endava

—

Programming Principles

Agenda

1

KISS

Keep It Stupid Simple

Similar/equal to DTSTTCPW - Do the simplest thing that could possibly work.

We try to consider the simplest way to implement the current stories. Then we choose a practical solution that is as close to that simplicity as we can practically get.

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

3

YAGNI

You Aren’t Gonna Need It

This principle states a programmer should not add functionality until deemed necessary.

The team starts from the assumption that it isn't going to need something more than the simplest solution. The team puts the infrastructure in only if it has proof, or at least very compelling evidence, that putting the infrastructure in now will be more cost-effective than waiting.

2

DRY

Don’t Repeat Yourself

Also known as “Once and only once”.

Developers don't tolerate duplication of code. Whenever they find it, they eliminate it. When we find duplicates, we eliminate them by creating a function or a base class.

The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".

Violations of DRY are typically referred to as WET solutions, commonly taken to stand for "write everything twice" (alternatively "write every time", "we enjoy typing" or "waste everyone's time").

4

SOLID

SOLID Principles

SOLID principles are object-oriented design principles that help developers eliminate the symptoms of poor design - design smells - and build the best designs for the current set of features.

These principles are the hard-won product of decades of experience in software engineering. They are not the product of a single mind but represent the integration of the thoughts and writings of a large number of software developers and researchers. Although they are presented here as principles of object-oriented design, they are really special case of long-standing principles of software engineering.

IMPORTANT

Remember that an agile developer does not apply those principles and patterns to a big, up-front design. Rather, they are applied from iteration to iteration in an attempt to keep the code, and the design it embodies, clean. (KISS and YAGNI)

"Fool me once, shame on you. Fool me twice, shame on me" - This is a powerful attitude in software design.
To keep from loading our software with needless complexity, we may permit ourselves to be fooled once.

5

SRP

The Single-Responsibility Principle

A class should have only one reason to change.

When the requirements change, that change will be manifest through a change in responsibility among the classes. If a class assumes more than one responsibility, that class will have more than one reason to change.

The Single-Responsibility Principle is one of the simplest of the principles but one of the most difficult to get right.
Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities is much of what software design is really about.

The rest of the SOLID principles come back to this issue in one way or another.

6

OCP

The Open/Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Modules that conform to OCP have two primary attributes.
1. They are open for extension.
This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes.
2. They are closed for modification.
Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the module - whether in a linkable library, a DLL, or a .EXE file - remains untouched.

The primary mechanisms behind the Open/Closed Principle are abstraction and polymorphism.
In many ways, the Open/Closed Principle is at the heart of object-oriented design.

7

LSP

The Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Presume that we have a function f that takes as its argument a reference to some base class B. Presume also that when passed to f in the guise of B, some derivative D of B cause f to misbehave. Then D violates the LSP. Clearly, D is fragile in the presence of f.
The authors of f will be tempted to put in some kind of test for D so that f can behave properly when a D is passed to it (if type == D). This test violates OCP because now, f is not closed to all the various derivatives of B.

Violation of LSP often forces violation of OCP.
The LSP is one of the prime enablers of OCP.

8

ISP

The Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

This principle deals with the disadvantages of "fat" interfaces. Classes whose interfaces are not cohesive have "fat" interfaces. In other words, the interfaces of the class can be broken into groups of methods. Each group serves a different set of clients.

When one client forces a change on the fat class, all the other clients are affected. Thus, clients should have to depend only on methods that they call. This can be achieved by breaking the interface of the fat class into many client-specific interfaces. Each client-specific interface declares only those functions that its particular client or client group invoke. This breaks the dependence of the clients on methods that they don't invoke and allows the clients to be independent of one another.

9

DIP

The Dependency-Inversion Principle

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

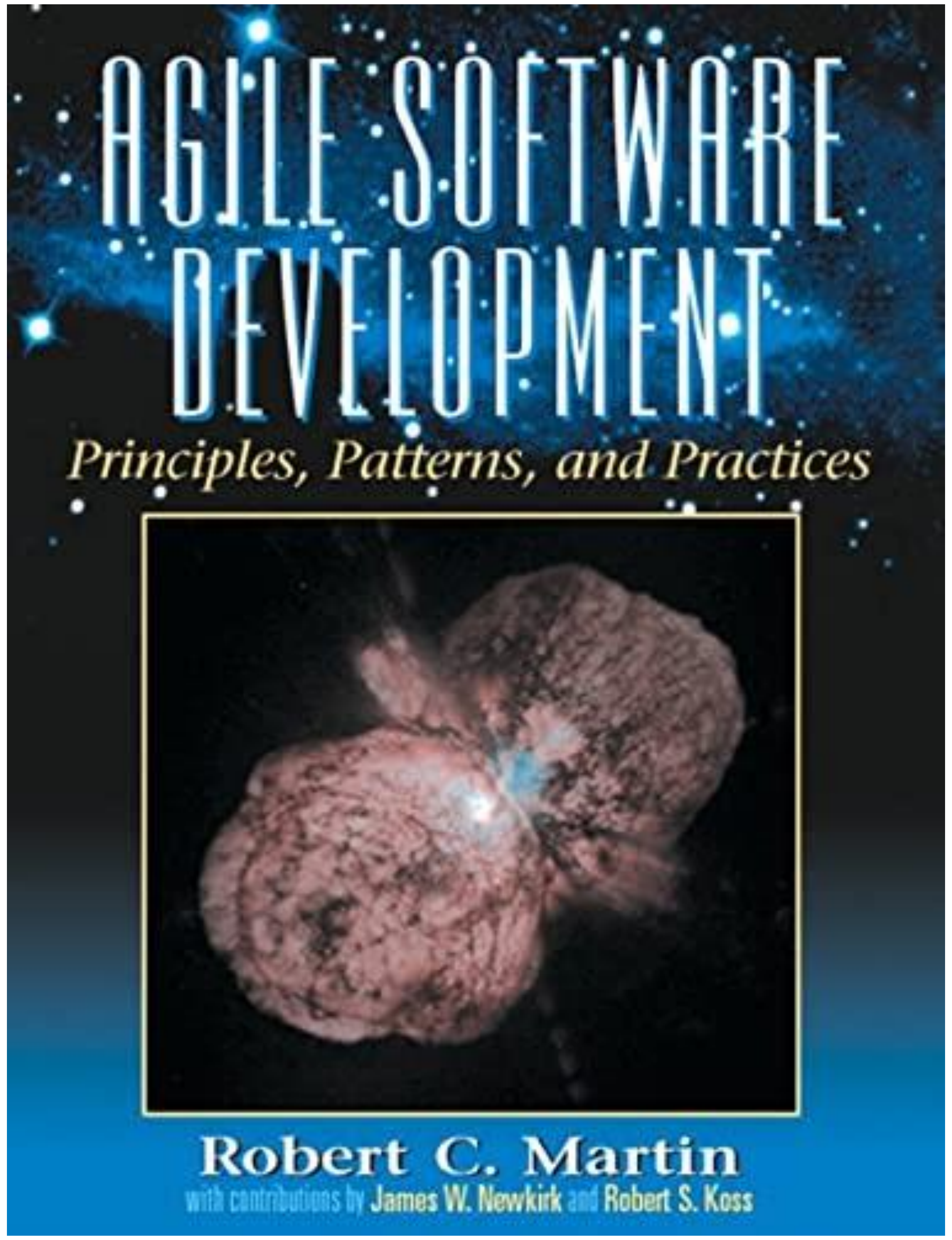
"Depend on abstractions“
You should not depend on a concrete class and that rather, all relationships in a program should terminate on an abstract class or an interface.
- No variable should hold a reference to a concrete class.
- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes.

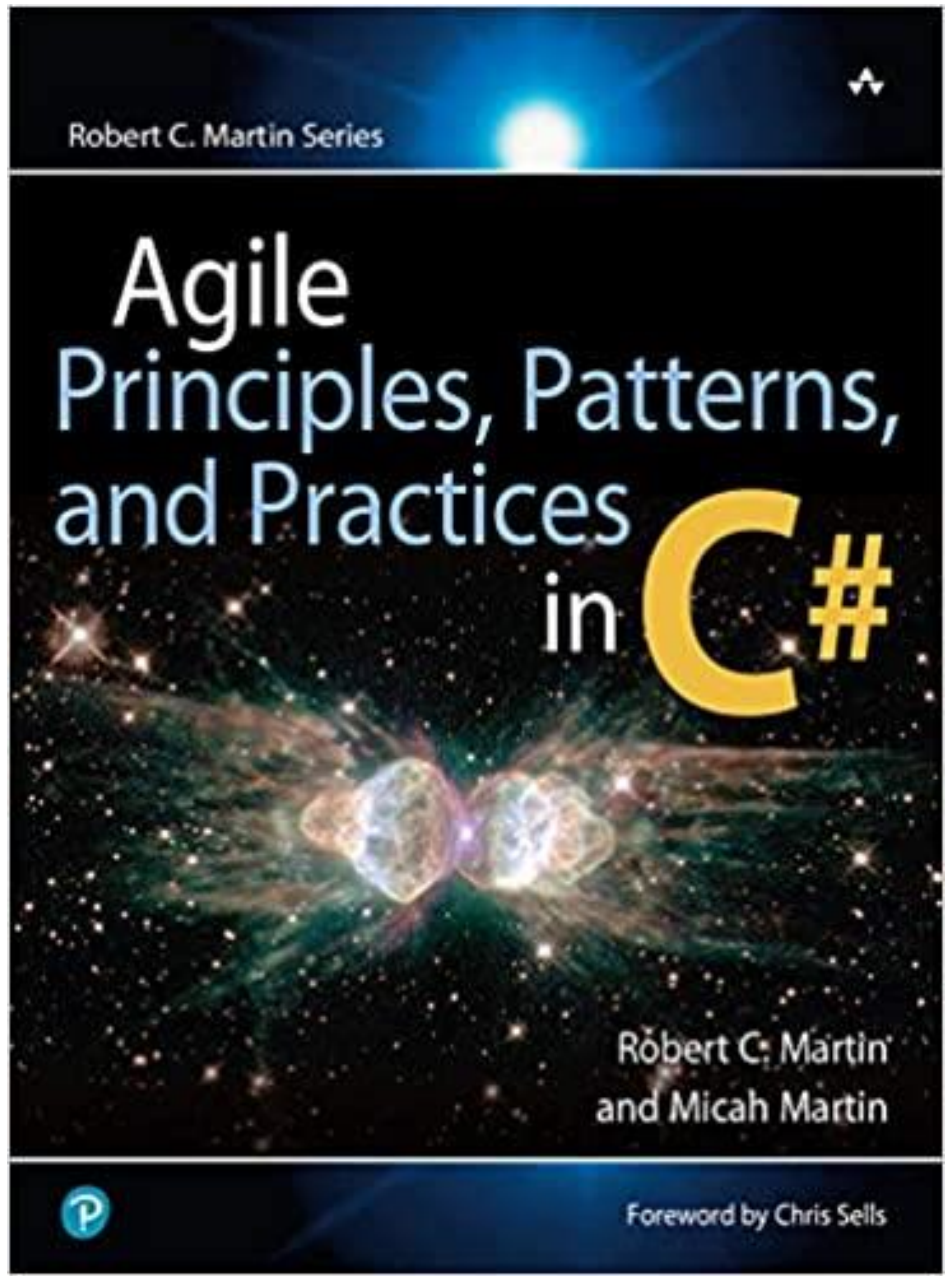
Inversion of dependencies is the hallmark of good object-oriented design.

10

Resources

Agile Software Development, Principles, Patterns, and Practices (Robert C. Martin) - Also Known as PPP (Java and C# Versions)


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss


Robert C. Martin Series
Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin
Foreword by Chris Sells

endava