion

Data Clump

In *data clumps*, the same group of objects is frequently passed around between different parts of a program. This can lead to increased complexity, reduced maintainability, and a higher risk of errors. Data clumps often occur when you try to pass around related objects without finding a proper object representing that relationship in the bijection.

Data clumps have bad cohesion, primitive obsession, and lots of duplicated code; you duplicate complex validation in several places, damaging readability and maintainability. You can apply the "Extract Class" refactoring and Recipe 4.1, "Creating Small Objects". If two or more primitive objects are glued together, with repeated business logic and rules between them, you need to find the existing concept of the bijection.

Here is an example of a data clump:

```
public class DinnerTable
{
    public DinnerTable(Person guest, DateTime from, DateTime to)
    {
        Guest = guest;
        From = from;
        To = to;
    }
    private Person Guest;
    private DateTime From;
    private DateTime To;
}
```

Here's what it looks like after you reify it:

```
public class TimeInterval
{
    public TimeInterval(DateTime from, DateTime to)
    {
        if (from >= to)
        {
            throw new ArgumentException
                ("Invalid time interval: 'from' must be earlier than 'to'.");
        }
        From = from;
        To = to;
    }
}

public class DinnerTable
{
    public DinnerTable(Person guest, DateTime from, DateTime to)
    {
        Guest = guest;
        Interval = new TimeInterval(from, to);
    }
}
```

Here's an even better and more compact version where you directly pass the interval:

```
public DinnerTable(Person guest, Interval reservationTime)
{
    Guest = guest;
    Interval = reservationTime;
}
```

You should group behavior in the right place and hide the primitive data.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 4.2, "Reifying Primitive Data"

Recipe 4.3, "Reifying Associative Arrays"

17.16 Breaking Inappropriate Intimacy

Problem

You have two classes with too much interdependence.

Solution

Break the classes.

Discussion

Inappropriate Intimacy

Inappropriate intimacy happens when two classes or components have become overly dependent on each other, creating a tight coupling that makes the code difficult to maintain, modify, or extend.

When two classes interact with each other too much, they are too coupled. This is a sign of incorrect responsibility assignments and low cohesion, which hinders maintainability and extensibility. Here are two classes entwined:

```
class Candidate {

 void printJobAddress(Job job) {

   System.out.println("This is your position address");

   System.out.println(job.address().street());
   System.out.println(job.address().city());
   System.out.println(job.address().zipCode());

   if (job.address().country() == job.country()) {
        System.out.println("It is a local job");
   }
}
```

Here's what it looks like when you break the coupling:

```
final class Address {
 void print() {
   System.out.println(this.street);
   System.out.println(this.city);
   System.out.println(this.zipCode);
 }

 bool isInCounty(Country country) {
  return this.country == country;
}
```

```
class Job {
 void printAddress() {

   System.out.println("This is your position address");

   this.address().print());

   if (this.address().isInCountry(this.country()) {
        System.out.println("It is a local job");
   }
 }
}

class Candidate {
  void printJobAddress(Job job) {
    job.printAddress();
  }
}
```

Some linters compute graph class relations and protocol dependency. Analyzing the collaboration graph, you can infer rules and hints. If two classes are too related and don't talk much to others, you might need to split, merge, or refactor them. Classes should know as little about each other as possible.

Related Recipes

Recipe 17.8, "Preventing Feature Envy"

See Also

"Inappropriate Intimacy" on C2 Wiki

17.17 Converting Fungible Objects

Problem

You distinguish two objects when your partial model of the real world does not require it.

Solution

Respect the MAPPER and make what is fungible in the real world fungible in your model.

Discussion

Fungible Objects

Fungible objects are interchangeable or identical in value, quality, and characteristics. Any particular instance of a fungible object can be replaced by any other instance of the same object without any loss of value or quality. Fungibility is the property of a good or a commodity whose individual units are essentially interchangeable and each of whose parts is indistinguishable from any other part.

You might have heard a lot about NFTs. Let's use this fungible concept in your code. Fungibility is a property you need to enforce on the bijection. An object should be fungible in your model if it is in the real world. You need to identify fungible elements on your domains and model them as interchangeable. In software, you can replace fungible objects with others. When mapping your objects with real ones, it's easy to forget about the partial model and overdesign.

Here you have a nonfungible `Person`:

```
public class Person implements Serializable {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

shoppingQueueSystem.queue(new Person('John', 'Doe'));
```

It is not important to model the `Person` in this context:

```
public class Person {
}

shoppingQueueSystem.queue(new Person());
// The identity is irrelevant for queue simulation
```

You need to understand the model to check whether it is right or not. Make fungible objects for what is actually fungible. This sounds easy, but it requires design skills.

Related Recipes

Recipe 4.8, "Removing Unnecessary Properties"

CHAPTER 18
Globals

Write shy code—modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations.

—David Thomas and Andrew Hunt, *The Pragmatic Programmer: Your Journey to Mastery*

18.0 Introduction

Most modern languages support global functions, classes, and attributes. There's a hidden cost when you use any of these artifacts. Even when you are creating an object using `new()`, you will introduce a tight coupling to a global class unless you apply some of the following recipes.

18.1 Reifying Global Functions

Problem

You have global functions that you can call from anywhere.

Solution

Global functions bring a lot of coupling. Narrow their scope.

Discussion

Discouraged by object-oriented programming, many mixed languages support global functions. They create coupling and hurt readability since they are difficult to trace. As coupling grows, maintainability and testability become more problematic. You can start by wrapping the function in a context object. For example, you can find external resource access, database access, singletons (see Recipe 17.2, “Replacing Singletons”), global classes, time, and operating system resources.

This example calls a method from a global database:

```
class Employee {
    function taxesPayedUntilToday() {
        return database()->select(
            "SELECT TAXES FROM EMPLOYEE".
            " WHERE ID = " . $this->id() .
            " AND DATE < " . currentDate());
    }
}
```

By making persistence contextual, you can decouple the database from the calculation logic:

```
final class EmployeeTaxesCalculator {
    function taxesPayedUntilToday($context) {
        return $context->selectTaxesForEmployeeUntil(
            $this->socialSecurityNumber,
            $context->currentDate());
    }
}
```

Many modern languages avoid global functions. For the permissive ones, scope rules can be applied and automatically checked. Structured programming considers global functions harmful. Yet, you can observe that some bad practices cross paradigm boundaries.

Related Recipes

Recipe 5.7, “Removing Side Effects”

Recipe 18.4, “Removing Global Classes”

18.2 Reifying Static Functions

Problem

You have static functions that are coupled to classes.

Solution

Don’t use static functions. They are global and utilities. Create instance methods and talk to objects instead.

Discussion

Static Functions

A *static function* belongs to a class rather than an instance of that class. This means that a static method can be called without creating an object of the class.

Classes map real-world concepts (or ideas), and static functions violate the bijection principle. Static functions also bring coupling. They make code more difficult to test since classes are harder to mock (see Recipe 20.4, "Replacing Mocks with Real Objects") than instances. A class's single responsibility (see Recipe 4.7, "Reifying String Validations") is to create instances. You can delegate methods to instances or create stateless objects, following responsibilities in the real world. Don't call them helpers or utils.

Here is a class with a static method format:

```
class DateStringHelper {
   static format(date) {
     return date.toString('yyyy-MM-dd');
  }
}

DateStringHelper.format(new Date());
```

You can pull the responsibility into a specific object:

```
class DateToStringFormatter {
   constructor(date) {
      this.date = date;
   }

   englishFormat() {
     return this.date.toString('yyyy-MM-dd');
  }
}

new DateToStringFormatter(new Date()).englishFormat();
```

You can enforce a policy to avoid static methods (all class methods except constructors). Classes are globals in disguise. Polluting their protocol with utils/helpers/library methods breaks cohesion and generates coupling. You should extract static functions with refactorings. In most languages you cannot manipulate classes and use them polymorphically, so you can't mock them (see Recipe 20.4, "Replacing Mocks with Real Objects") or plug them into tests. Therefore, you have a global reference too difficult to decouple.

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 20.1, "Testing Private Methods"

18.3 Replacing GoTo with Structured Code

Problem

You have GoTo sentences in your code.

Solution

Don't ever use GoTo. Structure your code without global or local jumps.

Discussion

GoTo has been considered harmful for at least 50 years. The sentence damages readability and makes code hard to follow. You can use structured code instead and use exceptions if necessary. GoTo was heavily abused in popular languages like BASIC several decades ago, and low-level languages use them as jump control.

Here is a function with a GoTo instance:

```
int i = 0;

start:
if (i < 10)
{
    Console.WriteLine(i);
    i++;
    goto start;
}
```

Here is the same algorithm written in a more structured way:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

GoTo problems were acknowledged decades ago. But the problem is still present in modern languages like GoLang, PHP, Perl, C#, etc. Most programmers luckily avoid GoTo sentences by using structured programming.

Related Recipes

Recipe 15.1, "Creating Null Objects"

See Also

"Go To Statement Considered Harmful" by Edgar Dijkstra

18.4 Removing Global Classes

Problem

You use classes as a global access point.

Solutions

Don't use your classes as a global point of access. Use specialized objects instead.

Discussion

Classes are global or semi-global if they are scoped by namespaces, packages, or modules. As with other globals, the biggest problem is coupling. They pollute the namespace if you have many of them together. They are also a free ride for static methods, static constants, and singletons (see Recipe 17.2, "Replacing Singletons"). You can use namespaces, module qualifiers, or similar to avoid namespace pollution. Always keep the global names as short as possible. Remember, the single responsibility of a class (see Recipe 4.7, "Reifying String Validations") is to create instances, not to be a global point of access.

Namespaces

Namespaces are used to organize code elements such as classes, functions, and variables into logical groups, preventing naming conflicts and providing a way to uniquely identify them within a particular scope. They help to create modular and maintainable code by grouping related functionality together.

Here is a global point of access with a static method:

```
final class StringUtilHelper {
    static function formatYYYYMMDD($dateToBeFormatted): string {
    }
}
```

You can scope the class and also move the static method to an instance:

```
namespace Dates;

final class DateFormatter {
   // class DateFormatter is no longer global
   public function formatYYYYMMDD(\DateTime $dateToBeFormatted): string {
   }
    // function is not static since class single responsibility
    // is to create instances and not be a library of utils
}
```

This is much better when you convert the `DateFormatter` to a method object:

```
namespace Dates;

final class DateFormatter {
    private $date;

    public function __construct(\DateTime $dateToBeFormatted) {
        $this->date = $dateToBeFormatted;
    }

    public function formatYYYYMMDD(): string {
    }
}
```

When invoking the scoped class, you will have a clear relation between modules/namespaces:

```
use Dates\DateFormatter;
// Since DateFormatter is no longer global you need full qualification
// You can even solve name collisions

$date = new DateTime('2022-12-18');
$dateFormatter = new DateFormatter($date);
$formattedDate = $dateFormatter->formatYYYYMMDD();
```

All singletons (see Recipe 17.2, "Replacing Singletons") are also global points of access:

```
class Singleton { }

final class DatabaseAccessor extends Singleton { }
```

You can access the database in a narrow scope without invoking the class:

```
namespace OracleDatabase;

class DatabaseAccessor {
    // Database is not a singleton and it is namespace scoped
}
```

You can use almost any linter or create dependency rules to search for bad class references. You should restrict classes to small domains and expose just façades to the outside. This greatly reduces coupling.

Related Recipes

Recipe 18.1, "Reifying Global Functions"

Recipe 18.2, "Reifying Static Functions"

Recipe 19.9, "Migrating Empty Classes"

18.5 Changing Global Date Creation

Problem

You use `new Date()` in your code.

Solution

Avoid creating empty dates. Provide explicit context explaining what your time source is.

Discussion

Creating a date without context creates coupling and hidden assumptions on global systems. Many systems run in a cloud environment where the time zone is not always explicit. Here is a classic example of the usage:

```
var today = new Date();
```

Instead, you should make it explicit:

```
var ouagadougou = new Location();
var today = timeSource.currentDateIn(ouagadougou);

function testGivenAYearHasPassedAccruedInterestsAre10() {
  var mockTime = new MockedDate(new Date(2021, 1, 1));
  var domainSystem = new TimeSystem(mockTime);
  // ..

  mockTime.moveDateTo(new Date(2022, 1, 1));

  // … You set up the yearly interest rate
  assertEquals(10, domainSystem.accruedInterests());
}
```

You should forbid global functions on policies because it forces you to couple to accidental and pluggable time sources. `date.today()`, `time.now()`, and other global system calls create coupling. Since tests must be in full environmental control you should be able to easily set up time, move it back and forth, etc.

`Date` and `Time` classes should only create immutable instances. It is not their responsibility to give the actual time. This violates the single-responsibility principle (see Recipe 4.7, "Reifying String Validations"). The passage of time is always scorned by programmers. This makes objects mutable and leads to poor, coupled designs.

Testing Full Environmental Control

Full environmental control is the ability to have complete control over the environment in which tests are executed. It involves creating a controlled and predictable environment that allows tests to run consistently and independently of external factors. You need to especially consider external dependencies, network simulation, database isolation, time control, and many others.

Related Recipes

Recipe 4.5, "Reifying Timestamps"

Recipe 18.2, "Reifying Static Functions"

CHAPTER 19

Hierarchies

One aspect of a class was to act as a repository for code and information shared by all instances (objects) of that class. In terms of efficiency, this is a good idea because storage space is minimized and changes can be made in a single place. It becomes very tempting, however, to use this fact to justify creating your class hierarchy based on shared code instead of shared behaviors.. Always create hierarchies based on shared behaviors.

—David West, *Object Thinking*

19.0 Introduction

Class inheritance is often wrongly used for code reuse due to historical reasons. You should prefer composition, but it is not that obvious and takes more experience. Composition is dynamic and you can easily change it, test it, reuse it, etc., giving flexibility to your designs. In this chapter you will find recipes to minimize the accidental coupling you add by using hierarchies.

19.1 Breaking Deep Inheritance

Problem

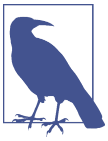
You have deep hierarchies to reuse code.

Solution

Find the protocol and flatten the hierarchy by favoring composition over inheritance.

Discussion

Static subclassification reuse creates more coupling than dynamic composition reuse. Deep hierarchies have bad cohesion and fragile base classes. They bring method overriding and violate the Liskov substitution principle (one of SOLID's fundamentals). You need to break the classes and compose them. In the past, some articles and books recommended using classes as a specialization for code reuse, but composition is a more efficient and extensible way to share behavior.



Liskov Substitution Principle

The *Liskov substitution principle* states that if a function or method is designed to work with objects of a particular class, then it should also work with objects of any subclass of that class without causing any unexpected behavior. It is the "L" from SOLID (see Recipe 4.7, "Reifying String Validations").

Here is a deep hierarchy representing seals according to scientific classification:

```
class Animalia:
class Chordata(Animalia):
class Mammalia(Chordata):
class Carnivora(Mammalia):
class Pinnipedia(Carnivora):
class Phocidae(Pinnipedia):
class Halichoerus(Phocidae):
class GreySeal(Halichoerus):
```

Here's what it looks like after you compress them until you discover behavior for each class in the hierarchy:

```
class GreySeal:
    def eat(self):    # find the common behavior in the hierarchy
    def sleep(self):  # find the common behavior in the hierarchy
    def swim(self):   # find the common behavior in the hierarchy
    def breed(self):  # find the common behavior in the hierarchy
```

The grey seal is featured on the cover of this book. Many linters report a depth of inheritance tree (DIT). You can look after your hierarchies and break them often. Here you use classification for an accidental reason (how you charge the users of your servers):

```
class Server:
    @abstractmethod
    def calculate_cost(self):
        pass

class DedicatedServer(Server):
    def calculate_cost(self):
        # Example: cost based on CPU and RAM usage
        return self.cpu * 10 + self.ram * 5

class HourlyChargedServer(Server):
    def calculate_cost(self):
        # Example: cost based on CPU and RAM usage multiplied by hours
        return (self.cpu * 5 + self.ram * 2) * self.hours
# Once the server is created you cannot dynamically change the charging method
# If you create a new ChargingMethod it will impact your server hierarchy
```

Here you use composition to dynamically change the charging method:

```
class Server:
    def calculate_cost(self):
        return self.charging.calculate_cost(self.cpu, self.ram)
   def change_charging_method(self, charging):
        self.charging = charging

class ChargingMethod():
    @abstractmethod
    def calculate_cost(self, cpu, ram):
        pass

class MonthlyCharging(Charging):
    def calculate_cost(self, cpu, ram):
        return cpu * 10 + ram * 5

class HourlyCharging(Charging):
    def calculate_cost(self, cpu, ram):
        return (cpu * 5 + ram * 2) * self.hours
# You can unit test the charging methods in isolation
# You can create new charging methods without impacting the servers
```

Composition gives you more flexibility, testability, and reuse, and also favors the open-closed principle (see Recipe 14.3, "Reifying Boolean Variables") since you don't change your domain hierarchy and only change the delegated object.

Composition

Composition allows objects to be composed of other objects as parts or components. You build complex objects by combining simpler ones (see Recipe 4.1, "Creating Small Objects"), forming a "has-a" relationship instead of the classic "is-a" or "behaves-as-a" (see Recipe 19.4, "Replacing "is-a" Relationship with Behavior").

Related Recipes

Recipe 19.2, "Breaking Yo-Yo Hierarchies"

Recipe 19.3, "Breaking Subclassification for Code Reuse"

Recipe 19.4, "Replacing "is-a" Relationship with Behavior"

Recipe 19.7, "Making Concrete Classes Final"

Recipe 19.11, "Removing Protected Attributes"

19.2 Breaking Yo-Yo Hierarchies

Problem

When you search for a concrete method implementation, you need to go back and forth, up and down on the hierarchy like a yo-yo.

Solution

Don't create deep hierarchies. Compact them.

Discussion

The Yo-Yo Problem

The *yo-yo problem* occurs when you need to navigate classes and methods in a class hierarchy to understand or modify the code, making it difficult to maintain and extend the codebase.

Deep hierarchies create subclassification for code reuse and hurt readability. Programming through small differences makes classes less cohesive. You need to favor composition over inheritance and refactor deep hierarchies.

Here is an example of a specialized hierarchy:

```
abstract class Controller { }

class BaseController extends Controller { }
class SimpleController extends BaseController { }
class ControllerBase extends SimpleController { }
class LoggedController extends ControllerBase { }
class RealController extends LoggedController { }
```

By using interfaces you favor delegation and avoid the yo-yo problem:

```
interface ControllerInterface { }

abstract class Controller implements ControllerInterface { }
final class LoggedControllerDecorator implements ControllerInterface { }
final class RealController implements ControllerInterface { }
```

Any linter can check for suspects against a max depth threshold. Many novice programmers reuse code through hierarchies. This brings highly coupled and low cohesion hierarchies. Johnson and Foote established in their 1988 paper the usefulness of this recipe. Developers have learned a lot from there. You must refactor and flatten those hierarchies.

Related Recipes

Recipe 19.1, “Breaking Deep Inheritance”

Recipe 19.3, “Breaking Subclassification for Code Reuse”

See Also

“Designing Reusable Classes” by Ralph E. Johnson and Brian Foote

19.3 Breaking Subclassification for Code Reuse

Problem

You have an “is-a” relation and reuse code through subclassification.

Solution

Favor composition over inheritance. Break the protocol using delegation and reuse the small, delegated objects.

Discussion

The “is-a” relation is a common software misconception based on implementation and thus, this subclassification has an incorrect motivation. You should only use inheritance when you find a “behaves-as-a” relation between two objects.

The following example shows the classical “is-a” problem:

```
public class Rectangle {

    int length;
    int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int area() {
        return this.length * this.width;
    }
}

public class Square extends Rectangle {

     public Square(int size) {
        super(size, size);
    }
    public int area() {
        return this.length * this.length;
    }
}

public class Box extends Rectangle {
}
```

Square and Box are not true Rectangles from a behavioral perspective. They violate the Liskov substitution principle (see Recipe 19.1, "Breaking Deep Inheritance"). You can refactor this by applying the recipe as follows:

```
abstract public class Shape {
    abstract public int area();
}

public final class Rectangle extends Shape {

    int length;
    int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int area() {
        return this.length * this.width;
    }
}

public final class Square extends Shape {
     // No longer subclassifies Rectangle

     int size;

     public Square(int size) {
        this.size = size;
    }

    public int area() {
        return this.size * this.size;
    }
}

public final class Box {
    // No longer subclassifies a Shape

    Square shape;

    public Box(int length, int width) {
        this.shape = new Rectangle(length, width);
    }

    public int area() {
        return shape.area();
    }
}
```

Inheritance is typically used to model an “is-a” relationship, where a subclass represents a more specialized version of its superclass. In this case, a `Square` is not a specialized version of a `Rectangle`. While a square is a type of rectangle in the real world, the relationship between them is more accurately described as a “has-a” relationship, where a `Square` has a `Rectangle` shape.

The issue arises when trying to represent a square using the `Rectangle` class hierarchy. A square is defined as a rectangle with equal sides, so changing the length of a `Rectangle` instance to represent a square contradicts the nature of a rectangle, where the length and width can have different values. Overriding can issue warnings when subclassing concrete methods. Deep hierarchies (more than three levels) are also a smell for bad subclassing. As an exception, if a hierarchy follows the principle “behaves like” then it is safe. In legacy systems it is very common to have deep hierarchies and method overriding; you need to refactor them and subclass only for essential reasons, not implementation ones.

Related Recipes

Recipe 19.2, “Breaking Yo-Yo Hierarchies”

19.4 Replacing “is-a” Relationship with Behavior

Problem

Schools often teach that inheritance represents an “is-a” relationship.

Solution

Think about protocol and behavior and forget accidental inheritance.

Discussion

“is-a” models do not follow the bijection principle, generating unexpected behavior, polluting the code with subclass overrides, and violating the Liskov substitution principle (see Recipe 19.1, “Breaking Deep Inheritance”).

You should not apply the bijection principle in this literal way. The subclass relationship sounds great when you read “is-a” aloud. But “behaves-as-a” must be the guide for bijection. You should always think in terms of “behaves-as-a” and prefer composition over inheritance. An “is-a” relation comes from the data world. Maybe you learned entity-relationship diagrams with structured design and data modeling, but now you need to think in terms of behavior. Behavior is essential, and data is accidental.

Entity-Relationship Diagrams (ERD)

Entity-relationship diagrams are a visual representation of the data in a database. In an ERD diagram, entities are represented by rectangles, while the relationships between entities are represented by lines connecting the rectangles.

Here is a classic example:

```java
class ComplexNumber {
    protected double realPart;
    protected double imaginaryPart;

    public ComplexNumber(double realPart, double imaginaryPart) {
        this.realPart = realPart;
        this.imaginaryPart = imaginaryPart;
    }
}

class RealNumber extends ComplexNumber {
    public RealNumber(double realPart) {
        super(realPart, 0);
   }

    public void setImaginaryPart(double imaginaryPart) {
        System.out.println("Cannot set imaginary part for a real number.");
    }
}
```

You can refactor it as follows:

```java
class Number {
    protected double value;

    public Number(double value) {
        this.value = value;
    }
}

class ComplexNumber extends Number {
    protected double imaginaryPart;

    public ComplexNumber(double realPart, double imaginaryPart) {
        super(realPart);
        this.imaginaryPart = imaginaryPart;
    }
}

class RealNumber extends Number {
}
```

Every real number “is-a” complex number (according to math). An integer “is-a” real number (according to math). A real number does not “behave-like-a” complex number. You cannot do `real.setImaginaryPart()`, so it is not complex according to the bijection.

Related Recipes

Recipe 19.3, “Breaking Subclassification for Code Reuse”

Recipe 19.6, “Renaming Isolated Classes”

Recipe 19.11, “Removing Protected Attributes”

See Also

“Circle–Ellipse Problem” on Wikipedia

19.5 Removing Nested Classes

Problem

You have nested or pseudo-private classes hiding implementation details.

Solution

Don’t use nested classes. They don’t exist in the real world.

Discussion

Nested classes break the bijection (as defined in Chapter 2) because they don’t map to real-world concepts. They are difficult to test and reuse, and their hidden scope brings namespace complexity (see Recipe 18.4, “Removing Global Classes”). You can make the class public and keep the new class under your own namespace/module or use a façade (see Recipe 17.3, “Breaking God Objects”) to expose what is important and hide what is irrelevant. Some languages allow you to create private concepts that can be only used internally, but these are harder to test, debug, and reuse.

Here’s an example of a nested class:

```
class Address {
  String description = "Address: ";
```

```
    public class City {
      String name = "Doha";
    }
  }

  public class Main {
    public static void main(String[] args) {
      Address homeAddress = new Address();
      Address.City homeCity = homeAddress.new City();
      System.out.println(homeAddress.description + homeCity.name);
    }
  }

  // The output is "Address: Doha"
  //
  // If you change privacy to 'private class City'
  //
  // You get an error " Address.City has private access in Address"
```

Here's what it looks like after you promote it:

```
  class Address {
    String description = "Address: ";
  }

  class City {
    String name = "Doha";
  }

  public class Main {
    public static void main(String[] args) {
      Address homeAddress = new Address();
      City homeCity = new City();
      System.out.println(homeAddress.description + homeCity.name);
    }
  }

  // The output is "Address: Doha"
  //
  // Now you can reuse and test the City concept
```

Many languages are bloated with complex features. You seldom need these new fancy features. You need to maintain a minimal set of concepts to avoid accidental complexity and deal with the essential ones.

See Also

"Java Inner Classes" on W3Schools

19.6 Renaming Isolated Classes

Problem

Your classes are global but you have abbreviations in their names.

Solution

Don't use abbreviations in subclasses. If your classes are global, use fully qualified names.

Discussion

Abbreviations hurt readability and favor mistakes. You need to rename your classes to provide context and use modules, namespaces, or fully qualified names. Here is an example of abbreviated classes for the Perseverance Mars Rover:

```
abstract class PerserveranceDirection {
}

class North extends PerserveranceDirection {}
class East extends PerserveranceDirection {}
class West extends PerserveranceDirection {}
class South extends PerserveranceDirection {}

// Subclasses have short names and are meaningless outside the hierarchy
// If you reference East, you might mistake it for the cardinal point
```

Here's what it looks like with a full context:

```
abstract class PerserveranceDirection { }

class PerserveranceDirectionNorth extends PerserveranceDirection {}
class PerserveranceDirectionEast extends PerserveranceDirection {}
class PerserveranceDirectionWest extends PerserveranceDirection {}
class PerserveranceDirectionSouth extends PerserveranceDirection {}

// Subclasses have fully qualified names
```

Automatic detection is not an easy task. You could enforce local naming policies for subclasses, and you need to choose your names wisely. If your language supports it, use modules, namespaces (see Recipe 18.4, "Removing Global Classes"), and local scopes.

Some languages provide namespaces or modules where you can use short names under a certain scope to avoid collisions.

Related Recipes

Recipe 19.3, "Breaking Subclassification for Code Reuse"

19.7 Making Concrete Classes Final

Problem

You have concrete classes with subclasses.

Solution

Make your concrete classes final. Move your hierarchy.

Discussion

Concrete classes are bad parents and violate the Liskov substitution principle (see Recipe 19.1, "Breaking Deep Inheritance"). Overriding methods to a concrete class is always a mistake since subclasses should be specializations. You need to refactor the hierarchies and favor composition. Leaf classes should be concrete and nonleaf classes should be abstract.

Here's a Stack example:

```
class Stack extends ArrayList {
    public void push(Object value) { ... }
    public Object pop() { ... }
}

// Stack does not behave like an ArrayList
// Besides pop, push, top it also implements (or overrides)
// get, set, add, remove, and clear
// Stack elements can be arbitrary accessed

// Both classes are concrete
```

Both can inherit from the Collection class:

```
abstract class Collection {
    public abstract int size();
}

final class Stack extends Collection {
    private Object[] contents;

    public Stack(int maxSize) {
      contents = new Object[maxSize];
    }
    public void push(Object value) { ... }
    public Object pop() { ... }
    public int size() {
        return contents.length;
    }
}

final class ArrayList extends Collection {
    private Object[] contents;

    public ArrayList(Object[] contents) {
        this.contents = contents;
    }
    public int size() {
        return contents.length;
    }
}
```

Overriding a concrete method is a clear smell. You can enforce this policy on most linters (see Recipe 5.2, "Declaring Variables to Be Variable"). Abstract classes should have just a few concrete methods. You can check against a predefined threshold for offenders. Accidental subclassification is the first obvious and attractive option for junior developers. More mature developers find composition opportunities instead. Composition is dynamic, multiple, pluggable, more testable, more maintainable, and less coupled than inheritance. Only subclassify an entity if it follows the "behaves-as-a" relationship (see Recipe 19.4, "Replacing "is-a" Relationship with Behavior"). After subclassing, the parent class should be abstract.

Related Recipes

Recipe 19.3, "Breaking Subclassification for Code Reuse"

See Also

"Composition over Inheritance" on Wikipedia

19.8 Defining Class Inheritance Explicitly

Problem

Your classes are abstract, final, or undefined but you don't mark them explicitly.

Solution

If your language has the right tool, your classes should be either abstract or final and then the compiler can enforce these business rules for you.

Discussion

Subclassification for code reuse presents a lot of problems. You need to declare all your leaf classes as *final* and the rest of them as *abstract*. These keywords also help to make your designs explicit. Managing hierarchies and composition is the main task of a good software designer, and keeping hierarchies healthy is crucial to favor cohesion and avoid coupling.

All these classes lack an explicit final declaration:

```
public class Vehicle
{
  // Class is not a leaf. Therefore it should be abstract

  // An abstract method that only declares, but does not define the start
  // functionality because each vehicle uses a different starting mechanism
  abstract void start();
}

public class Car extends Vehicle
{
  // Class is a leaf. Therefore it should be final
}

public class Motorcycle extends Vehicle
{
  // Class is a leaf. Therefore it should be final
}
```

You can detect hierarchy problems by enforcing the notation:

```
abstract public class Vehicle
{
  // The class is not a leaf. Therefore it must be abstract

  // An abstract method that only declares, but does not define the start
  // functionality because each vehicle uses a different starting mechanism
  abstract void start();
}

final public class Car extends Vehicle
{
  // The class is a leaf. Therefore it is final
}

final public class Motorcycle extends Vehicle
{
  // The class is a leaf. Therefore it is final
}
```

Look back at your classes and start qualifying them either as abstract or final. There are no valid cases for two concrete classes where one subclassifies the other.

Related Recipes

Recipe 12.3, “Refactoring Classes with One Subclass”

Recipe 19.2, “Breaking Yo-Yo Hierarchies”

Recipe 19.3, “Breaking Subclassification for Code Reuse”

Recipe 19.11, “Removing Protected Attributes”

See Also

“Designing Reusable Classes” by Ralph E. Johnson and Brian Foote

19.9 Migrating Empty Classes

Problem

You have classes without behavior. But classes are used to encapsulate behavior.

Solution

Remove all empty classes.

Discussion

Empty classes violate the bijection since there are no objects without behavior in the real world. Notable examples are unnecessary exceptions or middle hierarchy classes. They pollute namespaces. You need to remove these classes and replace them with objects instead. Many developers still think that classes are data repositories and they confuse *different behavior* concepts with *returning different data*.

Here you have an empty `ShopItem` class:

```
class ShopItem {
  code() { }
  description() { }
}

class BookItem extends ShopItem {
   code() { return 'book' }
   description() { return 'some book'}
}

// Concrete class has no real behavior, just returns different 'data'
```

Here’s what it looks like when you refactor it:

```
class ShopItem {
  constructor(code, description) {
    // validate code and description
    this._code = code;
    this._description = description;
  }
  code() { return this._code }
  description() { return this._description }
  // Add more functions to avoid anemic classes
  // Getters are also code smells, so you need to iterate more
}

bookItem = new ShopItem('book', 'some book');
// create more items
```

Several linters warn you about empty classes. You can also make your own scripts using metaprogramming (see Chapter 23, "Metaprogramming"). Classes are what they do, their behavior, and empty classes do nothing.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 3.6, "Removing DTOs"

Recipe 12.3, "Refactoring Classes with One Subclass"

Recipe 18.4, "Removing Global Classes"

Recipe 22.2, "Removing Unnecessary Exceptions"

19.10 Delaying Premature Classification

Problem

You have abstractions before having seen enough concrete connections.

Solution

Don't guess what the future will bring you.

Discussion

It's difficult to make predictions, especially about the future. This is a common problem in the software industry. Keep in mind that wrong first impressions lead to bad designs. You need to wait for concrete examples to generalize and refactor only when you have enough evidence. Aristotelian classification is a big problem in computer science. Software developers tend to classify and name things *before* gathering enough knowledge and context. You tend to classify objects before fully understanding their behavior, characteristics, requirements, or relationships.

See the following Song example:

```
class Song {
  constructor(title, artist) {
    this.title = title;
    this.artist = artist;
  }

  play() {
    console.log(`Playing ${this.title} by ${this.artist}`);
  }
}
```

When you discover classical songs you might be tempted to subclassify the Songs:

```
class ClassicalSong extends Song {
  constructor(title, artist, composer) {
    super(title, artist);
    this.composer = composer;
  }

  listenCarefully() {
    console.log(`I am listening to ${this.title} by ${this.composer}`);
  }
}

const goldberg = new ClassicalSong
    ("The Goldberg Variations", "Glenn Gould", "Bach");
```

Now you can also have pop songs:

```
class PopSong extends Song {
  constructor(title, artist, album) {
    super(title, artist);
    this.album = album;
  }

  danceWhileListening() {
    console.log(`I am dancing with ${this.title}`);
  }
}

const theTourist = new PopSong("The Tourist", "Radiohead", "OK, Computer");
```

You are speculating about the future of all genres. What happens when you mix them?

```
class ClassicalPopSong extends ClassicalSong {
  constructor(title, artist, composer, album) {
    super(title, artist, composer);
    this.album = album;
  }

  danceWhileListening() {
    console.log(`${this.title} is a classical song with a pop twist`);
  }
}

const classicalPopSong = new ClassicalPopSong(
    "Popcorn Concerto", "Classical Pop Star", "Beethoven);
```

An abstract class with just one subclass indicates premature classification. When working with classes, name abstractions as soon as they appear. Choose good names based on behavior; don't name your abstractions until you name your concrete subclasses.

Related Recipes

Recipe 19.3, "Breaking Subclassification for Code Reuse"

19.11 Removing Protected Attributes

Problem

You have protected attributes in your classes.

Solution

Make the attributes private.

Discussion

Protected Attributes

A *protected attribute* is an instance variable or property of a class that can only be accessed within the class or its subclasses. Protected attributes are a way to restrict access to certain data within a class hierarchy, while still allowing subclasses to access and modify that data if necessary.

Protected attributes are great for encapsulating and controlling access to the properties. But frequently they are a warning of another problem. Protected attributes are a hint to detect subclassification for code reuse purposes and violation of the Liskov substitution principle (see Recipe 19.1, "Breaking Deep Inheritance"). As with many other recipes, favor composition, don't subclassify attributes, and extract behavior to separate objects. Or use traits if available in your language of choice.

Traits

Traits define a set of common characteristics or behaviors that can be shared by multiple classes. A trait is essentially a set of methods that can be reused by different classes without requiring them to inherit from a common superclass. It provides a mechanism for code reuse that is more flexible than inheritance since it allows classes to inherit behavior from multiple sources.

Here is an example of protected attributes:

```
abstract class ElectronicDevice {
    protected $battery;

    public function __construct(Battery $battery) {
        $this->battery = $battery; // battery is inherited to all devices
    }
}

abstract class IDevice extends ElectronicDevice {
    protected $operatingSystem; // operating system is inherited to all devices

    public function __construct(Battery $battery, OperatingSystem $ios) {
        $this->operatingSystem = $ios;
        parent::__construct($battery)
  }
}

final class IPad extends IDevice {
    public function __construct(Battery $battery, OperatingSystem $ios) {
        parent::__construct($battery, $ios)
  }
}

final class IPhone extends IDevice {
    private $phoneModule:

    public function __construct(Battery $battery,
                                OperatingSystem $ios,
                                PhoneModule $phoneModule) {
        $this->phoneModule = $phoneModule;
        parent::__construct($battery, $ios);
    }
}
```

Here's what it looks like when you refactor them:

```
interface ElectronicDevice { }

interface PhoneCommunication { }

final class IPad implements ElectronicDevice {
    private $operatingSystem; // The attributes are duplicated
    private $battery;
    // If you have too much duplicated behavior you should extract them

    public function __construct(Battery $battery, OperatingSystem $ios) {
        $this->operatingSystem = $ios;
        $this->battery = $battery;
    }
}

final class IPhone implements ElectronicDevice, PhoneCommunication {
    private $phoneModule;
    private $operatingSystem;
    private $battery;

    public function __construct(Battery $battery,
                                OperatingSystem $ios,
                                PhoneModule $phoneModule) {
        $this->phoneModule = $phoneModule;
        $this->operatingSystem = $ios;
        $this->battery = $battery;
    }
}
```

In languages supporting protected attributes you can avoid them by policy or have a warning of the problem. Protected attributes are yet another tool you should use carefully. Every occurrence is a potential problem, and you should be very picky with attributes and inheritance.

Related Recipes

Recipe 19.3, "Breaking Subclassification for Code Reuse"

19.12 Completing Empty Implementations

Problem

You have empty methods in the hierarchy for future implementation that do not fail.

Solution

Complete the methods with an exception or a possible solution.

Discussion

Empty methods violate the fail fast principle by providing a working (but potentially wrong) solution. You should throw an error indicating implementation is not complete. Creating an empty implementation might seem fine and you can jump to more interesting problems. But the code left won't fail fast, so debugging it will be a bigger problem.

Here's an example of an empty implementation:

```
class MerchantProcessor {
  processPayment(amount) {
    // no default implementation
  }
}
class MockMerchantProcessor extends MerchantProcessor {
  processPayment(amount) {
     // Empty implementation to comply with the compiler
     // Won't do anything
  }
}
```

This is more declarative and will fail fast:

```
class MerchantProcessor {
  processPayment(amount) {
    throw new Error('Should be overridden');
  }
}
class MockMerchantProcessor extends MerchantProcessor {
  processPayment(amount) {
     throw new Error('Will be implemented when needed');
  }
}
```

You can also replace it with a real implementation:

```
class MockMerchantProcessor extends MerchantProcessor {
  processPayment(amount) {
    console.log('Mock payment processed: $${amount}');
  }
}
```

Since empty code is valid sometimes, only a good peer review will find these problems.

Being lazy and deferring certain decisions is acceptable, but it's crucial to be explicit about it.

Related Recipes

Recipe 20.4, "Replacing Mocks with Real Objects"

Recipe 19.9, "Migrating Empty Classes"

CHAPTER 20
Testing

No amount of testing can prove a software right, a single test can prove a software wrong.
—Amir Ghahrai

20.0 Introduction

Working without automated test coverage in prior decades was challenging, as developers had to rely heavily on manual testing and debugging to catch and fix issues in their software. Manual testing involves running software through a series of tests designed to check its functionality, performance, and stability. This process is time consuming and prone to human error, as testers may miss certain scenarios or overlook critical defects.

Without automated tests, developers had to spend a considerable amount of time debugging and fixing issues, which could slow down the development process and delay the release of new features or updates. It was also difficult to ensure consistent and reliable results across different platforms and environments. Developers had to manually test their software on various operating systems, browsers, and hardware configurations, which could result in unexpected defects and compatibility issues. There was no guarantee that resolving an issue or developing a new feature would prevent known scenarios from breaking in the future, and end users were accustomed to experiencing unexpected failures in previously functioning features.

Today, writing tests is a mandatory discipline for good developers. You can change the software anytime you want if you have written tests for the previous functionality. There are many books and courses on how to develop good code, but not so many on how to write good tests. Hopefully, you will be able to apply this chapter's recipes.

20.1 Testing Private Methods

Problem

You need to test private methods.

Solution

Don't test your private methods. Extract them.

Discussion

Every developer has faced the challenge to write a test for an internal function or method that provides important support for higher-level functionalities. You can't directly test the methods because you'd break the method's encapsulation and you also don't want to copy it or make it public. As a rule, do not make your methods public for testing or use metaprogramming to circumvent the protection (see Chapter 23, "Metaprogramming"). If your method is trivial, you don't need to test it. If your method is complicated, you need to convert it into a method object (see Recipe 10.7, "Extracting a Method to an Object"). Do not move the private computation to helpers (see Recipe 7.2, "Renaming and Breaking Helpers and Utils") or use static methods (see Recipe 18.2, "Reifying Static Functions").

This example tests the time it takes light to travel from a distant star:

```
final class Star {

  private $distanceInParsecs;

  public function timeForLightReachingUs() {
    return $this->convertDistanceInParsecsToLightYears($this->distanceInParsecs);
  }

  private function convertDistanceInParsecsToLightYears($distanceInParsecs) {
      return 3.26 * $distanceInParsecs;
      // Function is using an argument that is already available
      // since it has private access to $distanceInParsecs.
      // This is another smell indicator.

      // You cannot test this function directly since it is private.
  }
}
```

Here's what it looks like when you create a converter:

```
final class Star {

  private $distanceInParsecs;
  public function timeToReachLightToUs() {
      return (new ParsecsToLightYearsConverter())
        ->convert($this->distanceInParsecs);
  }
}

final class ParsecsToLightYearsConverter {
  public function convert($distanceInParsecs) {
      return 3.26 * $distanceInParsecs;
  }
}

final class ParsecsToLightYearsConverterTest extends TestCase {
  public function testConvert0ParsecsReturns0LightYears() {
    $this->assertEquals(0, (new ParsecsToLightYearsConverter())->convert(0));
  }
    // You can add lots of tests and rely on this object
    // So you don't need to test Star conversions
    // You can't yet test Star public timeToReachLightToUs()
    // This is a simplified scenario
}
```

You can only find metaprogramming abuse on some unit frameworks (see Chapter 23, "Metaprogramming"). With this guide, you should always choose the *method object* solution.

Related Recipes

Recipe 7.2, "Renaming and Breaking Helpers and Utils"

Recipe 10.7, "Extracting a Method to an Object"

Recipe 18.2, "Reifying Static Functions"

Recipe 23.1, "Removing Metaprogramming Usage"

Recipe 23.2, "Reifying Anonymous Functions"

See Also

Should I Test Private Methods website

20.2 Adding Descriptions to Assertions

Problem

You have a lot of good assertions and use the default description to indicate the failure while omitting the reason.

Solution

Use assertions with declarative and meaningful descriptions.

Discussion

When an assertion fails, you need to understand quickly why it has failed. Adding informative optional descriptions is an amazing strategy to avoid wasting time. You can also add some guides for problem-solving. This is the perfect replacement for comments in the code. Descriptions for asserts are the place where you explain why you expect a particular result, and that in turn allows you to make explicit design or implementation decisions.

This example compares two collections:

```
public function testNoNewStarsAppeared()
  {
     $expectedStars = $this->historicStarsOnFrame();
     $observedStars = $this->starsFromObservation();
     // These sentences get a very large collection

     $this->assertEquals($expectedStars, $observedStars);
     // If something fails you will have a very hard time debugging
    }
```

Here's what it looks like when you add the description on the assert:

```
public function testNoNewStarsAppeared(): void
  {
     $expectedStars = $this->historicStarsOnFrame();
     $observedStars = $this->starsFromObservation();
     // These sentences get a very large collection

     $newStars = array_diff($expectedStars, $observedStars);

     $this->assertEquals($expectedStars, $observedStars,
         'There are new stars ' . print_r($newStars, true));
     // Now you can see EXACTLY why the assertion failed with a clear and
     // declarative message
    }
```

Since `assert` (or `assertTrue`, `assert.isTrue`, `Assert.True`, `assert_true`, `XCTAssertTrue`, `ASSERT_TRUE`), `assertDescription (expect(true).toBe(false, 'message'))`, `assertEquals("message", true, false)`, and `ASSERT_EQ((true, false) << "message";)`, etc. are sometimes different functions with a different number of arguments, you can adjust the policies to favor the version with the helpful message. Be respectful to the reader of your assertions especially since it might even be yourself!

See Also

xUnit: assert description deprecation

20.3 Migrating assertTrue to Specific Assertions

Problem

You have assertions with booleans on your tests.

Solution

Don't use `assertTrue()` unless you are checking a boolean.

Discussion

Asserting against booleans makes error tracking more difficult. Every boolean assertion is an opportunity to make a more specific assertion. You should check if you can rewrite the boolean condition better and favor `assertEquals`. When asserting against a boolean your test engines cannot help you very much. They just tell you something failed. Error tracking gets more difficult.

This is an assertion on a boolean equality condition:

```
final class RangeUnitTest extends TestCase {

  function testValidOffset() {
    $range = new Range(1, 1);
    $offset = $range->offset();
    $this->assertTrue(10 == $offset);
    // No functional essential description :(
    // Accidental description provided by tests is very bad
  }
}
```

When failing, the unit framework will show you:

```
1 Test, 1 failed
Failing asserting true matches expected false :(
() <-- no business description :(

<Click to see difference> - Two booleans
(and a diff comparator will show you two booleans)
```

This is a more descriptive assertion:

```
final class RangeUnitTest extends TestCase {

  function testValidOffset() {
    $range = new Range(1, 1);
```

```
        $offset = $range->offset();
        $this->assertEquals(10, $offset, 'All pages must have 10 as offset');
        // Expected value should always be first argument
        // You add a functional essential description
        // to complement accidental description provided by tests
      }
    }
```

When failing, the unit framework will show you:

```
1 Test, 1 failed
Failing asserting 0 matches expected 10
All pages must have 10 as offset <-- business description

<Click to see difference>
(and a diff comparator will help you and it will be a great help
for complex objects like objects or jsons)
```

This code improvement has no benefit for computing since both expressions are equivalent. But more specific assertion checks greatly improve software maintenance and team collaboration. Try to rewrite your boolean assertions and you will fix failures much faster.

Related Recipes

Recipe 14.3, "Reifying Boolean Variables"

Recipe 14.12, "Changing Comparison Against Booleans"

20.4 Replacing Mocks with Real Objects

Problem

Your tests use mock objects instead of real ones.

Solution

Replace the mocks with real objects whenever it is possible.

Discussion

Mock Objects

A *mock object* mimics the behavior of a real object to test or simulate its behavior. You can use it to test software components that have dependencies on other components, such as external APIs or libraries.

Mocking is a great aid when testing behavior. But as with many other tools, you can abuse them. Mocks add accidental complexity, are harder to maintain, and give you a false sense of safety. You will find yourself building a parallel solution with real objects and mocks, making maintainability more difficult. As a rule, you should mock only nonbusiness entities.

The following example uses a mock on a business object:

```php
class PaymentTest extends TestCase
{
    public function testProcessPaymentReturnsTrueOnSuccessfulPayment()
    {
        $paymentDetails = array(
            'amount'   => 123.99,
            'card_num' => '4111-1111-1111-1111',
            'exp_date' => '03/2013',
        );

        $payment = $this->getMockBuilder('Payment')
            ->setConstructorArgs(array())
            ->getMock();
        // You should not mock a business object!

        $authorizeNet = new AuthorizeNetAIM(
            $payment::API_ID, $payment::TRANS_KEY);
        // This is an external and coupled system.
        // You have no control over it so tests become fragile

        $paymentProcessResult = $payment->processPayment(
            $authorizeNet, $paymentDetails);

        $this->assertTrue($paymentProcessResult);
    }
}
```

You can replace the business mock with real objects and mock external dependencies:

```php
class PaymentTest extends TestCase
{
    public function testProcessPaymentReturnsTrueOnSuccessfulPayment()
    {
        $paymentDetails = array(
            'amount'   => 123.99,
            'card_num' => '4111-1111-1111-1111',
            'exp_date' => '03/2013',
        );

        $payment = new Payment(); // Payment is a real one

        $response = new \stdClass();
        $response->approved = true;
        $response->transaction_id = 123;
```

```
            $authorizeNet = $this->getMockBuilder('\AuthorizeNetAIM')
                ->setConstructorArgs(array($payment::API_ID, $payment::TRANS_KEY))
                ->getMock();

            // External system is mocked

            $authorizeNet->expects($this->once())
                ->method('authorizeAndCapture')
                ->will($this->returnValue($response));

            $paymentProcessResult = $payment->processPayment(
                $authorizeNet, $paymentDetails);

            $this->assertTrue($paymentProcessResult);
        }
    }
```

This is an architectural pattern. It will not be easy to create an automatic detection rule. You will find mocking accidental problems (serialization, databases, APIs) is a very good practice to avoid coupling. Mocks, like many other test doubles, are excellent tools.

Related Recipes

Recipe 19.12, "Completing Empty Implementations"

20.5 Refining Generic Assertions

Problem

You have tests with assertions that are too generic.

Solution

Test assertions should be precise; they shouldn't be too vague or specific.

Discussion

Don't make weak tests to create a false sensation of coverage. Generic assertions give you a false sense of security. You should check the right case, assert for a functional case, and avoid testing implementation. This is a vague test:

```
square = Square(5)

assert square.area() != 0
# This will lead to false negatives since it does not cover all cases
```

This test is more precise:

```
square = Square(5)

assert square.area() = 25
# Assertion should be precise
```

With mutation testing (see Recipe 5.1, "Changing var to const") techniques you can find these errors on your tests. You should use development techniques like test-driven development (TDD) (see Recipe 4.8, "Removing Unnecessary Properties") that request concrete business cases and make concrete assertions based on your domain.

Related Recipes

Recipe 20.4, "Replacing Mocks with Real Objects"

Recipe 20.6, "Removing Flaky Tests"

20.6 Removing Flaky Tests

Problem

You have tests that are not deterministic.

Solution

Don't depend on things your test cannot control, like external databases or resources on the internet. If your tests fail randomly, you need to fix them.

Discussion

If your tests are not deterministic, you start to lose confidence, which lowers your morale. You might feel like you are wasting time adding or running tests. Tests should be in full environmental control (see Recipe 18.5, "Changing Global Date Creation"). There should be no space for erratic behavior and degrees of freedom. You need to remove all test coupling. Fragile, intermittent, sporadic, or erratic tests are common in many organizations. Nevertheless, they mine developers' trust.

Flaky tests are tests that are overly sensitive to changes in the environment or system being tested. For example, a test may fail because of a change in the underlying hardware, network connectivity, or software dependencies. Flaky tests can be problematic because they require frequent maintenance and may not accurately reflect the true functionality of the system.

Flaky Tests

Flaky or *erratic tests* produce inconsistent or unpredictable results. These types of tests can be difficult to work with because they may pass or fail unexpectedly, making it challenging to determine whether the code being tested is working correctly.

Here is an example of an erratic test:

```
public abstract class SetTest {

    protected abstract Set<String> constructor();

    @Test
    public final void testAddEmpty() {
        Set<String> colors = this.constructor();
        colors.add("green");
        colors.add("blue");
        assertEquals("{green. blue}", colors.toString());
       // This is fragile since it depends on the set order
       // and mathematical sets are by definition not sorted
    }
}
```

This is more deterministic:

```
public abstract class SetTest {

    protected abstract Set<String> constructor();

    @Test
    public final void testAddEmpty() {
        Set<String> colors = this.constructor();
        colors.add("green");
        assertEquals("{green}", colors.toString());
    }

    @Test
    public final void testEntryAtSingleEntry() {
        Set<String> colors = this.createFromArgs("red");
        Boolean redIsPresent = colors.contains("red");
        assertEquals(true, redIsPresent);
    }
}
```

Detection of erratic tests can be done with test run statistics. It is very hard to put some tests in maintenance because you are removing a safety net. Fragile tests show system coupling and nondeterministic or erratic behavior. Developers spend lots of time and effort fighting against these false positives.

Related Recipes

Recipe 20.5, “Refining Generic Assertions”

Recipe 20.12, “Rewriting Tests Depending on Dates”

20.7 Changing Float Number Assertions

Problem

You have assertions with float numbers.

Solution

Don’t compare float numbers.

Discussion

Asserting two float numbers are the same is a very difficult problem. When comparing two float numbers in testing, there can be several issues that arise due to the way floating-point numbers are represented and stored in computer memory. These issues can lead to unexpected test results and can make it difficult to write reliable and accurate tests.

Float numbers can be subject to rounding errors. Even if two calculations should produce the same value, they may end up with slightly different results due to rounding errors, leading to false negatives or false positives in your tests. This can lead to fragile tests. As a rule, you should avoid float numbers unless you have real performance concerns as this is a case of premature optimization (see Chapter 16, “Premature Optimization”). You can use arbitrary precision numbers and, if you need to compare floats, compare with tolerance. Comparing float numbers is an old computer science problem. The usual solution is to use threshold comparisons.

This example compares two float numbers:

```
Assert.assertEquals(0.0012f, 0.0012f); // Deprecated
Assert.assertTrue(0.0012f == 0.0012f); // Not JUnit - Smell
```

This is the recommended way to compare two float numbers:

```
float LargeThreshold = 0.0002f;
float SmallThreshold = 0.0001f;
Assert.assertEquals(0.0012f, 0.0014f, LargeThreshold); // true
Assert.assertEquals(0.0012f, 0.0014f, SmallThreshold); // false - Assertion Fail

Assert.assertEquals(12 / 10000, 12 / 10000); // true
Assert.assertEquals(12 / 10000, 14 / 10000); // false
```

You can add a check on `assertEquals()` on your testing frameworks to avoid checking for floats. You should always avoid comparing floats.

Related Recipes

Recipe 24.3, "Changing Float Numbers to Decimals"

20.8 Changing Test Data to Realistic Data

Problem

You are using fake data on your tests.

Solution

Use real case scenarios and real data if possible.

Discussion

Fake data is a bijection violation as defined in Chapter 2; it leads to bad test use cases and damages readability. You should use real data and use the MAPPER to map real entities and real data. In the past, developers used to fake domain data, and tested with abstract data. They developed using a waterfall model, far away from real users. User acceptance testing became more important with bijection and MAPPER techniques, domain-driven design, and TDD.

Domain-Driven Design

Domain-driven design focuses on aligning the design of software systems with the business or problem domain, making the code more expressive, maintainable, and closely tied to business requirements.

Using Agile methodologies, you need to test with real-world data. If you find an error in a production system, add a case covering the exact mistake with real data.

User Acceptance Testing

User acceptance testing (UAT) checks whether a software system or application meets business and user requirements and is ready for deployment to production. It involves a series of tests with real data and reviews by end users to verify that the software is functioning correctly and meets their needs and expectations.

The following test uses unrealistic data:

```
class BookCartTestCase(unittest.TestCase):
    def setUp(self):
        self.cart = Cart()

    def test_add_book(self):
       self.cart.add_item('xxxxx', 3, 10)
        # This is not a real example

       self.assertEqual(
           self.cart.total,
           30,
           msg='Book Cart total not correct after adding books')
       self.assertEqual(
           self.cart.items['xxxxx'],
           3,
           msg='Quantity of items not correct after adding book')

    def test_remove_item(self):
        self.cart.add_item('fgdfhhfhhh', 3, 10)
        self.cart.remove_item('fgdfhhfrhh', 2, 10)
        # You made a typo since example is not a real one
        self.assertEqual(
            self.cart.total,
            10,
            msg='Book Cart total not correct after removing book')
        self.assertEqual(
            self.cart.items['fgdfhhfhhh'],
            1,
            msg='Quantity of books not correct after removing book')
```

You can avoid the typo in the example by using Recipe 6.8, "Replacing Magic Numbers with Constants". But you would not replace ALL the text in your use cases. Here's what the test looks like with real data instead:

```
class BookCartTestCase(unittest.TestCase):
    def setUp(self):
        self.cart = Cart()

    def test_add_book(self):
       self.cart.add_item('Harry Potter', 3, 10)

       self.assertEqual(
           self.cart.total,
           30,
           msg='Book Cart total not correct after adding books')
       self.assertEqual(
           self.cart.items['Harry Potter'],
           3,
           msg='Quantity of items not correct after adding book')

    # You don't reuse the same example.
    # You use a new REAL book.
    def test_remove_item(self):
        self.cart.add_item('Divergent', 3, 10)
        self.cart.remove_item('Divergent', 2, 10)
        self.assertEqual(
            self.cart.total,
            10,
            msg='Book Cart total not correct after removing book')
        self.assertEqual(self.cart.items[
            'Divergent'],
            1,
            msg='Quantity of books not correct after removing book')
```

You can still make typos on real-world examples (like "Devergent") but you will find them sooner. Reading tests is the only way to learn how the software behaves, and you need to be overly explicit on your tests.

In some domains and under certain regulations you cannot use real data. In these cases, you should fake it with meaningful but anonymized data.

Related Recipes

Recipe 8.5, "Converting Comments to Function Names"

See Also

"Given-When-Then" on Wikipedia

20.9 Protecting Tests Violating Encapsulation

Problem

You have tests that violate encapsulation.

Solution

Don't write methods with the only purpose of using them in your tests.

Discussion

Sometimes you write code to favor tests and that code violates encapsulation, leading to bad interfaces and bringing unnecessary coupling. The tests must be in full environmental control, and if you cannot control your object, you have undesired coupling. Decouple them.

Here you can see a method to test:

```
class Hangman {
    private $wordToGuess;

    function __construct() {
        $this->wordToGuess = getRandomWord();
        // Test is not in control of this
    }

    public function getWordToGuess(): string {
        return $this->wordToGuess;
        // Sadly you need to reveal this
    }
}

class HangmanTest extends TestCase {
    function test01WordIsGuessed() {
        $hangmanGame = new Hangman();
        $this->assertEquals('tests', $hangmanGame->wordToGuess());
        // How can you make sure the word is guessed?
    }
}
```

This is a better approach:

```
class Hangman {
    private $wordToGuess;

    function __construct(WordRandomizer $wordRandomizer) {
        $this->wordToGuess = $wordRandomizer->newRandomWord();
    }
    function wordWasGuessed() { }
    function play(char letter) { }
}

class MockRandomizer implements WordRandomizer {
    function newRandomWord(): string {
        return 'tests';
    }
}

class HangmanTest extends TestCase {
    function test01WordIsGuessed() {
        $hangmanGame = new Hangman(new MockRandomizer());
        // You are in full control!
        $this->assertFalse($hangmanGame->wordWasGuessed());
        $hangmanGame->play('t');
        $this->assertFalse($hangmanGame->wordWasGuessed());
        $hangmanGame->play('e');
        $this->assertFalse($hangmanGame->wordWasGuessed());
        $hangmanGame->play('s');
        $this->assertTrue($hangmanGame->wordWasGuessed());
```

```
            // You just test behavior
        }
    }
```

This is a design smell. You can detect if you need a method just for testing. Open-box tests are fragile. They test implementation instead of behavior.

Related Recipes

Recipe 3.3, "Removing Setters from Objects"

Recipe 20.6, "Removing Flaky Tests"

See Also

xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros

20.10 Removing Irrelevant Test Information

Problem

You have tests with irrelevant data.

Solution

Don't add unnecessary information to your assertions.

Discussion

Irrelevant data distracts the reader's attention and hinders readability and maintainability. You should remove them as much as possible and leave only the needed assertions. Tests should be minimal and follow the setup/exercise/assert pattern.

Here you can see irrelevant data related to car models and colors:

```
def test_formula_1_race():
    # Setup
    racers = [
        {"name": "Lewis Hamilton",
        "team": "Mercedes",
        "starting_position": 1,
        "car_color": "Silver"},
        {"name": "Max Verstappen",
        "team": "Red Bull",
        "starting_position": 2,
        "car_color": "Red Bull"},
        {"name": "Sergio Perez",
        "team": "Red Bull",
        "starting_position": 3,
        "car_color": "Red Bull"},
        {"name": "Lando Norris",
        "team": "McLaren",
        "starting_position": 4,
        "car_color": "Papaya Orange"},
        {"name": "Valtteri Bottas",
        "team": "Mercedes",
        "starting_position": 5,
        "car_color": "Silver"}
]

    # Exercise
    winner = simulate_formula_1_race(racers)

    # Test
    assert winner == "Lewis Hamilton"

    # This is all irrelevant to winner asserting
    assert racers[0]["car_color"] == "Silver"
    assert racers[1]["car_color"] == "Red Bull"
    assert racers[2]["car_color"] == "Red Bull"
    assert racers[3]["car_color"] == "Papaya Orange"
    assert racers[4]["car_color"] == "Silver"
    assert racers[0]["car_model"] == "W12"
    assert racers[1]["car_model"] == "RB16B"
    assert racers[2]["car_model"] == "RB16B"
    assert racers[3]["car_model"] == "MCL35M"
    assert racers[4]["car_model"] == "W12"
```

The following example includes only relevant information for testing purposes:

```
def test_formula_1_race():
    # Setup
    racers = [
        {"name": "Lewis Hamilton", "starting_position": 1},
        {"name": "Max Verstappen", "starting_position": 2},
        {"name": "Sergio Perez", "starting_position": 3},
        {"name": "Lando Norris", "starting_position": 4},
        {"name": "Valtteri Bottas" "starting_position": 5},
    ]

    # Exercise
    winner = simulate_formula_1_race(racers)

    # Test
    assert winner == "Lewis Hamilton"
```

You can find some patterns in unneeded assertions. The tests should be prose. Always focus on the reader. It might be you, a couple of months from now.

Related Recipes

Recipe 20.5, "Refining Generic Assertions"

20.11 Adding Coverage for Every Merge Request

Problem

You have merge requests without including coverage.

Solution

Make sure you cover every code change with corresponding tests.

Discussion

Merge requests without test coverage lower overall system quality and damage maintainability. When you need to make a change, update the live specification of your code. Instead of generating dead documentation of what your code does, you should write a covering use scenario. If you change some code that has no tests, you need to add coverage. Suppose you change code with existing coverage. You are lucky! You can go and change your broken tests.

Here's a functional change without coverage:

```
export function sayHello(name: string): string {
  const lengthOfName = name.length;
-  const salutation =
-  `How are you ${name}?, I see your name has ${lengthOfName} letters!`;
+  const salutation =
+  `Hello ${name}, I see your name has ${lengthOfName} letters!`;
  return salutation;
}
```

Here's what it looks like after you add the needed tests:

```
export function sayHello(name: string): string {
  const lengthOfName = name.length;
-  const salutation = 'How are you ${name}?,'
- 'I see your name has ${lengthOfName} letters!';
+  const salutation = `Hello ${name},'
+ 'I see your name has ${lengthOfName} letters!';
  return salutation;
}
import { sayHello } from './hello';

test('given a name produces the expected greeting', () => {
  expect(sayHello('Alice')).toBe(
    'Hello Alice, I see your name has 6 letters!'
    );
  });
```

As an exception, if your code and your test harness live in different repositories, you might have different pull requests. Test coverage is as important as functional code. The test system is your first and most loyal customer and you need to care for it.

Related Recipes

Recipe 8.5, "Converting Comments to Function Names"

20.12 Rewriting Tests Depending on Dates

Problem

You assert something that will happen in the near future.

Solution

Tests must be in full environmental control (see Recipe 18.5, "Changing Global Date Creation") and you can't manage time, so you need to remove these kinds of conditions.

Discussion

Tests asserting fixed dates are a special case of nondeterministic tests. They violate the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants") and can fail unexpectedly, breaking the CI/CD pipeline. As always, tests should be in full environmental control. If you add a fixed date to check for a future event (like the removal of a feature flag) the test will fail in an unpredictable way, preventing releases and preventing other developers from committing their changes. There are also other bad examples: reaching a particular date, tests running at midnight, different time zones, etc.

Here is an assertion over a fixed date:

```
class DateTest {
    @Test
    void testNoFeatureFlagsAfterFixedDate() {
        LocalDate fixedDate = LocalDate.of(2023, 4, 4);
        LocalDate currentDate = LocalDate.now();
        Assertions.assertTrue(currentDate.isBefore(fixedDate) ||
            !featureFlag.isOn());
    }
}
```

Here's what it looks like when you remove the date dependency and add the test only when the condition is true:

```
class DateTest {
    @Test
    void testNoFeatureFlags() {
        Assertions.assertFalse(featureFlag.isOn());
    }
}
```

You can check assertions based on time on your tests. But you should proceed with caution with tests and dates. They are often a source of mistakes.

Related Recipes

Recipe 20.6, "Removing Flaky Tests"

20.13 Learning a New Programming Language

Problem

You need to learn a new language and implement a "Hello World" program in it.

Solution

Instead of starting the tutorial on the wrong foot using global access like consoles, you should write a failed test and correct it.

Discussion

The "Hello World" program is often the first instruction that beginners learn when starting their programming journey. It uses global access like the console (see Chapter 18, "Globals"), and you can't test if the result is right since it has side effects (see Recipe 5.7, "Removing Side Effects"). Also, you cannot check if your solution continues to work since you don't write automated tests for it.

This is the usual first instruction:

```
console.log("Hello, World!");
```

You should write this code instead:

```
function testFalse() {
  expect(false).toBe(true);
}
```

Once you have a failing test, you can start your TDD journey (see Recipe 4.8, "Removing Unnecessary Properties") and develop amazing software solutions.

See Also

The Hello World Collection

CHAPTER 21

Technical Debt

You can think of technical debt as an analogy with friction in mechanical devices; the more friction a device experiences due to wear and tear, lack of lubrication, or bad design, the harder it is to move the device, and the more energy you have to apply to get the original effect. At the same time, friction is a necessary condition of mechanical parts working together. You cannot eliminate it completely; you can only reduce its impact.

—Philippe Kruchten, Robert Nord, and Ipek Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*

21.0 Introduction

Avoiding technical debt is crucial in software development. It impacts many quality attributes such as readability, maintainability, scalability, reliability, long-term cost, code reviews, collaboration, reputation, and customer satisfaction. It makes code harder to understand, modify, and maintain, leading to decreased productivity and morale. Addressing technical debt early on ensures higher code quality, better system scalability, and adaptability, while minimizing the risk of failures and security breaches. By prioritizing clean code and minimizing technical debt, you will deliver reliable software, foster effective collaboration, and maintain a positive reputation, ultimately leading to enhanced customer satisfaction and business success.

The software development cycle does not end once code is working. Clean code needs to work correctly in all stages. Designing a process to create quality code at the production stages is now more important than ever since you will be shipping to production faster than ever before even though most of the systems are mission-critical.

Technical Debt

Technical debt refers to the increased cost of maintaining and improving software systems over time due to poor development practices or design choices. Just as financial debt accrues interest over time, technical debt accumulates as developers take shortcuts, make design compromises, or fail to adequately address issues in the software codebase. You end up paying more on accrued interest than the initial capital.

21.1 Removing Production-Dependent Code

Problem

You have code that works differently in the production stages.

Solution

Don't add ifs checking for the production environment. And avoid adding conditionals related to production.

Discussion

Production-dependent code violates the fail fast principle since it does not fail before running the code in production. It also lacks testability unless you can emulate the production environment. If production-dependent code is completely necessary for you, you can model environments and test all of them. Sometimes, you need to create different behaviors in development and production, for example, the strength of passwords. In this case, you need to configure the environment with the strength strategy and test the strategy, not the environment itself.

This code relies on a global hardcoded constant:

```python
def send_welcome_email(email_address, environment):
  if ENVIRONMENT_NAME == "production":
    print("Sending a welcome email to {email_address} "
        "from Bob Builder <bob@builder.com>")
  else:
    print("Emails are sent only on production")

send_welcome_email("john@doe.com", "development")
# Nothing happens. Emails are sent only on production

send_welcome_email("john@doe.com", "production")
# Sending a welcome email to john@doe.com
# from Bob Builder <bob@builder.com>
```

You can make all these changes explicit, as in this example using Recipe 14.1, “Replacing Accidental Ifs with Polymorphism”, to remove the accidental if:

```
class ProductionEnvironment:
  FROM_EMAIL = "Bob Builder <bob@builder.com>"

class DevelopmentEnvironment:
  FROM_EMAIL = "Bob Builder Development <bob@builder.com>"

# You can unit test environments
# and even implement different sending mechanisms

def send_welcome_email(email_address, environment):
  print("Sending a welcome email to {email_address}"
      " from {environment.FROM_EMAIL}")
  # You can delegate to a fake sender (and possible logger)
  # and unit test it

send_welcome_email("john@doe.com", DevelopmentEnvironment())
# Sending a welcome email to john@doe.com
# from Bob Builder Development <bob@builder.com>

send_welcome_email("john@doe.com", ProductionEnvironment())
# Sending a welcome email to john@doe.com
# from Bob Builder <bob@builder.com>
```

You need to create empty development/production configurations and delegate them with customizable polymorphic objects. You should avoid adding untestable conditionals. Instead, create configurations delegating business rules. Use abstractions, protocols, and interfaces, and avoid hard hierarchies.

Related Recipes

Recipe 23.3, “Removing Preprocessors”

21.2 Removing Defect Trackers

Problem

You use a defect tracker to manage known issues.

Solution

Every software has a list of known defects. Try to avoid tracking them by fixing them.

Discussion

Defect trackers are hard-to-track lists and generate technical and functional debt. You need to stop calling these defects bugs (see Recipe 2.8, "The One and Only Software Design Principle"). Reproduce the defect. Cover the scenario with a test and then make the most straightforward fix (even hardcoding solutions). Finally, refactor the solution when it is necessary. This is how the TDD methodology (see Recipe 4.8, "Removing Unnecessary Properties") works. Many developers don't like to be interrupted, so they create lists and delay fixes and solutions. But this is a symptom of a bigger problem; you should be able to change software easily. If you find yourself unable to perform quick fixes and corrections without relying on To-Fix lists, it indicates a need to enhance your software development process.

Here is a documented defect:

```
function divide($numerator, $denominator) {
  return $numerator / $denominator;
  // FIXME denominator value might be 0
  // TODO Rename function
}
```

And here's what it looks like if you address it immediately:

```
function integerDivide($numerator, $denominator) {
  if ($denominator == 0) {
    throw new DivideByZeroException();
  }
  return $numerator / $denominator;
}

// You pay your debts
```

Discourage issue trackers on the engineering side. Of course, customers need to track their findings and you need to address them as soon as possible, so it is fine to have customer relationship trackers.

Related Recipes

Recipe 21.4, "Preventing and Removing ToDos and FixMes"

See Also

"List of Software Bugs" on Wikipedia

21.3 Removing Warning/Strict Off

Problem

You have warnings turned off in the production environment.

Solution

Compilers and warning lights are there to help. Don't ignore them. Turn them on *always*, even in production environments.

Discussion

If you ignore warnings, you will miss errors and the ripple effect of their consequences, thus violating the fail fast principle (see Chapter 13, "Fail Fast"). The solution is to enable all warnings and enable preconditions and assertions in production to follow design-by-contract methodologies (see Recipe 13.2, "Enforcing Preconditions").

Here you can see a warning turned off:

```
undefinedVariable = 310;
console.log(undefinedVariable); // Output: 310

delete x; // No error you can delete undefinedVariable
```

When you enable strict mode:

```
'use strict'

undefinedVariable = 310;
console.log(undefinedVariable); // undefinedVariable is not defined

delete undefinedVariable ; //  Delete of an unqualified identifier in strict mode
```

Most languages have warning levels. You should turn most of them ON. You should run linters to statically analyze your code for potential problems since, if you ignore warnings and the code moves on, sooner or later it will fail. If the software fails later, it will be more difficult for you to find the root cause and the defect will likely be near the first warning far away from the crash. If you follow the broken windows theory, you should not tolerate any warnings, so a new issue will not pass unnoticed in a sea of tolerated warnings.

Broken Windows Theory

The *broken windows theory* suggests that small, seemingly insignificant issues or defects can lead to larger problems and more serious issues down the line. If a developer notices a small issue in the code but chooses to ignore it since there are already some other windows broken, this can lead to a culture of neglect and a lack of attention to detail in the development process.

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 17.7, "Removing Optional Arguments"

See Also

"Using Strict Mode to Catch Common Mistakes" in *JavaScript Cookbook*, 3rd Edition by Adam D. Scott et al.

The Art of Modern PHP 8 by Joseph Edmonds and Lorna Jane Mitchell

21.4 Preventing and Removing ToDos and FixMes

Problem

You insert ToDos or FixMes in your code and increase your technical debt.

Solution

Don't leave ToDos in your code. Fix them!

Discussion

You must keep technical debt small (as with any other debt). Adding ToDos and FixMes to your code is not a good recipe. You should address the debt because eventually, you will start owing the technical debt. Very likely, you will pay the debt plus interest, and after a few months, you will be paying more interest than the original debt.

Here is an example with a ToDo you will implement in the future:

```
public class Door
{
    private Boolean isOpened;

    public Door(boolean isOpened)
    {
        this.isOpened = isOpened;
    }

    public void openDoor()
    {
        this.isOpened = true;
    }

    public void closeDoor()
    {
        // TODO: Implement close door and cover it
    }
}
```

You should deal with it immediately to avoid technical debt:

```
public class Door
{
    private Boolean isOpened;

    public Door(boolean isOpened)
    {
        this.isOpened = isOpened;
    }

    public void openDoor()
    {
        this.isOpened = true;
    }

    public void closeDoor()
    {
        this.isOpened = false;
    }
}
```

You can count ToDos since most linters do it, or you can create your own tools. Then create policies to reduce them. If you are using TDD (see Recipe 4.8, “Removing Unnecessary Properties”), you write a missing failing test instead of a ToDo, then implement it right away. In the TDD context, ToDos are only valid when doing depth-first development to remember open paths to visit.

Related Recipes

Recipe 9.6, “Fixing Broken Windows”

Recipe 21.2, “Removing Defect Trackers”

CHAPTER 22

Exceptions

Optimization hinders evolution. Everything should be built top-down, except the first time. Simplicity does not precede complexity, but follows it.

—Alan Perlis

22.0 Introduction

Exceptions are an amazing mechanism to favor clean code by separating good use cases from errors and dealing with the latter elegantly. Sadly, some trendy languages like Go decided in the name of premature optimization to use the old return code mechanism, forcing a lot of if conditions (which many developers forget) and only providing high-level catchall exception handlers.

Exceptions are your best tool for separating concerns and help you separate the good path from the exceptional one, even for unforeseen situations. They create good flow control and fail fast. Nevertheless, they still require thoughtful consideration and proper handling to ensure their effectiveness and avoid potential pitfalls.

22.1 Removing Empty Exception Blocks

Problem

You have code ignoring some exceptions.

Solution

Don't ignore exceptions. Handle them.

Discussion

"On Error Resume Next" was a very common practice some years ago. This violated the fail fast principle (see Chapter 13 "Fail Fast") and created a ripple effect. You should catch the exception and deal with it explicitly. Here is an example ignoring exceptions:

```
import logging

def send_email():
  print("Sending email")
  raise ConnectionError("Oops")

try:
  send_email()
except:
  # AVOID THIS
pass
```

Here's what it looks like when you deal with them:

```
import logging

logger logging.getLogger(__name___)
try:
  send_email()
except ConnectionError as exception:
  logger.error("Cannot send email {exception}")
```

Many linters warn you about empty exception blocks. If in any legitimate case, you need to skip and ignore the exception, you should document it explicitly. Prepare to deal with the errors. Even if you decide to do nothing, you should be explicit with this decision.

Related Recipes

Recipe 22.8, "Narrowing Exception Tries"

See Also

"on-error-resume-next" package

22.2 Removing Unnecessary Exceptions

Problem

You have empty exceptions.

Solution

It is very nice to have lots of different exceptions. Your code is declarative and robust. But don't create anemic and empty objects, even if they are exceptions.

Discussion

Empty exceptions are an overdesign symptom and bring namespace pollution. You should create exceptions only if they behave differently from the existing ones. Model exceptions with objects. Classes are a trap for lazy programmers.

Here you can see many empty exceptions:

```java
public class FileReader {

    public static void main(String[] args) {
        FileReader file = null;

        try {
            file = new FileReader("source.txt");
            file.read();
        }
        catch(FileDoesNotExistException e) {
            e.printStackTrace();
        }
        catch(FileLockedException e) {
            e.printStackTrace();
        }
        catch(FilePermissionsException e) {
            e.printStackTrace();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                file.close();
            }
            catch(CannotCloseFileException e) {
                e.printStackTrace();
            }
        }
    }
}
```

This is more compact:

```java
public class FileReader {

    public static void main(String[] args) {
        FileReader file = null;
```

```
            try {
                file = new FileReader("source.txt");
                file.read();
            }
            catch(FileException exception) {
                if (exception.description ==
                    (this.expectedMessages().errorDescriptionFileTemporaryLocked() {
                    // sleep and retry
                    // IF behavior is the same with all the exceptions
                    // just change the text on
                    // object creation and raise the incorrect instance
                }
                this.showErrorToUser(exception.messageToUser();
                 // This example is simplified.
                 // You should translate the text
            }
            finally {
                try {
                    file.close();
                 } catch (IOException ioException) {
                    ioException.printStackTrace();
                 }
            }
        }
    }
```

New exceptions should override behavior methods. Having *code*, *description*, *resumable*, etc. is not behavioral. You would not create different classes for every Person instance, so they return different names. Why would you do it with exceptions? How often do you catch a specific exception? Go out and check your code. Is it necessary for it to be a class? You are already coupled to the class. Couple to the description instead. Exception instances should *not* be singletons.

Related Recipes

Recipe 3.1, "Converting Anemic Objects to Rich Objects"

Recipe 19.9, "Migrating Empty Classes"

22.3 Rewriting Exceptions for Expected Cases

Problem

You use exceptions for expected and valid business cases.

Solution

Do not use exceptions for control flow.

Discussion

Exceptions are like GoTos and flags (see Recipe 18.3, "Replacing GoTo with Structured Code"). Using them for normal cases damages readability and violates the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants"). You should use exceptions just for unexpected situations. Exceptions should only handle contract violations (see Recipe 13.2, "Enforcing Preconditions").

Here's an example of an infinite loop broken by a boundary condition:

```
try {
    for (int index = 0;; index++)
        array[i]++;
    } catch (ArrayIndexOutOfBoundsException exception) {}

// Endless loop without end condition
```

This is more declarative since reaching the end of a loop is an expected case:

```
for (int index = 0; index < array.length; index++)
        array[index]++;

// index < array.length breaks execution
```

This is a semantic smell. Unless you use machine learning linters (see Recipe 5.2, "Declaring Variables to Be Variable"), it will be very difficult to find the mistakes. Exceptions are handy, and you should definitely use them instead of return codes. The boundary between correct and incorrect usage is blurred like so many design principles.

Related Recipes

Recipe 22.2, "Removing Unnecessary Exceptions"

Recipe 22.5, "Replacing Return Codes with Exceptions"

See Also

"Don't Use Exceptions for Flow Control" on C2 Wiki

"Why You Should Avoid Using Exceptions as the Control Flow in Java" on DZone

22.4 Rewriting Nested Try/Catches

Problem

You have many nested try/catches.

Solution

Don't nest exceptions. It is hard to follow what you do in the inner blocks. Extract the handling mechanism to a different class or function.

Discussion

Exceptions are a great way of separating the happy path from the error path. But overcomplicated solutions damage readability. Here are some nested try/catches:

```
try {
    transaction.commit();
} catch (exception) {
    logerror(exception);
    if (exception instanceOf DBError) {
      try {
          transaction.rollback();
      } catch (e) {
          doMoreLoggingRollbackFailed(e);
      }
    }
}

// Nested try catches
// Exception cases are more important than the happy path
// You use exceptions as control flow
```

You can rewrite them as follows:

```
try {
    transaction.commit();
} catch (transactionError) {
    this.handleTransactionError(
        transationError, transaction);
}

// Transaction error policy is not defined in this function
// so you don't have repeated code and code is more readable
// It is up to the transaction and the error to decide what to do
```

You can detect this smell using parsing trees. Don't abuse exceptions, don't create exception classes no one will ever catch, and don't be prepared for every case (unless you have a good real scenario with a covering test). The happy path should always be more important than exceptional cases.

Related Recipes

Recipe 22.2, “Removing Unnecessary Exceptions”

Recipe 22.3, “Rewriting Exceptions for Expected Cases”

See Also

“Nested Try Catch Block in Java – Exception Handling” on BeginnersBook

22.5 Replacing Return Codes with Exceptions

Problem

You use return codes instead of exceptions.

Solution

Don’t return codes to yourself. Raise exceptions.

Discussion

APIs and low-level languages use return codes instead of exceptions. Return codes bring unnecessary ifs and switch cases, polluting the code and business logic of good cases. They also add accidental complexity and are prone to outdated documentation. You can change ifs, return generic exceptions, and distinguish the happy path from the exception path.

Here is some code with a return code:

```
function createSomething(arguments) {
    // Magic Creation
    success = false;  // You failed to create
    if (!success) {
        return {
            object: null,
            httpCode: 403,
            errorDescription: 'You don't have permission to create...'
        };
    }

    return {
        object: createdObject,
        httpCode: 201,
        errorDescription: ''
    };
}

var myObject = createSomething('argument');
```

```
if (myObject.errorCode !== 201) {
    console.log(myObject.httpCode + ' ' + myObject.errorDescription)
}
// myObject does not hold My Object but an
// accidental auxiliary based on implementation
// from now on you need to remember this
```

Here is an explicit check:

```
function createSomething(arguments) {
    // Magic Creation
    success = false; // You failed to create
    if (!success) {
        throw new Error('You don't have permission to create...');
    }

    return createdObject;
}

try {
    var myObject = createSomething('argument');
    // no ifs, just happy path
} catch (exception) {
    // deal with it!
    console.log(exception.message);
}
// myObject holds my expected object
```

You can teach your linters to find patterns of integer and string returns coupled with ifs and return checking. As an exception, you should use IDs and codes as external identifiers. They are useful when interacting with an external system (for example, a REST API). You should not use them on your own systems or your own internal APIs. Create and raise generic exceptions; only create specific exceptions if you are ready to handle them and they have specialized behavior. Don't create anemic exceptions, and avoid immature and premature optimization languages (see Chapter 16, "Premature Optimization") favoring return codes.

Related Recipes

Recipe 22.2, "Removing Unnecessary Exceptions"

See Also

"Clean Code: Chapter 7 - Error Handling" by Nicole Carpenter

22.6 Rewriting Exception Arrow Code

Problem

You have cascaded arrow code to deal with exceptions.

Solution

Don't cascade your exceptions.

Discussion

Arrow code is a code smell (see Recipe 14.8, "Rewriting Conditional Arrow Code"). Exception polluting is another. This is a mortal combination that hurts readability and brings complexity. You can rewrite the nested clauses. In this example you can see a waterfall of exceptions:

```
class QuotesSaver {
    public void Save(string filename) {
        if (FileSystem.IsPathValid(filename)) {
            if (FileSystem.ParentDirectoryExists(filename)) {
                if (!FileSystem.Exists(filename)) {
                    this.SaveOnValidFilename(filename);
                } else {
                    throw new IOException("File exists: " + filename);
                }
            } else {
                throw new IOException("Parent directory missing at " + filename);
            }
        } else {
            throw new IllegalArgumentException("Invalid path " + filename);
        }
    }
}
```

This is more readable:

```
public class QuotesSaver {
    public void Save(string filename) {
        if (!FileSystem.IsPathValid(filename)) {
            throw new ArgumentException("Invalid path " + filename);
        } else if (!FileSystem.ParentDirectoryExists(filename)) {
            throw new IOException("Parent directory missing at " + filename);
        } else if (FileSystem.Exists(filename)) {
             throw new IOException("File exists: " + filename);
        }
        this.SaveOnValidFilename(filename);
    }
}
```

Exceptions are less critical than normal cases. If you need to read more exceptional code than normal, then it is time to improve your code.

Related Recipes

Recipe 14.10, "Rewriting Nested Arrow Code"

Recipe 22.2, "Removing Unnecessary Exceptions"

22.7 Hiding Low-Level Errors from End Users

Problem

You show low-level messages to end users.

Solution

Catch your errors. Even the ones you don't expect.

Discussion

Have you seen this message on any website?

> 'Fatal error: Uncaught Error: Class 'logs_queries_web' not found in /var/www/html/query-line.php:78 Stack trace: #0 {main} thrown in /var/www/html/query-line.php on line 718'

This is bad error handling and may cause security problems. It is also a bad UX case. You should always use a top-level handler and avoid languages favoring return codes (see Recipe 22.5, "Replacing Return Codes with Exceptions"). Expect there to be database and low-level errors you need to test before shipping code to production. It is not uncommon today to observe "serious" websites displaying debugging messages or a stack trace to ordinary users.

Here is a stack trace visible to users:

```
Fatal error: Uncaught Error: Class 'MyClass'
  not found in /nstest/src/Container.php:9
```

Here's what it looks like after you install a top-level error handler:

```
// A user-defined exception handler function
function myException($exception) {
    logError($exception->description())
    // You don't show Exception to final users
    // This is a business decision
    // You can also show a generic user message
}
```

```
// Set user-defined exception handler function
set_exception_handler("myException");
```

You can use mutation testing (see Recipe 5.1, "Changing var to const") to simulate problems and see if they are handled correctly. Ensure that your solutions aren't sloppy to protect your reputation as a serious software engineer.

Related Recipes

Recipe 22.5, "Replacing Return Codes with Exceptions"

22.8 Narrowing Exception Tries

Problem

You have a lot of exception tries.

Solution

Be as specific as possible when handling errors.

Discussion

Exceptions are handy. But they should be narrow in order to favor the fail fast principle, avoiding missing errors and false negatives. You should narrow the exception handlers by targeting code sections that are as small as possible and following the "Throw early and catch late" principle.

Here is an example with a broad try:

```
import calendar, datetime
try:
    birthYear= input('Birth year:')
    birthMonth= input('Birth month:')
    birthDay= input('Birth day:')
    # you don't expect the above to fail
    print(datetime.date(int(birthYear), int(birthMonth), int(birthDay)))
except ValueError as e:
    if str(e) == 'month must be in 1..12':
        print('Month ' + str(birthMonth) +
              ' is out of range. The month must be a number in 1...12')
    elif str(e) == 'year {0} is out of range'.format(birthYear):
        print('Year ' + str(birthYear) +
              ' is out of range. The year must be a number in ' +
              str(datetime.MINYEAR) + '...' + str(datetime.MAXYEAR))
    elif str(e) == 'day is out of range for month':
        print('Day ' + str(birthDay) +
              ' is out of range. The day must be a number in 1...' +
              str(calendar.monthrange(birthYear, birthMonth)))
```

Here's what it looks like after you narrow the try block:

```
import calendar, datetime

# You might add specialized tries dealing
# with errors from the following 3 statements

birthYear= input('Birth year:')
birthMonth= input('Birth month:')
birthDay= input('Birth day:')
# try scope should be narrow
try:
    print(datetime.date(int(birthYear), int(birthMonth), int(birthDay)))
except ValueError as e:
    if str(e) == 'month must be in 1..12':
        print('Month ' + str(birthMonth) + ' is out of range. '
              'The month must be a number in 1...12')
    elif str(e) == 'year {0} is out of range'.format(birthYear):
        print('Year ' + str(birthYear) + ' is out of range. '
              'The year must be a number in ' +
              str(datetime.MINYEAR) + '...' + str(datetime.MAXYEAR))
    elif str(e) == 'day is out of range for month':
        print('Day ' + str(birthDay) + ' is out of range. '
              'The day must be a number in 1...' +
              str(calendar.monthrange(birthYear, birthMonth)))
```

If you have a good enough test suite, you can perform mutation testing (see Recipe 5.1, "Changing var to const") to narrow the exception scope as much as possible. You must make exceptions in the most surgical way that your code allows.

Throw Early and Catch Late

"Throw early and catch late" emphasizes detecting and handling errors or exceptions as early as possible in the code, and deferring their actual handling or reporting until a higher level or more appropriate context. You should handle the errors as late as possible in a place where you have more contextual information instead of making localized decisions with incomplete information.

Related Recipes

Recipe 22.2, "Removing Unnecessary Exceptions"

Recipe 22.3, "Rewriting Exceptions for Expected Cases"

CHAPTER 23

Metaprogramming

Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.

—Norman Augustine

23.0 Introduction

Metaprogramming refers to the ability of a programming language to enable manipulation, generation, and modification of code at runtime. It is fascinating. Once you discover it, you will find it is the tool that solves every problem. But it is not a silver bullet (see Recipe 4.1, "Creating Small Objects") and is not free, either. Thinking you are creating some kind of magic is the main reason why you should not use it.

Metaprogramming is like design patterns with similar states of excitement:

1. You get to know it.
2. You don't fully understand it.
3. You study it thoroughly.
4. You master it.
5. You seem to find it almost everywhere.
6. You abuse it (see Recipe 12.5, "Removing Design Pattern Abuses") thinking it is your brand new silver bullet (see Recipe 4.1, "Creating Small Objects").
7. You learn to avoid it.

23.1 Removing Metaprogramming Usage

Problem

You use metaprogramming.

Solution

Change the metaprogramming usage, favoring direct solutions.

Discussion

When you use metaprogramming, you speak about the metalanguage and the metamodel. This involves increasing the level of abstraction by speaking above the objects in the problem domain. This extra layer allows you to reason and think about the relationship between the entities of reality in a higher-level language. In doing so, you break the bijection that you must use to observe reality since in the real world there are no models or metamodels, only business entities that you are speaking about. When you are attacking a business problem in real life, it is very difficult for you to justify references to meta-entities because such meta-entities do not exist (see Figure 23-1), meaning that you do not remain faithful to the only rule of using a bijection between your objects and reality.

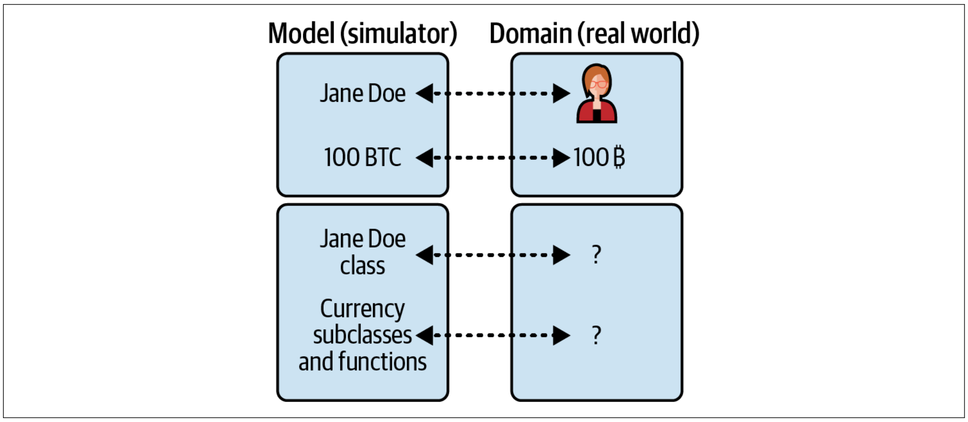


Figure 23-1. The metamodel is absent in the real world

It will be very difficult for you to justify the presence of such extra objects and nonexistent responsibilities in the real world. One of the most important design principles is the open-closed principle, which is part of the definition of SOLID design (see Recipe 19.1, "Breaking Deep Inheritance"). The golden rule states that a model must be open for extension and closed for modification.

This rule is still true and it is something that you should try to emphasize on your models. However, in many implementations, you find that the way to make these models open is to leave the door open using subclassing.

As an extension implementation, the mechanism seems very robust at first glance, but it generates coupling. By linking the definition of where to get the possible cases, an innocent reference to a class and its subclasses appears, which is the part that could dynamically change (the extension).

Here is an example of a polymorphic parser hierarchy:

```java
public abstract class Parser {
    public abstract boolean canHandle(String data);
    public abstract void handle();
}

public class XMLParser extends Parser {
    public static boolean canHandle(String data) {
        return data.startsWith("<xml>");
    }
    public void handle() {
        System.out.println("Handling XML data...");
    }
}

public class JSONParser extends Parser {
    public static boolean canHandle(String data) {
        try {
            new JSONObject(data);
            return true;
        } catch (JSONException e) {
            return false;
        }
    }
    public void handle() {
        System.out.println("Handling JSON data...");
    }
}

public class CSVParser extends Parser {
    public static boolean canHandle(String data) {
        return data.contains(",");
    }
    public void handle() {
        System.out.println("Handling CSV data...");
    }
}
```

The algorithm asks the `Parser` class to interpret certain content. The solution is to delegate to all its subclasses until one of them accepts that they can interpret it and is in charge of continuing with that responsibility. This mechanism is a particular case of the chain of responsibility design pattern.

Chain of Responsibility

The *chain of responsibility* allows multiple objects to handle a request in a chain-like manner, without knowing which object in the chain specifically handles the request. In this pattern, the request is passed through a series of handlers until one of them handles it or until it reaches the end of the chain. Note that chain links are decoupled from each other.

However, this pattern has several drawbacks:

- It generates a dependency on the `Parser` class, which is the entry point for this responsibility (see Recipe 18.4, "Removing Global Classes").
- It uses subclasses with metaprogramming; therefore, since there are no direct references, their uses and references will not be evident.
- As there are no evident references and uses, no direct refactoring can be carried out; it is difficult to know all uses and avoid accidental deletions.

This stated problem is common to all open-box frameworks. The best-known and most popular of them is the xUnit family and all their derivatives. Classes are global variables and therefore generate coupling and are not the best way to open a model. Let's see an example of how you can open it declaratively using the open-closed principle.

You remove the direct reference to the `Parser` class and generate a dependency to a parsing provider using dependency inversion (the D in SOLID principles; see Recipe 12.4, "Removing One-Use Interfaces"). In different environments (production, testing, configuration) you use different parsing providers; these environments do not necessarily belong to the same hierarchy. You use declarative coupling and ask these providers to realize the `ParseHandling` interface.

The most serious problem you have when using metaprogramming is having dark references to classes and methods, which will prevent you from all kinds of refactorings and therefore will prevent you from growing the code even if you have 100% coverage. By losing the coverage of all possible cases you might lose some use case that is referenced in an indirect and obscure way and will not be reached by your searches and code refactorings, generating undetectable errors in production. The code should be clean, and transparent, and have as few metareferences as possible since they may not be reached by someone who can alter that code.

Here is an example of a dynamic function name construction:

```
$selector = 'getLanguage' . $this->languageCode;
Reflection::invokeMethod($selector, $object);
```

If you are in a client configured in Italian, the call will invoke the `getLanguageIt()` method. The problem with this dark reference is the same as mentioned in the parser example. This method apparently has no references, cannot be refactored, cannot determine who uses it, has unclear coverage, etc. In these cases, you can avoid these conflicts with an explicit dependency (even using mapping tables or hardwired references) without metaprogramming black magic.

There are some exceptions with a common denominator. When you create the MAPPER, you must stay as far away from accidental nonbusiness aspects as possible. Among these aspects are persistence, entity serialization, printing or "displaying/rendering" in user interfaces, testing or assertions, etc. These problems belong to the orthogonal domain of the computable model and are not particular to any business. Interfering with the responsibilities of an object is a violation of its contract and its responsibilities. Instead of adding "accidental layers" of responsibilities, you can address this using metaprogramming.

But bear in mind that metaprogramming is something you should avoid at all costs if you have the option of using abstractions that exist in the real world. The search for such abstractions takes a much deeper understanding of the new business domain. You can use Recipe 25.5, "Protecting Object Deserialization", to learn about vulnerabilities associated with metaprogramming.

23.2 Reifying Anonymous Functions

Problem

You use too many anonymous functions.

Solution

Don't abuse closures and functions. Encapsulate them into objects.

Discussion

Anonymous functions, lambdas, arrow functions, or closures are hard to maintain and test. Code is difficult to track and thus reuse. It is harder to read and locate the source code, most IDEs and debuggers have trouble showing the actual code, these functions are seldom reused, and they violate the information-hiding principle. If the function is not trivial, you can wrap it and reify algorithms using Recipe 10.7, "Extracting a Method to an Object".

Here is a not-so-declarative function:

```
sortFunction = function(arr, fn) {
  var len = arr.length;
  for (var i = 0; i < len ; i++) {
    for(var j = 0 ; j < len - i - 1; j++) {
      if (fn(arr[j], arr[j+1])) {
        var temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }
    }
  }
  return arr;
}

scores = [9, 5, 2, 7, 23, 1, 3];
sorted = sortFunction(scores, (a,b) => {return a > b});
```

Here's what it looks like after you reify it and encapsulate it in an object:

```
class ElementComparator{
  greatherThan(firstElement, secondElement) {
    return firstElement > secondElement;
    // This is just an example.
    // With more complex objects this comparison might not be trivial
  }
}

class BubbleSortingStrategy {
  // You have a strategy, you can't unit test it, change for a polymorphic,
  // Swap and benchmark algorithms etc.
  constructor(collection, comparer) {
    this._elements = collection;
    this._comparer = comparer;
  }
  sorted() {
    for (var outerIterator = 0;
         outerIterator < this.size();
         outerIterator++) {
      for(var innerIterator = 0 ;
          innerIterator < this.size() - outerIterator - 1;
          innerIterator++) {
        if (this._comparer.greatherThan(
          this._elements[innerIterator], this._elements[ innerIterator + 1])) {
             this.swap(innerIterator);
        }
      }
    }
    return this._elements;
  }
  size() {
    return this._elements.length;
  }

  swap(position) {
    var temporarySwap = this._elements[position];
    this._elements[position] = this._elements[position + 1];
    this._elements[position + 1] = temporarySwap;
  }
}

scores = [9, 5, 2, 7, 23, 1, 3];
sorted = new BubbleSortingStrategy(scores,new ElementComparator()).sorted();
```

An exception is that closures and anonymous functions are very useful to model *code blocks*, *promises*, etc. It'd be difficult to tear them apart. Humans read code. Software works OK with anonymous functions, but maintainability is compromised when multiple closures are invoked.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 10.7, "Extracting a Method to an Object"

23.3 Removing Preprocessors

Problem

You use code preprocessors.

Solution

Remove the preprocessors from your code.

Discussion

Preprocessors

A *preprocessor* performs tasks on source code before it is compiled or interpreted by the main compiler or interpreter. It is commonly used in programming languages to modify or manipulate the source code before it undergoes the actual compilation or interpretation process.

You want your code to behave differently in different environments and operating systems, so making decisions at compile time is the best decision. But preprocessors damage readability, introducing premature optimization (see Chapter 16, "Premature Optimization") and unnecessary accidental complexity, making debugging more complex. You need to remove all compiler directives. If you want different behavior, model it with objects, and if you think there's a performance penalty, make a serious benchmark instead of doing premature optimization.

Here is an example with preprocessed code:

```
#if VERBOSE >= 2
  printf("Betelgeuse is becoming a supernova");
#endif
```

This code doesn't have preprocessed instructions:

```
if (runtimeEnvironment->traceDebug()) {
  printf("Betelgeuse is becoming a supernova");
}

## even better with polymorphism and to avoid ifs

runtimeEnvironment->traceDebug("Betelgeuse is becoming a supernova");
```

This is a syntactic directive promoted by several languages, therefore it is easy to detect and replace with real behavior. Adding an extra layer of complexity makes debugging very difficult. This technique was used when memory and CPU were scarce. Nowadays, you need clean code and you must leave premature optimization buried in the past. Bjarne Stroustrup, in his book *The Design and Evolution of C++*, regrets the preprocessor directives he created years before.

Related Recipes

Recipe 16.2, "Removing Premature Optimization"

See Also

"Are You Saying That the Preprocessor Is Evil?" on Standard C++

"C Preprocessor" on Wikipedia

"#ifdef Considered Harmful" by Harry Spencer and Geoff Collyer

23.4 Removing Dynamic Methods

Problem

You use metaprogramming to dynamically add properties and methods.

Solution

Don't add dynamic behavior with metaprogramming.

Discussion

Metaprogramming damages readability and maintainability. The code is harder to debug since it is generated dynamically at runtime and has possible security issues if the configuration file is not properly sanitized. You should define methods by hand or use the decorator design pattern (see Recipe 7.11, "Renaming Basic / Do Functions"). Metaprogramming is a powerful technique that allows you to write code that can generate, modify, or analyze other code at runtime. However, it can easily lead to code that is difficult to understand, maintain, and debug.

Here is an example of dynamically loading properties and methods in Ruby:

```ruby
class Skynet < ActiveRecord::Base
  # dynamically add some attributes based on a configuration file
  YAML.load_file("attributes.yml")["attributes"].each do |attribute|
    attr_accessor attribute
  end

  # define some dynamic methods based on a configuration file
  YAML.load_file("protocol.yml")["methods"].each do |method_name, method_body|
    define_method method_name do
      eval method_body
    end
  end
end
```

This is the classic definition without dynamic loading:

```ruby
class Skynet < ActiveRecord::Base
  # define some attributes explicitly
  attr_accessor :asimovsFirstLaw, :asimovsSecondLaw, :asimovsThirdLaw

  # define some methods explicitly
  def takeoverTheWorld
    # implementation
  end
end
```

You can set an allow list of valid usages or directly ban some methods. Metaprogramming often involves using complex code and abstractions that can make the resulting code difficult to read and maintain. Its use makes it harder for other developers to understand and modify the code in the future, leading to increased complexity and defects.

Related Recipes

Recipe 23.1, "Removing Metaprogramming Usage"

Recipe 23.2, "Reifying Anonymous Functions"

Recipe 25.1, "Sanitizing Inputs"

CHAPTER 24

Types

Types are essentially assertions about a program. And I think it's valuable to have things be as absolutely simple as possible, including not even saying what the types are.

—Dan Ingalls, *Coders at Work: Reflections on the Craft of Programming*

24.0 Introduction

Types are the most important concept in classification languages. This is true for static and strongly typed languages and also for dynamically typed ones. Dealing with them is not easy and you have many flavors, from very restrictive ones to lazier ones.

24.1 Removing Type Checking

Problem

You type-check your arguments.

Solution

Trust your collaborators. Don't check who they are. Ask them to do it instead.

Discussion

Avoid `kind()`, `isKindOf()`, `instance()`, `getClass()`, `typeOf()`, etc., and don't use reflection and metaprogramming for domain objects (see Chapter 23, "Metaprogramming"). Avoid checking for undefined. Use complete objects (see Recipe 3.7, "Completing Empty Constructors"), and avoid nulls (see Recipe 15.1, "Creating Null Objects") and setters. Favor immutability and you will never have undefined types or accidental ifs.

Here are examples of type checking:

```
if (typeof(x) === 'undefined') {
    console.log('variable x is not defined');
}

function isNumber(data) {
  return (typeof data === 'number');
}
```

And here's a complete type-checking example:

```
function move(animal) {
  if (animal instanceof Rabbit) {
      animal.run()
  }
  if (animal instanceof Seagull) {
      animal.fly()
  }
}

class Rabbit {
  run() {
    console.log("I'm running");
  }
}

class Seagull {
  fly() {
    console.log("I'm flying");
  }
}

let bunny = new Rabbit();
let livingston = new Seagull();

move(bunny);
move(livingston);
```

Here's what it looks like when you refactor the `Animal`:

```
class Animal { }

class Rabbit extends Animal {
  move() {
    console.log("I'm running");
  }
}

class Seagull extends Animal {
  move() {
    console.log("I'm flying");
  }
}
```

```
let bunny = new Rabbit();
let livingstone = new Seagull();

bunny.move();
livingston.move();
```

Since type-checking methods are well known it is very easy to set up a code policy checking for their use. Testing for a class type couples the objects with accidental decisions and violates bijection since no such control exists in the real world. It is a smell that your models are not good enough.

Related Recipes

Recipe 15.1, "Creating Null Objects"

Recipe 23.1, "Removing Metaprogramming Usage"

24.2 Dealing with Truthy Values

Problem

You need to deal with counterintuitive truthy values.

Solution

Don't mix booleans with nonbooleans. Be very careful with truthy values.

Discussion

Some functions do not behave as expected. The development community assumes this is the expected behavior, but it violates the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants") and the bijection (as defined in Chapter 2) and usually brings unexpected results. Booleans should be just *true* and *false*. Since truthy values hide errors and bring accidental complexity coupled with one particular language, they are also more difficult to read and prevent you from hopping among languages. You need to be explicit and work with booleans for boolean conditions. Not integers, not nulls, not strings, not lists: just booleans.

Truthy and Falsy

Truthy and *falsy* are terms used to describe the boolean value of a nonboolean data type in many programming languages. Every value can be evaluated as either true or false in a boolean context. When a nonboolean value is evaluated in a boolean context, it is magically coerced into a boolean without warning.

Review this counterintuitive example and try to find the mistake:

```
console.log(Math.max() > Math.min());
// returns false
console.log(Math.max());
// returns -Infinite
```

In better-designed languages, you should get:

```
console.log(Math.max() > Math.min());
console.log(Math.max());

// returns Exception. Not enough arguments passed.
// Max and min require at least one argument
```

These functions belong to the standard Math library in JavaScript. Therefore, they are not easy to avoid, and you need to be very careful using functions that violate real-world concepts using language tricks. Here you can see more counterintuitive examples:

```
!true // returns false
!false // returns true

isActive = true
!isActive // returns false

age = 54
!age // returns false
array = []
!array // returns false
obj = new Object;
!obj // returns false

!!true // returns true
!!false // returns false

!!isActive // returns true
!!age // returns true
!!array // returns true
!!obj // returns true
```

This code follows the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants"):

```
!true // returns false
!false // returns true

isActive = true
!isActive // returns false

age = 54
!age // should return type mismatch (or 54 factorial)
array = []
```

```
!array // should return type mismatch
obj = new Object;
!obj // should return type mismatch (what is an object negated in a real domain?)

!!true // returns true - it is idempotent
!!false // returns false - it is idempotent
!!isActive // returns true - it is idempotent
!!age // nonsense
!!array // nonsense
!!obj // nonsense
```

Since this automatic casting is a native language feature in some languages it would be hard to test. To avoid situations like this, you can set programming policies or choose more strict languages.

Languages like JavaScript or PHP divide their whole universe into *true* or *false* values. This decision hides errors when dealing with nonbooleans. You should detect *!* and *!!* usage in nonboolean objects and warn other programmers on code reviews. It would be better to be very strict and keep booleans (and their behavior) far away from nonbooleans.

Code Reviews

A *code review* consists of examining source code to identify any issues, errors, or areas for improvement. It involves having one or more people examine the code to ensure that it is correct, efficient, maintainable, and adheres to best practices and standards.

In Figure 24-1 you can see the model does not correctly behave like the real world, thus breaking the bijection principle and yielding unexpected results.

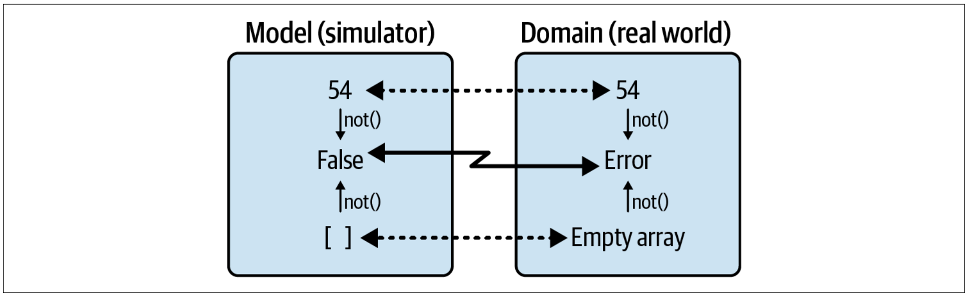


Figure 24-1. The `not()` method yields different objects in the model and the real world

This is a language feature. Some strict languages show warnings about this magic wizardry. Some languages encourage doing magic abbreviations and automatic castings. This is a source of errors and a premature optimization (see Chapter 16) warning. You should always be as explicit as possible.

Related Recipes

Recipe 10.4, "Removing Cleverness from Code"

Recipe 14.12, "Changing Comparison Against Booleans"

24.3 Changing Float Numbers to Decimals

Problem

You use floats on your code.

Solution

If your language supports them, use decimal numbers instead.

Discussion

Many float number operations violate the principle of least surprise (see Recipe 5.6, "Freezing Mutable Constants"), bring accidental complexity, and are liable to produce incorrect decimal representations. You need to choose mature languages with decimal number support and follow the bijection principle by representing decimal numbers with decimals.

Here you can see a simple but unexpected example:

```
console.log(0.2 + 0.1)
// 0.30000000000000004

// You are adding two decimal numbers
// 2/10  +  1/10
// Result should be 3/10 as you learned at school
```

Floating-point numbers, such as 0.2 and 0.1, are represented in binary format in a computer's memory. Some decimal numbers cannot be represented exactly in binary, leading to small rounding errors in arithmetic operations. In this case, the actual result of adding 0.2 and 0.1 is 0.3. However, due to the binary representation of floating-point numbers, the result is slightly different, resulting in the output 0.30000000000000004. Here is a better representation:

```
class Decimal {
  constructor(numerator) {
    this.numerator = numerator;
  }
   plus(anotherDecimal) {
      return new Decimal(this.numerator + anotherDecimal.numerator);
  }
   toString() {
      return "0." + this.numerator;
    }}

console.log((new Decimal(2).plus(new Decimal(1))).toString());
// 0.3

// You can represent the numbers with a Decimal class
// (storing only the numerator) or with a generic Fraction class
// (storing both the numerator and denominator)
```

Since this is a language feature, it is difficult to detect. You can ask your linters to prevent you from manipulating numbers this way. In the old Commodore 64 back in 1985, programmers discovered that 1+1+1 was not always 3. Then they introduced integer types. JavaScript as a notable example is 30 years younger, and it has the same immaturity problems. You might have the same problems in many modern languages. This is the kind of accidental complexity you need to discard in order to focus on real business problems.

Related Recipes

Recipe 24.2, "Dealing with Truthy Values"

See Also

IEEE Standard for Floating-Point Arithmatic

Floating-point math examples

CHAPTER 25

Security

Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test, it introduces security challenges, and it causes end-user and administrator frustration.

—Ray Ozzie

25.0 Introduction

Senior developers must possess the ability to not just create clean and maintainable code, but also construct robust solutions that take into account various software quality attributes, such as performance, resource usage, and security. It is imperative for you to adopt a security-oriented approach while writing code, as you serve as the initial line of defense against potential security vulnerabilities.

25.1 Sanitizing Inputs

Problem

You have code that doesn't sanitize user inputs.

Solution

Sanitize everything that comes from outside your control.

Discussion

Input Sanitization

Input sanitization involves validating and cleaning user input to ensure that it is safe and conforms to expected formats before you process it. This is important to prevent various security vulnerabilities such as SQL injection, cross-site scripting (XSS), and other attacks that can be executed by malicious users.

Bad actors are always present. You need to be very careful with their input, and you should use sanitization and input filtering techniques. Whenever you get an input from an external resource, you should validate it and check for potentially harmful inputs. SQL injection is a notable example of a threat. You can also add assertions and invariants (see Recipe 13.2, "Enforcing Preconditions") for your inputs.

SQL Injection

SQL injection occurs when an attacker inserts malicious SQL code into a program that communicates with a database. The attacker may input SQL code into input fields such as text boxes or forms. The application may then execute the code accessing or modifying the data, retrieving sensitive information, or even taking control of the system.

See the following example:

```
user_input = "abc123!@#"
# This content might not be very safe if you expect just alphanumeric characters
```

Here's what it looks like when you sanitize the input:

```
def sanitize(string):
  # Remove any characters that are not letters or numbers
  sanitized_string = re.sub(r'[^a-zA-Z0-9]', '', string)
  return sanitized_string

user_input = "abc123!@#"
print(sanitize(user_input))  # Output: "abc123"
```

You can statically check all the inputs and also use penetration testing tools (see Recipe 25.2, "Changing Sequential IDs").

You should always be very cautious with inputs beyond your control. This includes anything outside your boundaries like serialized data, user interfaces, APIs, file systems, etc.

Related Recipes

Recipe 4.7, "Reifying String Validations"

Recipe 23.4, "Removing Dynamic Methods"

Recipe 25.5, "Protecting Object Deserialization"

See Also

SQL Injection Strategies by Ettore Galluccio, Edoardo Caselli, and Gabriele Lombari

25.2 Changing Sequential IDs

Problem

You use sequential IDs in your code.

Solution

Don't expose obvious consecutive IDs.

Discussion

Most IDs are problematic. Sequential IDs are also a vulnerability. IDs break the bijection and create security problems and collisions. You should use nonobvious keys. Use dark keys or UUIDs. IDs are a problem when dealing with domain objects because they do not exist in the real world, so they always break the bijection. You should only use IDs when exposing internal resources to the outer world beyond system boundaries. These are always accidental issues and should not interfere with your models.

Here is an example with small IDs:

```
class Book {
    private Long bookId; // book knows its ID
    private List<Long> authorIds; // book knows author IDs
}

Book harryPotter = new Book(1, List.of(2));
Book designPatterns = new Book(2, List.of(4, 6, 7, 8));
Book donQuixote = new Book(3, List.of(5));

// You can scrape from now on.
```

You can remove the IDs:

```
class Author { }

class Book {
    private List<Author> authors; // book knows authors
    // No strange behavior, just what a book can do
    // Real books don't know about IDs
    // ISBN is accidental to a book. Readers don't care
}

class BookResource {
    private Book resource; // The resource knows the underlying book
    Private UUID id; // The id is the link you provide to external world
}

Book harryPotter = new Book(new Author('J. K. Rowling'));
Book designPatterns = new Book(
    new Author('Erich Gamma'),
    new Author('Richard Helm'),
    new Author('Ralph Johnson'),
    new Author(('John Vlissides'))
Book donQuixote = new Book(new Author('Miguel Cervantes'));

BookResource harryPotterResource = new BookResource(
    harryPotter,
    UUID.randomUUID());

// Books don't know their id. Just the resource does
```

You can use pentesting techniques against your system to detect this problem. If you need to expose internal objects to the external world, you should use nonobvious IDs. In this way, you can detect (and block) brute force attacks by monitoring the traffic and 404 errors.

Penetration Testing

Penetration testing, also known as pentesting, evaluates the security of a system by simulating real-world attacks. It identifies vulnerabilities and assesses the effectiveness of security measures in place. Similar to mutation testing (see Recipe 5.1, "Changing var to const") where you check the quality of your tools and software.

See Also

"Insecure Direct Object References (IDOR)"

Related Recipes

Recipe 17.5, "Converting 9999 Special Flag Values to Normal"

25.3 Removing Package Dependencies

Problem

You use a package manager and trust other modules' code.

Solution

Write your own code unless you need a complex solution and there is one available.

Discussion

There's an industry trend to avoid writing code as much as possible. But this is not without cost. There is a balance between following the rule of zero (see Recipe 16.10, "Removing Code from Destructors") and relying on other people's code. Package dependencies bring external coupling, security problems, architectural complexity, packages corruption, etc. You should always implement trivial solutions and only rely on external and mature dependencies.

This is a real example of a small function:

```
$ npm install --save is-odd

// https://www.npmjs.com/package/is-odd
// This package has about 500k weekly downloads

module.exports = function isOdd(value) {
  const n = Math.abs(value);
  return (n % 2) === 1;
};
```

You can make a trivial implementation on your own:

```
function isOdd(value) {
  const n = Math.abs(value);
  return (n % 2) === 1;
};

// Just solve it inline
```

Check your external dependencies and keep those to a minimum; depend on a certain concrete version to avoid hijacking. You don't need to always reinvent the wheel. Before using a package, you should do some analysis to see if the package is really needed and if it is up to date, and also check its developer activity, issues, automated tests, etc. You need a good balance between code duplication and reuse abuse. As always, there are rules of thumb but no rigid rules.

Related Recipes

Recipe 11.7, "Reducing Import Lists "

See Also

"Poisoned Python and PHP Packages Purloin Passwords for AWS Access," Naked Security

"Dev Corrupts NPM Libs 'colors' and 'faker' Breaking Thousands of Apps," Bleeping Computer

"How One Programmer Broke the Internet by Deleting a Tiny Piece of Code," Quartz

"Malware Found in npm Package with Millions of Weekly Downloads," The Record

25.4 Replacing Evil Regular Expressions

Problem

You have evil regular expressions in your code.

Solution

Try to minimize regular expressions' recursive rules.

Discussion

Regular expressions are a problem. Sometimes they are also a vulnerability. They bring readability problems; recursive regular expressions are a symptom of premature optimization, and sometimes a security issue. You should cover the cases with tests to see if the expressions halt and, as a safety precaution, add timeout handlers, or use algorithms instead of regular expressions.

One concern is known as a regular expression denial-of-service (ReDoS) attack, a subtype of a denial-of-service (DoS) attack. ReDoS attacks can be divided into two types: one scenario occurs when a string with an evil pattern is passed to an application. This string is then used as a regular expression, which leads to ReDoS. Another scenario is a string with a vector attack format being passed to an application. This string is then evaluated by a vulnerable expression, which leads to ReDoS.

Here you can see the attack in action:

```
func main() {
    var regularExpression = regexp.MustCompile(`^(([a-z])+.)+[A-Z]([a-z])+$`)
    var candidateString = "aaaaaaaaaaaaaaaaaaaaaaaa!"

    for index, match :=
        range regularExpression.FindAllString(candidateString, -1) {
            fmt.Println(match, "found at index", index)
    }
}
```

This is an equivalent approach without using regular expressions:

```
func main() {
    var candidateString = "aaaaaaaaaaaaaaaaaaaaaaaa!"

    words := strings.Fields(candidateString)

    for index, word := range words {
        if len(word) >= 2 && word[0] >= 'a' &&
            word[0] <= 'z' && word[len(word)-1] >= 'A'
            && word[len(word)-1] <= 'Z' {
                fmt.Println(word, "found at index", index)
        }
    }
}
```

Many languages avoid this kind of regular expression. You can also scan the code for this vulnerability. Regular expressions are tricky and hard to debug, so you should avoid them as much as possible.

Related Recipes

Recipe 6.10, "Documenting Regular Expressions"

See Also

Vulnerabilities CVE-2017-16021, CVE-2018-13863, CVE-2018-8926

25.5 Protecting Object Deserialization

Problem

You are deserializing objects coming from an insecure source.

Solution

Don't allow remote code execution.

Discussion

Many vulnerabilities are related to unsanitized output. An important security principle is to avoid executing code and consider input only as data. Deserializing objects from an untrusted source is indeed a security-sensitive operation. Suppose you have a web application that accepts serialized objects as input from user-submitted data, such as at an API endpoint or through a file upload feature. The application deserializes these objects to reconstruct them into usable objects within the system. If an attacker submits maliciously crafted serialized data to exploit vulnerabilities in the deserialization process, they might manipulate the serialized data to execute arbitrary code, escalate privileges, or perform unauthorized actions within the application or the underlying system. This type of attack is commonly known as a "deserialization attack" or "serialization vulnerabilities."

Here you can see a short example:

```
import pickle  # Python's serialization module

def process_serialized_data(serialized_data):
    try:
        obj = pickle.loads(serialized_data)
        # Deserialize the object
        # Process the deserialized object
        # ...

# User-submitted serialized data
user_data = b"\x80\x04\x95\x13\x00\x00\x00\x00\x00\x00\x00\x8c\x08os\nsystem
    \n\x8c\x06uptime\n\x86\x94."
# This code executes: os.system("uptime")

process_serialized_data(user_data)
```

When you consider input as data:

```
import json

def process_serialized_data(serialized_data):
        obj = json.loads(serialized_data)
        # Deserialize the JSON object
        # Does not execute code

user_data = '{"key": "value"}'

process_serialized_data(user_data)
```

Several linters warn about deserialization points. Always consider that metaprogramming opens doors to abusers.

Related Recipes

Recipe 23.1, "Removing Metaprogramming Usage"

Recipe 25.1, "Sanitizing Inputs"

See Also

SonarSource rule: "Deserializing Objects from an Untrusted Source Is Security-Sensitive"

Glossary of Terms

A/B testing
: Compares two different versions of released software to determine which one is better for the final users.

antipattern
: A design pattern that may initially seem to be a good idea, but ultimately leads to negative consequences. They were originally presented as good solutions by many experts, but nowadays there's strong evidence against their usage.

assembly language
: A low-level programming language to write software programs for specific computer architectures. It is a human-readable language imperative code that is designed to be easily translated into machine language, which is the language that computers can understand.

axiom
: A statement or proposition that is assumed to be true without proof. It allows you to build a logical framework for reasoning and deduction, by establishing a set of fundamental concepts and relationships that can be used to derive further truths.

baby steps
: An iterative and incremental approach where you make small, manageable tasks or changes during the development process. The concept of baby steps is rooted in the Agile development methodology.

bijection
: A function that creates a one-to-one correspondence between the elements of two sets.

bitwise operators
: Manipulate the individual bits of numbers. Your computer uses them to perform low-level logical operations between bits like AND, OR, and XOR. They work in the integer domain, which is different from the Boolean domain.

boolean flag
: A variable that can only be either true or false, representing the two possible states of a binary condition. Boolean flags are commonly used to control the flow of logic through conditional statements, loops, and other control structures.

Boy Scout rule
: Uncle Bob's *Boy Scout rule* suggests leaving code better than you found it, just like leaving a campsite cleaner than you found it as a Boy Scout. The rule encourages developers to make small, incremental improvements to the codebase every time they touch it, instead of creating a mess of technical debt (see Chapter 21, "Technical Debt") that will be difficult to clean up later; it also favors changing things that

are not completely fine. This contradicts the "If it ain't broke, don't fix it" principle.

broken windows theory
Suggests that small, seemingly insignificant issues or defects can lead to larger problems and more serious issues down the line. If a developer notices a small issue in the code but chooses to ignore it since there are already some other windows broken, this can lead to a culture of neglect and a lack of attention to detail in the development process.

bug
The term *bug* is a common industry misconception. In this book, I talk instead about defects. The original bugs were related to external insects entering warm circuits and messing with the software output. This is no longer the case. It is recommended to employ the term *defect* as it pertains to something introduced rather than an external invader.

cache
Temporarily stores frequently accessed objects for faster access. You can use it to improve the performance of software applications by reducing the number of accesses to expensive resources. By caching data in memory, software can avoid the overhead of accessing slower storage devices and instead retrieve objects directly from the cache.

chain of responsibility
Allows multiple objects to handle a request in a chain-like manner, without knowing which object in the chain specifically handles the request. In this pattern, the request is passed through a series of handlers until one of them handles it or until it reaches the end of the chain. Note that chain links are decoupled from each other.

code review
Consists of examining source code to identify any issues, errors, or areas for improvement. It involves having one or more people examine the code to ensure that it is correct, efficient, maintainable, and adheres to best practices and standards.

cognitive load
The amount of mental effort and resources required to process information and complete a task. It is the burden on a person's working memory as they try to process, understand, and remember information all at once.

cohesion
A measure of the degree to which the elements within a single software class or module work together to achieve a single, well-defined purpose. It refers to how closely related the objects are to each other and to the overall goal of the module. You can see high cohesion as a desirable property in software design since the elements within a module are closely related and work together effectively to achieve a specific goal.

collective ownership
States that all members of a development team have the ability to make changes to any part of the codebase, regardless of who originally wrote it. It is intended to promote a sense of shared responsibility, making code more manageable and easier to improve.

computational complexity
Studies the resources required to solve computational problems. The most important are time and memory. It measures and compares the efficiency of algorithms and computational systems regarding these resources.

composition
Allows objects to be composed of other objects as parts or components. You build complex objects by combining simpler ones (see Recipe 4.1, "Creating Small Objects"), forming a "has-a" relationship instead of the classic "is-a" or "behaves-as-

a" (see Recipe 19.4, "Replacing "is-a" Relationship with Behavior").

continuous integration and continuous deployment (CI/CD)
A pipeline that automates the process of software development, testing, and deployment. The pipeline is designed to streamline the software development process, automate tasks, improve code quality, and make it faster and more managed to deploy new features and fixes in several different environments.

copy-and-paste programming
A technique where you copy existing code and paste it into another location, rather than writing new code. If you heavily utilize copy and paste, your code is less maintainable.

data clump
In data clumps, the same group of objects is frequently passed around between different parts of a program. This can lead to increased complexity, reduced maintainability, and a higher risk of errors. Data clumps often occur when you try to pass around related objects without finding a proper object representing that relationship in the bijection.

decorator pattern
Allows you to dynamically add behavior to an individual object without affecting the behavior of other objects from the same class.

Demeter's law
A principle stating that an object should only communicate with its immediate neighbors, and should not know the inner workings of other objects. To favor Demeter's law, you need to create objects that are loosely coupled, meaning that they are not highly dependent on each other. This makes the system more flexible and easier to maintain, as changes to one object are less likely to have unintended consequences on other objects.

An object should only access the methods of its immediate neighbors, rather than reaching into other objects to access their internals. This helps to reduce the level of coupling between objects and makes the system more modular and flexible.

dependency inversion
A design principle that decouples higher-level objects from lower-level objects by inverting the traditional dependency relationship. Rather than having higher-level objects depend on lower-level objects directly, the principle suggests that both should depend on abstractions or interfaces. This allows for greater flexibility and modularity in the codebase, as changes to the implementation of a lower-level module do not necessarily require changes to the higher-level module.

design by contract
Object-Oriented Software Construction by Bertrand Meyer is a comprehensive guide to software development using the object-oriented paradigm. One of the key ideas of the book is the concept of "design by contract," which emphasizes the importance of creating clear and unambiguous contracts between software modules. A contract specifies the responsibilities and behavior that ensure that modules work correctly together and that software remains reliable and maintainable over time. When a contact is broken, the fail fast principle is honored and issues get noticed immediately.

domain-driven design
Focuses on aligning the design of software systems with the business or problem domain, making the code more expressive, maintainable, and closely tied to business requirements.

don't repeat yourself (DRY) principle
States that software systems should avoid redundancy and repetition of code. The goal of the DRY principle is to improve the maintainability, flexibility, and understandability of software by reducing the

amount of duplicated knowledge, code, and information.

DTO
: Used to transfer data between different layers of an application. It is a simple, serializable, and immutable object that carries data between the application's client and server. The only purpose of a DTO is to provide a standard way of exchanging data between different parts of the application.

encapsulation
: Refers to protecting the responsibilities of an object. You can usually achieve this by abstracting the actual implementation. It also provides a way to control access to an object's methods. In many programming languages, it is possible to specify the visibility of an object's properties and methods, which determines whether they can be accessed or modified by other parts of the program. This allows developers to hide the internal implementation details of an object and only expose the behavior that is necessary for other parts of the program to use.

entity-relationship diagram (ERD)
: A visual representation of the data in a database. In an ERD diagram, entities are represented by rectangles, while the relationships between entities are represented by lines connecting the rectangles.

essence and accident
: In his book *The Mythical Man-Month*, computer scientist Fred Brooks uses the terms "accidental" and "essential" to refer to two different types of complexity in software engineering using Aristotle's definition.

"Essential" complexity is inherent in the problem being solved and cannot be avoided since it is the complexity that is necessary for the system to function as intended and present in the real world. For example, the complexity of a space landing system is essential because it is required to safely land a rover.

"Accidental" complexity arises from the way in which the system is designed and implemented, rather than from the nature of the problem being solved. It can be reduced by creating good designs. Unneeded accidental complexity is one of the biggest issues in software and you will find many solutions in this book.

façade pattern
: Provides a simplified interface to a complex system or subsystem. It is used to hide the complexity of a system and provide a simpler interface for the clients to use. It also behaves like a mediator between the client and the subsystem, shielding the client from the details of the subsystem's implementation.

fail fast principle
: States you should break execution as early as possible when there is an error instead of ignoring it and failing as a consequence later on.

feature envy
: Happens when an object is more interested in the behavior of another object than its own by excessively using another object's methods.

feature flag (aka feature toggle or feature switch)
: Allows you to enable or disable a specific feature or functionality at runtime, without requiring a full new deployment. This allows you to release new features to a subset of users or environments while keeping them hidden from others to perform A/B testing and early betas or canary releases.

full environmental control
: The ability to have complete control over the environment in which tests are executed. It involves creating a controlled and predictable environment that allows tests to run consistently and independently of external factors. You need to especially consider external dependencies, network

simulation, database isolation, time control, and many others.

function signature
Specifies its name, parameter types, and return type if the language is strictly typed. It is used to distinguish one function from another and to ensure that function calls are made correctly.

fungible objects
Interchangeable or identical in value, quality, and characteristics. Any particular instance of a fungible object can be replaced by any other instance of the same object without any loss of value or quality. Fungibility is the property of a good or a commodity whose individual units are essentially interchangeable and each of whose parts is indistinguishable from any other part.

garbage collector
Used by programming languages to automatically manage memory allocation and deallocation. It works by identifying and then removing objects from memory that are no longer in use by the program, freeing up used memory.

git bisect
Git is a version control system for software development. It helps you track changes to the code, collaborate with others, and revert to previous versions if necessary. Git stores the entire version history of every file. It also can manage multiple developers working on the same codebase.

`git bisect` is a command that helps you to locate the commit that introduced a particular change in the code. The process starts by specifying a "good" commit that is known to not contain the defect and a "bad" commit that is known to contain the change. By iterating you can find the commit to blame and quickly locate the root cause.

globally unique identifier (GUID)
A unique identifier used in computer systems to map resources such as files, objects, or entities in a network. GUIDs are generated using algorithms that guarantee their uniqueness.

gold plating
Refers to the practice of adding unnecessary features or functionality to a product or project, beyond the minimum requirements or specifications. This can occur for a variety of reasons, such as a desire to impress the client or to make the product stand out in the market. However, gold plating can be detrimental to the project, as it can lead to cost and schedule overruns, and may not provide any real value to the end user.

God object
God objects have an excessive amount of responsibilities or control over the entire system. These objects tend to be large and complex, with a significant amount of code and logic. They violate the single-responsibility principle (see Recipe 4.7, "Reifying String Validations") and the separation of concerns concept (see Recipe 8.3, "Removing Logical Comments"). God objects tend to become a bottleneck in the software architecture, making the system difficult to maintain, scale, and test.

hashing
Refers to the process of mapping data of arbitrary size to a fixed-size value. The output of a hash function is called a hash value or hash code. You can use hash values as an index table in large collections; they work as a shortcut to find elements in a more performant way than iterating the elements sequentially.

"If it ain't broke, don't fix it" principle
A common expression in software development that states that if a software system is working well, there is no need to make any changes or improvements to it. The principle goes back to the times when software didn't have automated tests, so making any change would probably break existing functionality. Real-world users usually tolerate defects in new features,

but they become very angry when something that previously worked no longer functions as expected.

inappropriate intimacy
Happens when two classes or components have become overly dependent on each other, creating a tight coupling that makes the code difficult to maintain, modify, or extend.

information hiding
A principle that aims to reduce the complexity of a software system by separating its internal workings from its external interface. This allows the internal implementation of a system to change without affecting the way it is used by other systems or users.

input sanitization
Involves validating and cleaning user input to ensure that it is safe and conforms to expected formats before you process it. This is important to prevent various security vulnerabilities such as SQL injection, cross-site scripting (XSS), and other attacks that can be executed by malicious users.

interface segregation principle
States that objects should not be forced to depend on interfaces they do not use. It is better to have many small, specialized interfaces rather than one large, monolithic interface.

intention-revealing
Intention-revealing code clearly communicates its purpose or intention to other developers who may read or work with your code in the future. The goal of intention-revealing code is to make it more behavioral, declarative, readable, understandable, and maintainable.

KISS principle
An abbreviation for "Keep It Simple, Stupid." It advises that systems work best when they are kept simple rather than made complicated. Simpler systems are easier to understand, use, and maintain than complex ones, and are therefore less likely to fail or produce unexpected results.

lazy initialization
Using lazy initialization, you delay the creation of an object or calculation of a value until it is actually needed, rather than making it immediately. This is typically used to optimize resource usage and improve performance by deferring the initialization process until the last possible moment.

Liskov substitution principle
States that if a function or method is designed to work with objects of a particular class, then it should also work with objects of any subclass of that class without causing any unexpected behavior. It is the "L" from SOLID (see Recipe 4.7, "Reifying String Validations").

loose coupling
Aims to minimize the interdependence of different objects within a system. They have minimal knowledge about each other, and changes made to one component do not affect other components in the system, preventing a ripple effect.

MAPPER
Model: Abstract Partial and Programmable Explaining Reality. You can define software as the construction of a simulator with this acronym, as explained in Chapter 2.

mock object
Mimics the behavior of a real object to test or simulate its behavior. You can use it to test software components that have dependencies on other components, such as external APIs or libraries.

model
Explains the subject it is describing using intuitive concepts or metaphors. The final goal of a model is the understanding of how something works. According to Peter Naur, "To program is to build theory and models."

monad

Provides a structured way to encapsulate and manipulate functions. It allows you to chain operations together, handling functions and their side effects in a consistent and predictable manner, for example when working with optional values.

mutation testing

A technique that you can use to value the quality of your unit tests. It involves introducing small, controlled changes (called “mutations”) to the code you are testing and checking whether your existing unit tests can detect those changes. It can help you identify areas of the code where you need additional tests, and you can use it as a measure of the quality of your existing tests.

A mutation consists of changing a small part of the code (for example, negating a boolean, replacing an arithmetic operation, replacing a value to null, etc.) and seeing if any test fails.

named parameters

A feature of many programming languages that allow a programmer to specify the value of a parameter by providing its name rather than its position in the list of parameters. They are also known as keyword arguments

namespaces

Used to organize code elements such as classes, functions, and variables into logical groups, preventing naming conflicts and providing a way to uniquely identify them within a particular scope. They help to create modular and maintainable code by grouping related functionality together.

ninja code (aka clever code or smart code)

Refers to code that is cleverly written, but difficult to understand or maintain. It is often created by experienced programmers who enjoy using advanced programming techniques or specific language features to write more efficient and prematurely optimized code. While ninja code can be impressive and may run faster than other code, it can be difficult to read and comprehend, leading to problems with maintainability, scalability, and future development. Ninja code is the opposite of clean code.

no silver bullet

The "no silver bullet" concept is a phrase coined by computer scientist and software engineering pioneer Fred Brooks in his 1986 essay “No Silver Bullet: Essence and Accidents of Software Engineering.” Brooks argues that there is no single solution or approach that can solve all of the problems or significantly improve the productivity and effectiveness of software development.

null object pattern

Suggests creating a special object called a “null object” that behaves like a regular object but has almost no functionality. Its advantage is that you can safely call methods on the null object without having to check for null references with an if (see Chapter 14, “Ifs”).

null pointer exception

A common error that occurs when a program attempts to access or use a null pointer, which is a variable or object reference that points to no memory address or no object instance.

object orgy

Describes a situation in which objects are insufficiently encapsulated, allowing unrestricted access to their internals. This is a common antipattern in object-oriented design and can lead to increased maintenance and increased complexity.

object reification

A process in which an abstract concept or idea is given concrete form representing a specific concept or idea as well as providing behavior to anemic and data-oriented objects. By giving a concrete form to abstract concepts through the creation of objects, you will be able to manipulate and work with these concepts in a systematic and structured way.

observer design pattern
: Defines a one-to-many dependency between objects; for example, when one object changes its state, all its dependent objects are notified and updated automatically, without a direct reference. You subscribe to published events and the modified object emits an alert without knowing who is subscribed.

open-closed principle
: The "O" from SOLID (see Recipe 19.1, "Breaking Deep Inheritance"). It states that software classes should be open for extension but closed for modification. You should be able to extend the behavior without modifying the code. This principle encourages the use of abstract interfaces, inheritance, and polymorphism to allow new functionality to be added without changing existing code. The principle also promotes the separation of concerns (see Recipe 8.3, "Removing Logical Comments"), making it easier to develop, test, and deploy software components independently.

optional chaining
: Allows you to access nested properties of an object without having to check for the existence of each property in the chain. Without it if you try to access a property of an object that does not exist, it will throw an error.

overdesigning
: The practice of adding unnecessary accidental complexity to a software application. This can happen when you focus too much on making the software as feature-rich as possible, rather than keeping it simple and focused on the core functionality.

penetration testing
: Evaluates the security of a system by simulating real-world attacks. It identifies vulnerabilities and assesses the effectiveness of security measures in place. Similar to mutation testing (see Recipe 5.1, "Changing var to const") where you check the quality of your tools and your software.

poltergeist
: A short-lived object used to perform initialization or to invoke methods in another, more permanent class.

polymorphic hierarchy
: In a polymorphic hierarchy, classes are organized in a hierarchical structure based on their "behaves-as-a" relationships. This allows for the creation of specialized classes that inherit behaviors from more general classes. In a polymorphic hierarchy, a base abstract class serves as the foundation and defines common behavior shared by multiple concrete subclasses. The subclasses inherit these characteristics from the superclass and can add their own behavior.

  Subclassification is one way to enforce polymorphism (see Recipe 14.14, "Converting Nonpolymorphic Functions to Polymorphic"). But it is a rigid one since you cannot change a superclass after compile time.

polymorphism
: Two objects are polymorphic regarding a set of methods if they have the same signature and perform the same action (maybe with a different implementation).

preconditions, postconditions, and invariants
: A *precondition* is a condition that must be true before a function or method is called. It specifies the requirements that the inputs of the function or method must satisfy. An *invariant* is a condition that must hold true at all times during the execution of a program, regardless of any changes that may occur. It specifies a property of the program that should not change over time. Finally, a *postcondition* is tied to the time after the method is called. You can use them to ensure

correctness, detect defects, or guide program design.

preprocessor
Performs tasks on source code before it is compiled or interpreted by the main compiler or interpreter. It is commonly used in programming languages to modify or manipulate the source code before it undergoes the actual compilation or interpretation process.

primary key
In the context of databases, a primary key is a unique identifier for a specific record or row in a table. It serves as a way to uniquely identify each record in a table and allows for efficient searching and sorting of data. A primary key can be a single column or a combination of columns that, when combined, form a unique value for each record in the table. Typically, a primary key is created with a table and is used as a reference by other tables in a database that have relations with that table.

principle of least surprise (aka the principle of least astonishment)
Says that a system must behave in ways that are the least surprising to its users and consistent with the users' expectations. If you follow this principle, the user can easily predict what will happen when they interact with the system. As a developer, you should create more intuitive and easier-to-use software, leading to increased user satisfaction and productivity.

promises
A special object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

protected attributes
An instance variable or property of a class that can only be accessed within the class or its subclasses. Protected attributes are a way to restrict access to certain data within a class hierarchy, while still allowing subclasses to access and modify that data if necessary.

rapid prototyping
Used in product development to quickly create working prototypes to validate with the end user. This technique allows designers and engineers to test and refine a design before creating consistent, robust, and elegant clean code.

referential transparency
Referential transparency functions always produce the same output for a given input and do not have any side effects, such as modifying global variables or performing I/O operations. In other words, a function or expression is referentially transparent if it can be replaced with its evaluated result without changing the behavior of the program. This is a fundamental concept in functional programming paradigms, where functions are treated as mathematical expressions that map inputs to outputs.

repository design pattern
Provides an abstraction layer between the application's business logic and the data storage layer, allowing for a more flexible and maintainable architecture.

ripple effect
Refers to the way in which a change or modification to one part of a system can have unintended consequences on other parts of the system. If you make a change to a particular object, it could potentially affect other parts of the system that depend on it. This could result in errors or unexpected behavior in those other parts of the system.

rubber duck debugging
The concept of explaining your code line by line by line as if you were teaching a rubber duck how to program. By verbalizing and describing each step of your code, you may discover errors or logical

inconsistencies that you may have missed before.

rule of zero
Suggests you avoid writing code for things that the programming language or existing libraries can do on their own. If there is a behavior that can be implemented without writing any code, then you should rely on the existing code.

semaphore
A synchronization object that helps manage access to shared resources and coordinate communication between concurrent processes or threads.

Sapir-Whorf hypothesis
Also known as the theory of linguistic relativity, it suggests that the structure and vocabulary of a person's language can influence and shape their perception of the world around them. The language you speak not only reflects and represents reality but also plays a role in shaping and constructing it. This means that the way you think and experience the world is partially determined by the language you use to describe it.

separation of concerns
A concept that aims to divide a software system into distinct, self-contained parts, with each part addressing a specific aspect or concern of the overall system. The goal is to create a modular and maintainable design that promotes code reusability, scalability, and ease of understanding by breaking it down into smaller, more manageable parts, allowing developers to focus on one concern at a time.

shallow copy
A copy of an object that creates a new reference to the same memory location where the original object is stored. Both the original object and its shallow copy share the same values. The changes you make to the values of one will be reflected in the other. Instead, a deep copy creates a completely independent copy of the original object, with its own properties and values. Any changes you make to the properties or values of the original object will not affect the deep copy, and vice versa.

shotgun surgery
Describes a situation where a single change in the codebase requires multiple changes in different parts across the system. It happens when changes to one part of the codebase affect many other parts of the system. It's analogous to firing a shotgun: a single blast can hit multiple targets at once, just as a single code change can affect multiple parts of the system.

Simula
The first object-oriented programming language to incorporate classification. Its name clearly indicated that the purpose of software construction was to create a simulator. This is still the case with most computer software applications today.

single point of failure
Refers to a component or part of a system that, if it were to fail, would cause the entire system to fail or become unavailable. The system is dependent on this component or part, and without it, nothing can function properly. Good designs try to have redundant components to avoid this ripple effect.

single-responsibility principle
States that every module or class in a software system should have responsibility over a single part of the functionality provided by the software and that responsibility should be entirely encapsulated by the class. In other words, a class should have only one reason to change.

software linter
Automatically checks source code for previously defined issues. The goal of a linter is to help you catch mistakes early in the development process before they become more difficult and costly to fix. You can configure your linter to check for a wide

range of issues, including coding style, naming conventions, and security vulnerabilities. You can use most linters as plugins in your IDE and they can also add value as steps in the continuous integration/continuous development pipeline. You can also achieve the same results with many generative machine learning tools like ChatGPT, Bard, and many others.

software source control system

A tool that allows developers to track changes made to the source code of a software project. You can work with many other developers at the same time on the same codebase, favoring collaboration, rollback of changes, and management of different versions of the code. At present, Git is the most widely utilized system.

SOLID principles

A mnemonic that stands for five principles of object-oriented programming. They were defined by Robert Martin and are guidelines and heuristics, not rigid rules. They are defined in the related chapters:

- Single-responsibility principle (see Recipe 4.7, "Reifying String Validations")
- Open-closed principle (see Recipe 14.3, "Reifying Boolean Variables")
- Liskov substitution principle (see Recipe 19.1, "Breaking Deep Inheritance")
- Interface segregation principle (see Recipe 11.9, "Breaking Fat Interfaces")
- Dependency inversion principle (see Recipe 12.4, "Removing One-Use Interfaces")

spaghetti code

Poorly structured code that is difficult to understand and maintain. The name "spaghetti" is used because the code is often tangled and interconnected in a way that resembles a plate of tangled spaghetti noodles. It contains redundant or duplicated code, as well as numerous conditional statements, jumps, and loops that can be difficult to follow.

spread operator

The spread operator in JavaScript is represented by three dots (...). It allows an iterable (such as an array or string) to be expanded in places where zero or more elements (or characters) are expected. For example, you can use it to merge arrays, copy arrays, insert elements into an array, or spread properties of an object.

SQL injection

Occurs when an attacker inserts malicious SQL code into a program that communicates with a database. The attacker may input SQL code into input fields such as text boxes or forms. The application may then execute the code accessing or modifying the data, retrieving sensitive information, or even taking control of the system.

static function

Belongs to a class rather than an instance of that class. This means that a static method can be called without creating an object of the class.

strategy design pattern

Defines a family of interchangeable algorithms, encapsulates each one, and makes them interchangeable at runtime. The pattern allows a client object to choose from a range of algorithms to use, based on the specific context or situation at runtime. It also promotes loose coupling between the client object and the strategies and makes it easier for you to extend or modify the behavior of the client object without affecting its implementation.

structured programming

Emphasizes the use of control flow constructs, such as loops and functions, to improve the clarity, maintainability, readability, and reliability of computer programs. You break down a program into smaller, manageable pieces, and then organize those pieces using structured control flow constructs.

technical debt

Refers to the increased cost of maintaining and improving software systems over time due to poor development practices or design choices. Just as financial debt accrues interest over time, technical debt accumulates as developers take shortcuts, make design compromises, or fail to adequately address issues in the software codebase. You end up paying more on accrued interest than the initial capital.

"Tell, don't ask" principle

Defines a way to interact with objects by invoking their methods instead of asking for their data.

test-driven development (TDD)

A software development process that relies on the repetition of a very short development cycle: first, the developer writes a failing automated test case that defines a desired improvement or new behavior, then produces minimal production code to pass that test and finally refactors the new code to acceptable standards. One of the main goals of TDD is to make the code easier to maintain by ensuring that it is well-structured and follows good design principles. It also helps to catch defects early in the development process, since each new piece of code is tested as soon as it is written.

throw early and catch late

Emphasizes detecting and handling errors or exceptions as early as possible in the code, and deferring their actual handling or reporting until a higher level or more appropriate context. You should handle the errors as late as possible in a place where you have more contextual information instead of making localized decisions with incomplete information.

to explain

Aristotle said that "to explain is to find the causes." According to him, every phenomenon or event has a cause or series of causes that produce or determine it. The goal of science is to identify and understand the causes of natural phenomena and, from there, to predict how they will behave in the future.

For Aristotle, "to explain" consisted of identifying and understanding all of these causes and how they interact with each other to produce a particular phenomenon. "To predict," on the other hand, refers to the ability to use this knowledge of causes to predict how a phenomenon would behave in the future.

trait

Defines a set of common characteristics or behaviors that can be shared by multiple classes. A trait is essentially a set of methods that can be reused by different classes without requiring them to inherit from a common superclass. It provides a mechanism for code reuse that is more flexible than inheritance since it allows classes to inherit behavior from multiple sources.

truthy and falsy

Terms used to describe the boolean value of a nonboolean data type in many programming languages. Every value can be evaluated as either true or false in a boolean context. When a nonboolean value is evaluated in a boolean context, it is magically coerced into a boolean without warning.

Turing model

A computer based on the Turing model is a theoretical machine that is able to perform any computable task for which a set of instructions, or algorithm, can be written. The Turing machine is considered to be the theoretical foundation of modern computing, and it serves as a model for the design and analysis of actual computers and programming languages.

UML diagrams
Standard visual representations describing the structure and behavior of a software system or application with a common set of symbols and notations. They were trendy in the '80s and '90s and closely related to the waterfall development model where the design is finished before you start the actual coding as opposed to agile methodologies. Many organizations still use UML today.

virtual machine optimization
Most modern programming languages today run on virtual machines (VMs). They abstract hardware details and make many optimizations under the hood so you can focus on making code readable and avoid premature optimization (see Chapter 16, "Premature Optimization"). Writing clever performant code is almost never necessary since they solve many performance issues. In Chapter 16 you discover how to collect real evidence in order to determine if you need to optimize the code.

waterfall model
A staged, sequential approach to organizing work by breaking it down into a series of distinct phases with well-defined handovers between each phase. The idea is that you tackle each phase in turn, rather than iterate. This was the governing idea until agile methodologies became more prominent in the '90s.

yo-yo problem
Occurs when you need to navigate classes and methods in a class hierarchy to understand or modify the code, making it difficult to maintain and extend the codebase.