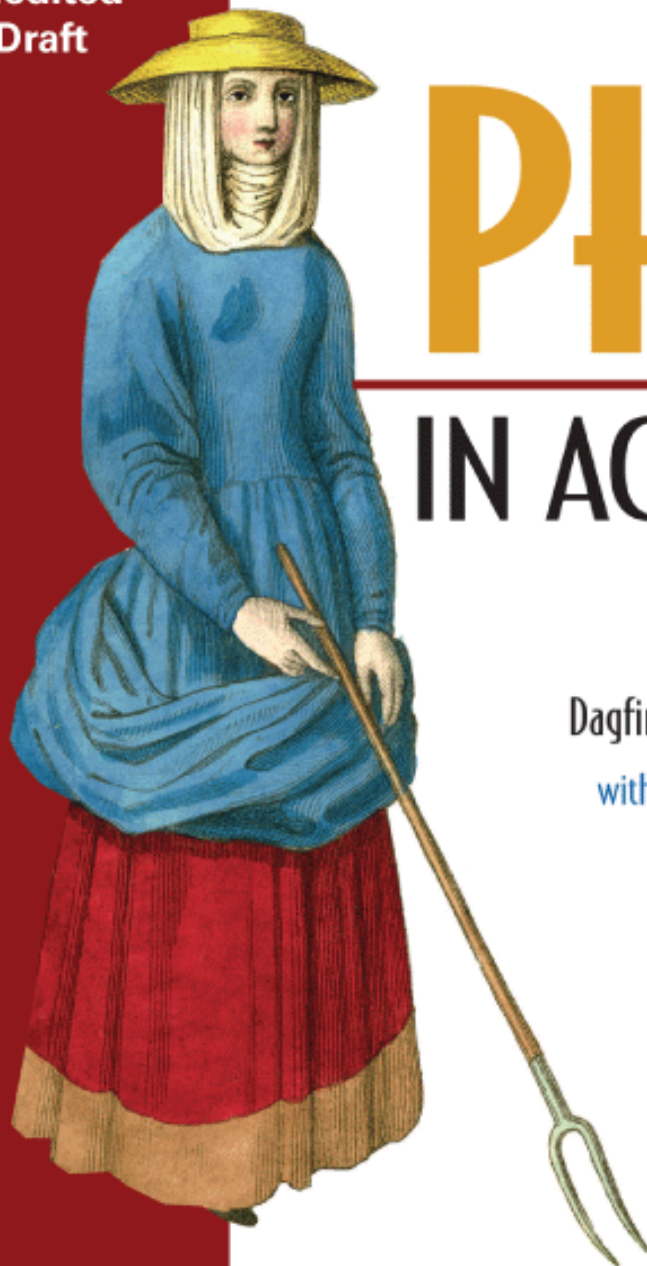


Unedited
Draft

Objects, Design, Agility



PHP

IN ACTION

Dagfinn Reiersøl

with Marcus Baker
Chris Shiflett

 MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2006 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=277>

Contents

PART 1: Basic tools and concepts

- Chapter 1 - Introduction
- Chapter 2 - Objects in PHP
- Chapter 3 - Using PHP classes effectively
- Chapter 4 - Understanding objects and classes
- Chapter 5 - Understanding class relationships
- Chapter 6 - Object-oriented principles
- Chapter 7 - Design patterns
- Chapter 8 - Design HOW-TO: date and time handling

PART 2: Testing and refactoring

- Chapter 9 - Test-driven development
- Chapter 10 - Advanced testing techniques
- Chapter 11 - Refactoring web applications
- Chapter 12 - Taking control with web tests

PART 3: Building the Web interface

- Chapter - 13 Web presentation and templates
- Chapter - 14 Constructing complex web pages
- Chapter - 15 User interaction
- Chapter - 16 Controllers
- Chapter - 17 Input validation
- Chapter - 18 Form handling

Part 4: Databases and infrastructure

- Chapter - 19 Database connection, abstraction and configuration
- Chapter - 20 Objects and SQL
- Chapter - 21 Data class design

Appendix A Tools and tips for testing

Appendix B Security

Chapter 1

PHP and modern software development

A cartoon depicts a man in a business suit, apparently a doctor, talking on the telephone: “Yes Mr. Jones, acupuncture may work for a while. Any quack treatment may work for a while. But only scientific medical practice can keep a person alive forever.”¹

This absurd and arrogant statement is obviously not likely to convince the patient. And yet, if we ignore the bizarre specifics, we can see that the fictitious doctor is at least addressing an important issue: the importance of keeping long-term goals in mind.

The long-term benefit of medical treatment is a long way from the subject matter of this book, but the long-term perspective in software development is another matter. Modern software engineering may not attempt to make software last forever, but long-term productivity is one of the key issues in the development of new technologies: principles and methodologies. This is the reason why object-oriented programming is the *de facto* standard today; it is considered a way of making software easier to maintain and to develop beyond the first version. Other buzzwords such as design patterns and agile development are also related to this.

Version 5 of PHP (recursive acronym for PHP: Hypertext Processor) is, among other things, an attempt to make it easier to use these conceptual and methodological tools in PHP.

In this book, we start from that and discover how that changes everything. We will cover three interacting goals:

- * **Explore and maximize usage of the toolkit.** We will use modern methods and tools to raise our development skills to a new level.
- * **Provide full coverage.** We will be applying the toolkit to every facet of web programming, from the user interface to database interaction.
- * **Keep it simple.** We will follow Albert Einstein's recommendation to keep everything as simple as possible, but no simpler.

Whatever your reasons for using PHP (they may be somewhat accidental, as they were for me), it's helpful to understand its strong points and even more useful to know how to overcome its limitations. That's why we start this chapter by discussing some of the pros and cons of PHP itself. Then we introduce modern object-oriented and agile methods and see how they relate to PHP.

¹ This is quoted from memory. I saw this cartoon years ago in the office of a colleague and have not seen it since.

1.1 How PHP can help you

PHP has always been a language which is especially useful for web programming. It still is, and with PHP 5 (and PHP 6, which may be released by the time you read this) it has been brought up to date and been established as a language that is fully compatible with modern object-oriented methods, practices and principles. In the following sections, we will see why PHP has become so popular as a web programming language and how to deal with the limitations of the language.

1.1.2 Why PHP is so popular

There is no doubt that PHP is a popular web programming language, at least in the sense of being heavily used. Studying the URLs of pages you visit on the Web should be enough to demonstrate that. And there has to be a reason for that fact. Some commercial products may gain popularity through massive marketing efforts, but PHP clearly is not among them.

In this section, we will see how PHP encourages a pragmatic attitude and how convenient it is—being easy to use and deploy, having important security features built in, and supporting standard ways of doing basic things. Finally, we will note how PHP also works with “enterprise” design and technology including commercial database management systems and layered or tiered architectures.

A pragmatic attitude

One thing I like about PHP is the attitude of the people who use it. PHP has always been a pragmatic solution to real problems. It's only natural that PHP programmers tend to be pragmatic rather than dogmatic, humble and open rather than conceited and pretentious. They like PHP, but they know that there is no perfect technology, no perfect programming language. Everything has its pros and cons, its advantages and disadvantages. PHP programmers tend not to start language wars. That's fortunate; often arrogance on behalf of a programming language—or any other software—is based in ignorance. You know all the things your favorite language can do, and you don't know how to do the same things in other languages. It's easy to assume that it can't be done. But it's rather like assuming that your car is the only one in the universe that has air conditioning.

Finding faults with a programming language is easy. If it lacks a feature you desperately feel you need, you can use that as a reason to put it down. If it has a feature you think is totally unnecessary, you can frown upon that. PHP 4 had no visibility constraints such as private methods; this of course was a Bad Thing to programmers who were used to languages like C++ and Java. PHP 5 has visibility constraints, and I'm sure there are others—who are accustomed to other languages that lack these features—who find this appalling.

The fact is you don't know how a feature or the lack of it works in real life until you've used it for a while. PHP has been criticized for having too many functions, in contrast to Perl who has fewer. I've used both languages extensively, and I happen to prefer the way lots of functions are easily available in PHP. Someone else may feel differently, but the most important point is that the difference is much less dramatic than some people think. Language wars are too often fought over differences that may have a marginal impact on overall productivity.

Easy to use and deploy

PHP is easy to learn. The threshold for starting to make simple web pages with dynamic content is low. Anyone who is capable of creating an HTML page will also be able to add simple dynamic content to it using PHP.

Some will lament the fact that this will let you do (some) web programming even if you are not a properly educated software engineer. But this is the way the world works. And a large part basic software development

has been about empowering users who are not computer experts, allowing them to do more and more tasks that were previously reserved for the technical gurus. In the 1960s, you couldn't even use a computer without the aid of a technical expert. That changed as interactive terminals, PCs and office software appeared. The invention of the electronic spreadsheet made it possible for end users to do calculations that previously required a programmer. And today, most applications allow a fairly wide range of customization without programming. Search engines provide easy ways to specify a search without using Boolean expressions. These are just some examples of tasks that used to require programming skills, but no longer do.

Another, more relevant, objection to PHP's low threshold of entry is the fact that it can make it seem too easy. It may foster a false impression that complex web applications using databases and having complex dynamic user interfaces can be created and maintained with just basic knowledge. But web applications are like any other software: developing and maintaining large systems with complex logic and processing requires knowledge of design principles, development methodology and programming practices. That is why books like this one exist.

Yet the simplicity of PHP for the most basic web pages—coupled with improvements that make it easier to create complex object-oriented designs—allows it to serve a continuum of needs from the simplest, humblest web sites that may have a hit counter and one simple form, to complex, highly interactive, high-volume, high-availability sites.

Another factor that makes PHP convenient is easy availability. PHP is free software; it often comes installed on Linux platforms. About 70 per cent of web servers run Apache and the PHP Apache module is installed on about half of them. Nearly all hosting services offer PHP, and it's usually cheap. So PHP is widely available, and once it's available, adding new PHP web pages is as easy as with plain HTML.

In addition, PHP programming does not require an IDE or similar development aids. There are IDEs available for PHP, but any simple text editor will do if nothing fancy is available.

“Inherently safe” features

There has been a lot of focus on the security of PHP applications in recent years. Making sure a web application is secure requires real commitment on the part of the programmer, whether the platform is PHP or something else. Many security aspects will be addressed in this book.

In spite of the difficulty of securing an application, security may be part of the reason for the success of PHP. On the operating system level, the way PHP is usually packaged and installed makes it relatively secure even when little effort and expertise is spent on security. When PHP is run as an Apache module, PHP scripts are protected and restrained by Apache. Typically, they cannot use the file system except for the pages that are available on the Web and PHP-specific include files. The scripts typically run as a user with very limited access to files on the server, and are unable to crash Apache itself.

Web application standards

Years ago, I used to say that web programming in PHP was like going on a package tour: being able to order flight and hotel reservations and even activities in one easy bundle. In a word, convenient. Perl web programming was more like having to order the hotel and the flight for yourself, while Java web programming could be likened to getting the in parts by mail order kit and having to build it yourself.

I hasten to add that this is no longer a fair description, especially in the case of Java. Although the initial cost is still higher than in PHP, you no longer have to build your own class to do something as relatively simple as encoding and decoding HTML entities.

PHP web programming is still every bit as convenient as it was, though.

When I say *standards*, I'm not referring directly to the recommendations put out by the World Wide Web Consortium (W3C). I mean built-in basic infrastructure for developing web applications. This is part of

the reason why PHP is so easy to use for simple web applications. Among other things, PHP has the following built into the language:

- A way of mixing HTML and dynamic content.
- Session handling.
- Readily available functions for all common tasks in web programming—as well as many uncommon ones.

The typical ones include functions to handle HTTP, URLs, regular expressions, database, and XML.

For simple web programming, there is little need in PHP to get and install extra packages or to build your own infrastructure beyond what's already present.

Beyond simple convenience, there is another, not widely recognized, benefit of built-in web programming infrastructure: it makes communication easier. If everybody knows the same basic mechanisms, we can assume this knowledge when explaining more advanced concepts. Session handling, for instance, can be taken for granted with no separate explanations required. So it becomes easier to keep focus on the advanced subjects. Books such as this one benefit from that fact.

Encourages use of modern principles and patterns

It may be an exaggeration to say that PHP 5 is a giant leap for programmer-kind, but for PHP programmers it represents an opportunity to use modern object-oriented programming techniques without twisting their brains into knots. Unnecessary knots, anyway, such as those caused by the awkward object reference model in PHP 4.

References really are the one impediment when using techniques like design patterns in PHP 4. Advanced object-oriented designs tend to require the ability to pass an object around without creating copies of it. It's essential that more than one object is able to hold a reference to the same object, and that changes in the referenced object is seen by the other objects. All of this is possible in PHP 4, but cumbersome. In PHP 5, it becomes as easy as in most other object-oriented languages.

PHP 5 has many other object-oriented enhancements as well, but none of them are strictly necessary to take advantage of the advances in object-oriented design.

Connects both to MySQL and other databases

One of the strengths of PHP is how easy it is to use MySQL and PHP together; there are approximately 40 books that have both PHP and MySQL in the title.

But PHP also connects to other open-source databases like PostgreSQL and to commercial ones like Oracle, DB2, Microsoft SQL server, and many others.

This is no surprise to PHP developers. But it's worth pointing out, since so-called enterprise applications typically use commercially available database management systems, and it's important to recognize that this does not preclude the use of PHP.

Works in layered architectures

Layered or tiered architectures are another mainstay of enterprise systems. As Martin Fowler points out in his book *Patterns of Enterprise Application Architecture* [P of EEA], the word “tier” usually implies a physical separation: the layers are not just separated conceptually and syntactically, but they are also running on different machines.

Either way, PHP is an option for parts of the system or all of it. This book will explore how to build all the parts of a web application using a layered architecture in PHP. There are other possibilities as well: for

example, PHP can be used as a presentation layer for a J2EE-based application. PHP will play along with most other relevant technologies and communication protocols.

We have seen some of the reasons why PHP is a successful web programming language. But what about its limitations and weaknesses? We need to know something about those, too.

1.1.3 Overcoming the limitations of PHP

Does PHP have limitations and weaknesses? Of course. As I've already admitted, there is no perfect programming language.

It's harder to decide exactly what those limitations are. They can only be judged by comparing PHP to other programming languages, and you can't do a fair comparison without extensive real-world experience of both or all the languages you are comparing

One anti-PHP web page has the following claim: "PHP works best when you forget everything you've ever learn about good programming practices. Unfortunately, that still doesn't mean that good practice is expendable. Your code will still bite." This book attempts to prove otherwise—to show exactly how good programming practices can be used effectively in PHP.

We will look at some of the criticisms of PHP and ask what can be done about them. What follows is a list of some possible or potential weaknesses and how they will be addressed in this book.

Lacks type safety

There is a never-ending discussion between programmers: some prefer statically typed languages like C, C++, Java and many others. Others prefer dynamically typed languages like PHP, Perl, Smalltalk, Python and Ruby.

Static typing means that compiler checks the types of variables before the program runs. To make this possible, the programmer must tell the compiler which variables are supposed to belong to which types. In Java, you have to explicitly name the types of all instance variables, temporary variables, return values and method arguments. In PHP, a dynamically typed language, no such declarations are necessary.

The idea of static typing is that it provides *type safety*. It's harder to introduce the wrong content into a variable because the content is likely to be of the wrong type, and in a statically typed language, the compiler will catch that during compilation. So some bugs in a program will be caught at compile time.

This is undeniably an advantage. The never-ending discussion concerns the question of whether this advantage outweighs the advantages and the convenience of dynamic typing. Are the bugs that are caught by static typing frequent and important ones? Are they bugs that would be caught early on anyway? Will statically typed languages make the code more verbose, thus making bugs harder to spot?

Whatever your position on this issue, there are ways to improve the situation. The compiler or interpreter is the first line of defense even in a dynamically typed language. The second line of defense is unit tests: testing the program in bits and pieces. Later in this chapter, we will see how unit testing is not necessarily a chore, but potentially a way to make programming less stressful and more pleasant.

The emphasis on unit testing has led some software gurus, such as Robert C. Martin, to move away from the idea that static typing is essential and to become more favorably inclined towards dynamically typed languages. This is based on the argument that type errors can be intercepted by the unit tests even when the compiler is not able to identify them.

Furthermore, object-orientation in itself increases type safety. The reason is that objects tend to fail if you try to treat them as something they're not, and that makes problems come to the surface earlier, making it easier to diagnose them. We will discuss this further in chapter 4.

Lacks namespaces

Although this may be remedied in version 6, PHP lacks a namespace feature that would make it easier to define large-scale structure and prevent name conflicts between classes. This is a real deficiency in my opinion, especially for large projects and library software. But even then it may be more of an annoyance than an insurmountable obstacle. In chapter 8, we will discuss some ways around this.

Performance and scalability issues

Critiques of PHP frequently point out specific problems that are believed to limit the performance of PHP applications.

The best comment to this is the “cranky, snarky” one from George Schlossnagle: “Technical details aside, I think PHP can be made to scale because so many people think that it can't.”

Performance, like security, depends on skill and work more than on the programming language you're using. If you believe that using a specific programming language, or even a set of software tools, will guarantee performance and scalability, you will likely fail to achieve it.

Good program design—as outlined in this book—helps you when you need high performance by making it easier to implement generic optimization strategies such as caching cleanly and without being overwhelmed by complexity.

Security loopholes

As mentioned, PHP has some security advantages. It also has some weaknesses, especially if you use older versions and features such as `register_globals`.

The only way to achieve security in web applications is to understand security and follow practices that protect against specific threats. There is an introduction to security in appendix B and secure practices are discussed throughout this book.

Security loopholes are often caused by bugs. The frequency of bugs and other defects can be drastically reduced by good program design and agile practices such as unit testing and refactoring. We will get an overview of these practices in the following section.

1.2 The new design and programming tools: languages, principles, and patterns

The evolution of software engineering and methodology since 1990 has transformed “object-oriented” from buzzword to household word (in programmer households, that is). During this time, there have also been some conceptual innovations and shifts in the object-oriented paradigm. Design patterns have become widely popular. The idea of using objects to model real-world entities has been modified or deemphasized. And the concepts of agile development have become acceptable even in polite society. PHP 5 is an attempt to incorporate some of these ideas into PHP.

In this section, we will get an overview of the most important ideas in agile development and object-oriented design. We will introduce design patterns, refactoring and unit testing, take a look at how and why they work and how they fit together, and begin to see how they can be implemented in PHP.

1.2.1 Agile methodologies: from hacking to happiness

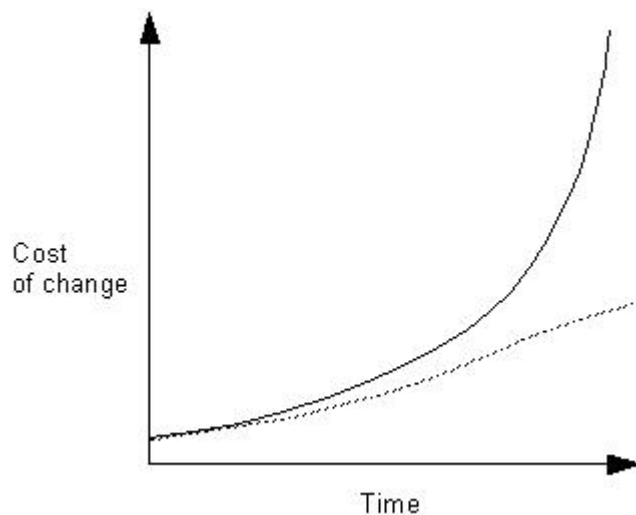
You can hack your way to success. Just start coding with no thought for the morrow, pushing eagerly ahead along the path of least resistance. It can work; that is a provable fact and worth keeping in mind. I've seen

several commercially successful programming projects with little methodology, structure, or systematic design effort.

That does not mean that I recommend it. In fact, this book is largely about how *not* to develop applications this way. Yes, you can cook spaghetti code in large batches; you can duplicate everything every time you need a variation on a feature. You can avoid planning ahead, so you understand nothing in the first place and then write code that is a complete mess, so you won't understand anything afterward either. And this may work for a while. Muddling through may be effective here as in other areas of life. But eventually, you will run into trouble.

If you choose to hack, you can typically get a lot of features done quickly in the beginning, but as your application grows in complexity, you will be slowed down by hard-to-find bugs and the need to maintain duplicated code.

The traditional alternative is typically to emphasize up-front design. The idea is that you need to plan well ahead and design everything in a relatively detailed manner before you start to code. And if you're good



at doing the design, the resemblance between what you do and what the customer needs will be sufficient to get you through to the first release without major problems. But along the way, you will probably have yielded to the temptation to make some changes that weren't planned but make the software more palatable to the users. The fact is, user requirements change, and this tends to corrupt pretty designs. As programming guru Martin Fowler puts it: "Over time the code will be modified, and the integrity of the system, its structure according to that design, slowly fades. The code slowly sinks from engineering to hacking." [Fowler Refactoring]

The problem is illustrated by the so-called cost of change curve. With time, it becomes increasingly time-consuming and costly to change the software. The problem is often illustrated by something like an exponential curve, as in figure 1.1. Agile methods are an attempt to achieve a flattening or at least a lowering of the curve, as suggested by the dotted curve in figure 1.1.

Figure 1.1 The cost of change curve in two variations

The way that agile methodologies such as Extreme Programming (XP) attempt to solve this problem is by doing less up-front design and making sure it is always possible to make design changes and by constantly improving the structure of the code using a set of systematic procedures known as refactoring.

While such a lightweight, or agile, methodology may be considered a sort of compromise between a heavy methodology and no methodology at all, it does not compromise on the quality of code or design.

Another idea that's of central importance in XP is developing software incrementally and delivering frequent releases to the customer. Developing an application without feedback from users is only slightly less dangerous than driving a car blindfolded. Unlike driving, it won't injure you physically, but you can easily end up with a product no one wants to use.

The idea is that specifying the user interface up front is insufficient. Users need to try the "look and feel" of an application. You can draw pictures of the interface, but that exposes the users to only the look, not the feel. So in agile development, it's important to get an actual application up and running as quickly as possible.

This is not a book on methodology. There are endless discussions on the merits of agile methodologies and the various practices involved, but they are beyond the scope of this book. Although some of what I will be presenting in this book may be placed in the category of agile practices, I believe that most of it falls comfortably within the realm of consensus. Whatever your methodological preferences, they should not determine this book's usefulness to you (or lack of it).

Our recipe for success is to combine the best methodology with the best software tools, and our main software tool is PHP. So let's look next at how PHP 5 relates to the methodology.

1.2.2 PHP 5 and software trends

Version 5 of PHP can be seen as the expression of at least two different trends in modern software engineering: the object-oriented trend and the simplicity trend.

The object-oriented trend has carried with it a number of innovations including several object-oriented languages, design patterns and various rules and principles. The new features of PHP 5 are specifically designed to allow PHP programmers to be a part of this trend.

On the other hand, and especially in agile development, there has been a realization that problems aren't solved simply by throwing ever more complex object-oriented constructs at them, and that complexity should be kept at a minimum. PHP helps with this, too, owing to the convenience and simplicity of PHP for basic web programming tasks, and the fact that the new object-oriented features are mostly optional.

1.2.4 The evolving discipline of object-oriented programming

When object-oriented programming started to take over the world, it was generally considered a way to model the real world. Since the real world contains objects and actions, object-oriented languages seemed appropriate. And it seemed natural to model relationships between concepts as relationships between classes. Class inheritance is supposed to model an “is-a” relationship, so since a news article is a document, the

NewsArticle class should inherit from the Document class as shown in the UML class diagram in figure 1.2.

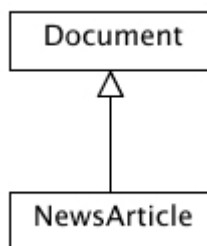


Figure 1.2 The “is-a” relationship

But the emphasis has shifted from modeling the real world to decoupling between software components. Programs are easier to maintain if you have “plug and play”: if you can use standardized components easily, replace one class with another without disturbing the rest of the system and add new features with as little change to existing code as possible. Decoupling refers to the fact that there is less dependency, less commitment so to speak, between parts of the program.

And decoupling does not necessarily, or even most of the time, imply modeling the real world. It is mostly related to the mechanical interaction in the software itself—and to the user requirements it satisfies—rather than with its theoretical and conceptual relationship to the rest of the world.

A conceptual inheritance relationship implemented in software helps decoupling to some extent. But often the way to decoupling is to isolate parts of the behavior of a class into a separate component. Just for the sake of the discussion, let's assume that the only difference between a news article and other documents is in the way summaries are handled. We could have a separate summary component that's used by the document class, as shown in figure 1.3. Ignoring the fact that there is now a new “is-a” relationship, the key fact expressed by the diagram is the ability of the Document class to use either of the document-specific summary classes interchangeably. The summary is separately pluggable. What we've done to get here is analyze the “is-a” relationship to find what behaviors are actually relevant in the particular case.

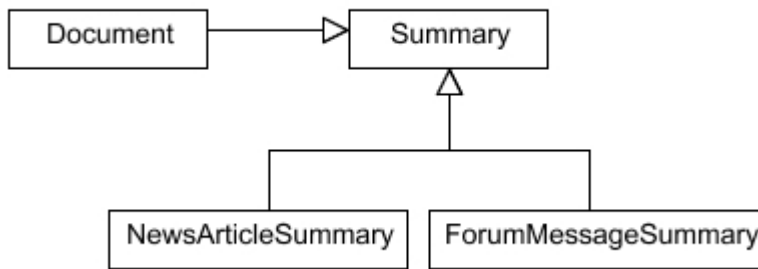


Figure 1.3 By analyzing the “is-a” relationship, we can focus on the behavior that is important

We will return to this issue repeatedly in later chapters, particularly chapters 5 and 6.

There is an area of overlap between real-world modeling and decoupling. It has to do with *abstraction*. Abstraction is a natural part of modeling the real world, in fact, it's a necessary part. In object-oriented programming, a class such as Document is an abstraction since it represents any number of concrete instances—any number of actual documents. Abstraction is also a way to achieve decoupling, since a component that is defined by an abstract interface can easily be replaced by another component with the same interface.

In programming, abstraction is often expressed by abstract classes and interfaces. In PHP, these were introduced in version 5. Whether they are actually necessary to abstract design is a question we will begin to answer in chapter 2.

1.2.5 Design patterns

Software design patterns started to become generally known after the book *Design Patterns*, by the so-called “Gang of Four” [Gang of Four], was published in 1995. It represents the trend away from a strong emphasis on real-world modeling, since the design patterns are primarily vehicles for decoupling: enabling parts of the software to vary independently of each other.

Later, there has been a virtual explosion in more or less complex design patterns. Today, there are so many available that simple finding the one you might need for a particular purpose may be a time-consuming task. This book will focus mostly on the patterns that are most relevant to web programming and to the web programmer's everyday tasks.

The interest in design patterns has started to become serious in the PHP community only in the last few years. PHP developers haven't had much of a culture for this kind of thing, but nowadays you can easily find examples of design patterns in PHP.

12.6 Refactoring

Refactoring means to improve the design of existing code. You're not adding features, just moving, splitting, merging, deleting and renaming. It is a way of keeping code simple so that it stays easy to maintain it and to add features even as it grows in complexity.

Without refactoring, it's easy to get into a one-way street that leads eventually to the death of the program. The poorer the structure of the code, the more you may have to resort to what some colleagues of mine used to call “approximate programming”. As I understand the expression, it refers to the fact that if you don't understand your own code, you can still make changes by acting on hunches and trying them out until you find something that works. Unfortunately, approximate programming muddles the code even further and make the job still harder the next time around. Frequently, you'll end up needing to re-implement the whole thing.

There are known and unknown species of refactoring. Martin Fowler and others have done us the service of cataloging a number of refactorings found in the field. Fowler's book *Refactoring* [Fowler Refactoring] has specific, step-by-step instructions on how to do each of them.

Automated tests are the key to refactoring. They make it possible to test the code between each small step in refactoring. Doing this kind of repeated testing manually would be far too time-consuming. So if you have no automated tests, you will put off testing until you are finished refactoring. When you finally start testing,

you will likely find—or fail to find, depending on your thoroughness—several bugs. And very likely there will be one or more bugs that are hard to find because you don't know where they are located.

When you have sufficient automated tests set up, you can refactor one small step at a time. You move or change some code and then you test. If a test fails, you know the problem is somewhere in the part of the program you just changed. You know approximately where the bug is, and you can locate it quickly.

For effective unit testing, you need a test framework. The best known unit testing frameworks for PHP are PHPUnit and SimpleTest. In this book, we will be using SimpleTest for the most part, but appendix A has the basics of the PHPUnit API.

At this writing, there are no refactoring tools for PHP. We have edit the code manually. Chapter 11 of this book introduces some of the techniques that are useful in typical web applications. In addition, refactoring PHP 5 is very similar to Java. The techniques in Martin Fowler's book classic book Refactoring [Fowler Refactoring] are not hard to apply in PHP.

1.2.7 Unit testing and test-driven development

“It tastes *healthy!*” my daughter objected when I tried to get her to take her medicine at age three. Software testing is similar. It's supposed to be good for you, to improve the quality of your programs and indirectly your success, your paycheck, and your quality of life in general. In spite of this, testing is not generally considered a pleasant or high-status activity. Kent Beck, who is one of the pioneers of agile development and the creator of Extreme Programming, calls it “the ugly stepchild of software development.” So maybe it just tastes too healthy.

That's how it's always been, anyway. But in recent years, testing has had a surge in popularity. Programmers are getting “test infected”, or you might say addicted. Some are even claiming that it's fun: “Test-driven development is a lot more fun than writing tests after the code seems to be working. Give it a try!” (<http://junit.sourceforge.net/doc/faq/faq.htm#best>).

The buzzword is test-driven development (TDD) or test-first development. But how does it work? How *can* it work? How can you test something that doesn't even exist? Why would any sane individual want to try it?

Part of the answer is that test-driven development is one of the sanest things you can do. It makes your programs work better, and it feels much better.

That automated testing would make programs work better because they have fewer bugs is at least logical. But why should test-driven development feel better? Why is it more fun?

It feels better because it's less stressful and more satisfying than most other ways to program. You spend less time searching for bugs and more time programming. That is one source of stress eliminated. You get fewer complaints from dissatisfied users/customers. You get the freedom to play with and change the structure of your code. That means you can learn more. I recently read that brain researchers had found that learning has some of the same effect on the brain that cocaine does. (I assume they weren't referring to harmful effects, or the educational system would be in deep trouble.)

TDD also helps you produce code of higher quality, code that you can read with satisfaction and say, “this is pretty good”.

Writing the tests before the code might seem like putting the cart before the horse, but if you think about it, it's perfectly reasonable. It's a way of getting more mileage from the tests. They do some good if even if you develop them afterwards, but you miss part of the value.

Why? Because the tests are a help from the very first time the code is running and even before that. If you develop a function and then write a test afterwards, you have no benefit of having an automated test during the early stages of debugging. Figure 1.4 shows how this works.

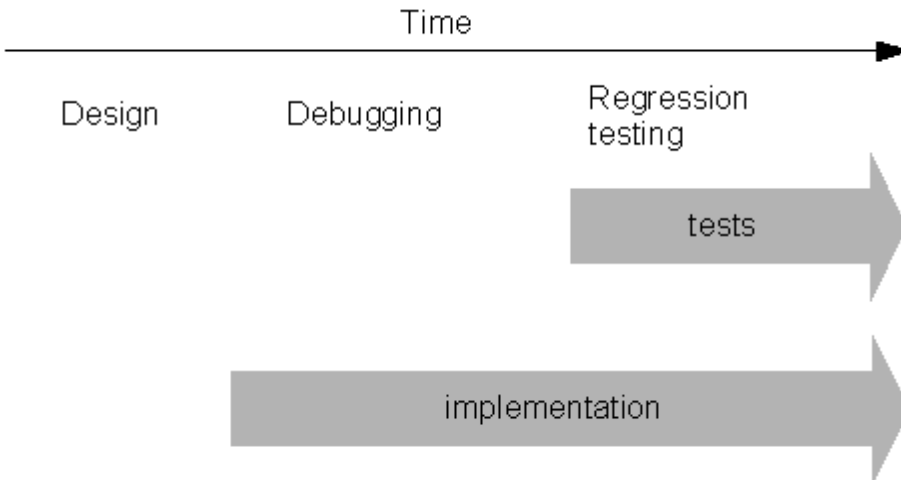


Figure 1.4 In traditional testing, the tests are helpful only after the features have been implemented.

In contrast, TDD lets you benefit from the tests while implementing the feature being tested, and even before implementation, as shown in figure 1.5.

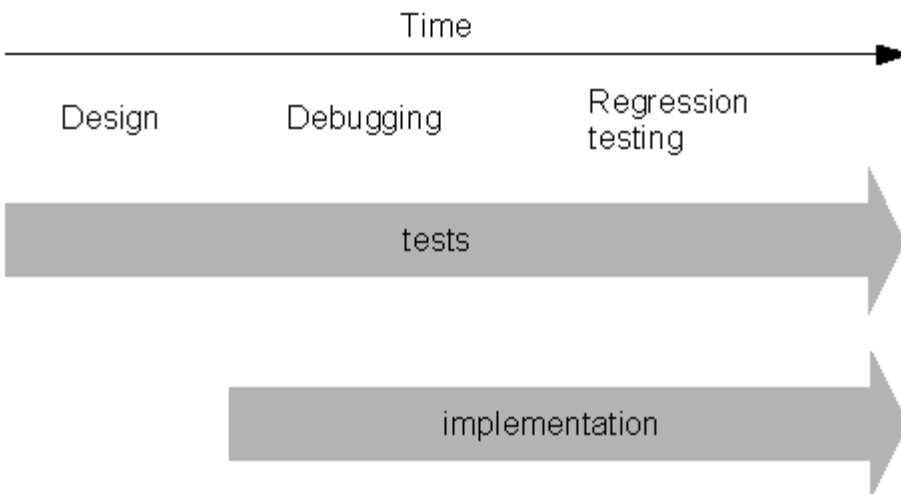


Figure 1.5 In test-driven development, the tests are doing useful work much earlier.

If you have a test ready from the start, the need for debugging tools is slight. The tests help you see what the code is doing, and help you pinpoint the location of a bug when it first appears.

All of this could be achieved as well by writing the function to be tested and then the test immediately afterwards, before actually running the function. But there is one more important advantage to writing tests first: it helps design, too. A unit test is client code for the function or method (usually) you want to develop. When writing client code first, you'll see what sequence of calls and what parameters are needed and what is a convenient way to structure them for actual use.

None of this means that you should test more than is necessary to make the program work. The agile principle is to test anything that might fail. Some pieces of code are so simple that in practice they don't fail. It's no fun writing pointless tests. On the other hand, it's easy to underestimate the likelihood of bugs.

When I recommend the test-first approach, don't take my word for it. Try it and see how it works. But you have to try it properly or you won't get the full benefit. You have to actually write the tests first and then

the code. If you've been programming for a while, this could mean breaking some ingrained habits; I certainly had to do that.

Unit testing is a prerequisite for the rest of the agile practices. Figure 1.6 shows how it interacts with some of the other practices.

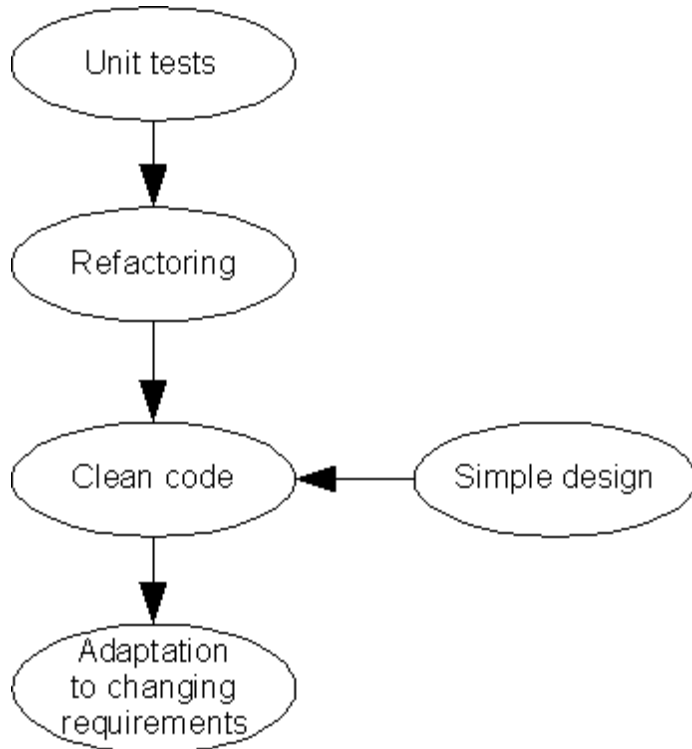


Figure 1.6 How some essential elements of agile development depend on each other.

Unit testing makes refactoring practicable. Refactoring and simple design enables us to achieve clean, maintainable code. And maintainable code is necessary if we want to be able to adapt to changing requirements.

Test-driven development will be covered in chapters 9, 10 and 12. For a deeper treatment, try Kent Beck's book *Test-Driven Development by Example*. [Beck]

1.3 Summary

PHP is a popular web programming language that is ready to meet today's design principles and practices. PHP 5 came at the right time; while keeping the convenience of earlier versions of PHP 4, it enables us to go further in implementing advanced object-oriented designs. To help us achieve this, we will use agile methods, object-oriented principles, design patterns, refactoring, and unit testing.

In the next chapter, we will start exploring how object-oriented programming works in PHP. We will look at the basics and some features that were introduced in PHP 5, including exceptions, object references and the ability to intercept method calls.