CHAPTER 2

# A Pragmatic Approach

There are certain tips and tricks that apply at all levels of software development, processes that are virtually universal, and ideas that are almost axiomatic. However, these approaches are rarely documented as such; you'll mostly find them written down as odd sentences in discussions of design, project management, or coding. But for your convenience, we'll bring these ideas and processes together here.

The first and maybe most important topic gets to the heart of software development: *The Essence of Good Design*. Everything follows from this.

The next two sections, *DRY—The Evils of Duplication* and *Orthogonality*, are closely related. The first warns you not to duplicate knowledge throughout your systems, the second not to split any one piece of knowledge across multiple system components.

As the pace of change increases, it becomes harder and harder to keep our applications relevant. In *Reversibility*, we'll look at some techniques that help insulate your projects from their changing environment.

The next two sections are also related. In *Tracer Bullets*, we talk about a style of development that allows you to gather requirements, test designs, and implement code at the same time. It's the only way to keep up with the pace of modern life.

*Prototypes and Post-it Notes* shows you how to use prototyping to test architectures, algorithms, interfaces, and ideas. In the modern world, it's critical to test ideas and get feedback before you commit to them whole-heartedly.

As computer science slowly matures, designers are producing increasingly higher-level languages. While the compiler that accepts "make it so" hasn't yet been invented, in *Domain Languages* we present some more modest suggestions that you can implement for yourself.

Finally, we all work in a world of limited time and resources. You can survive these scarcities better (and keep your bosses or clients happier) if you get good at working out how long things will take, which we cover in *Estimating*.

Keep these fundamental principles in mind during development, and you'll write code that's better, faster, and stronger. You can even make it look easy.


## The Essence of Good Design

The world is full of gurus and pundits, all eager to pass on their hard-earned wisdom when it comes to How to Design Software. There are acronyms, lists (which seem to favor five entries), patterns, diagrams, videos, talks, and (the internet being the internet) probably a cool series on the Law of Demeter explained using interpretive dance.

And we, your gentle authors, are guilty of this too. But we'd like to make amends by explaining something that only became apparent to us fairly recently. First, the general statement:

**Tip 14** Good Design Is Easier to Change Than Bad Design

A thing is well designed if it adapts to the people who use it. For code, that means it must adapt by changing. So we believe in the ETC principle: *Easier to Change. ETC.* That's it.

As far as we can tell, every design principle out there is a special case of ETC.

Why is decoupling good? Because by isolating concerns we make each easier to change. ETC.

Why is the single responsibility principle useful? Because a change in requirements is mirrored by a change in just one module. ETC.

Why is naming important? Because good names make code easier to read, and you have to read it to change it. ETC!

### ETC Is a Value, Not a Rule

Values are things that help you make decisions: should I do this, or that? When it comes to thinking about software, ETC is a guide, helping you choose between paths. Just like all your other values, it should be floating just behind your conscious thought, subtly nudging you in the right direction.

But how do you make that happen? Our experience is that it requires some initial conscious reinforcement. You may need to spend a week or so deliberately asking yourself "did the thing I just did make the overall system easier or harder to change?" Do it when you save a file. Do it when you write a test. Do it when you fix a bug.

There's an implicit premise in ETC. It assumes that a person can tell which of many paths will be easier to change in the future. Much of the time, common sense will be correct, and you can make an educated guess.

Sometimes, though, you won't have a clue. That's OK. In those cases, we think you can do two things.

First, given that you're not sure what form change will take, you can always fall back on the ultimate "easy to change" path: try to make what you write replaceable. That way, whatever happens in the future, this chunk of code won't be a roadblock. It seems extreme, but actually it's what you should be doing all the time, anyway. It's really just thinking about keeping code decoupled and cohesive.

Second, treat this as a way to develop instincts. Note the situation in your engineering day book: the choices you have, and some guesses about change. Leave a tag in the source. Then, later, when this code has to change, you'll be able to look back and give yourself feedback. It might help the next time you reach a similar fork in the road.

The rest of the sections in this chapter have specific ideas on design, but all are motivated by this one principle.

## Related Sections Include


- Topic 9, *DRY—The Evils of Duplication*, on page 30
- Topic 10, *Orthogonality*, on page 39
- Topic 11, *Reversibility*, on page 47
- Topic 14, *Domain Languages*, on page 59
- Topic 28, *Decoupling*, on page 130
- Topic 30, *Transforming Programming*, on page 147
- Topic 31, *Inheritance Tax*, on page 158


## Challenges

- Think about a design principle you use regularly. Is it intended to make things easy-to-change?

- Also think about languages and programming paradigms (OO, FP, Reactive, and so on). Do any have either big positives or big negatives when it comes to helping you write ETC code? Do any have both?

  When coding, what can you do to eliminate the negatives and accentuate the positives?[1]

- Many editors have support (either built-in or via extensions) to run commands when you save a file. Get your editor to popup an *ETC?* message every time you save[2] and use it as a cue to think about the code you just wrote. Is it easy to change?


## DRY—The Evils of Duplication

Giving a computer two contradictory pieces of knowledge was Captain James T. Kirk's preferred way of disabling a marauding artificial intelligence. Unfortunately, the same principle can be effective in bringing down *your* code.

As programmers, we collect, organize, maintain, and harness knowledge. We document knowledge in specifications, we make it come alive in running code, and we use it to provide the checks needed during testing.

Unfortunately, knowledge isn't stable. It changes—often rapidly. Your understanding of a requirement may change following a meeting with the client. The government changes a regulation and some business logic gets outdated. Tests may show that the chosen algorithm won't work. All this instability means that we spend a large part of our time in maintenance mode, reorganizing and reexpressing the knowledge in our systems.

Most people assume that maintenance begins when an application is released, that maintenance means fixing bugs and enhancing features. We think these people are wrong. Programmers are constantly in maintenance mode. Our understanding changes day by day. New requirements arrive and existing requirements evolve as we're heads-down on the project. Perhaps the environment changes. Whatever the reason, maintenance is not a discrete activity, but a routine part of the entire development process.

When we perform maintenance, we have to find and change the representations of things—those capsules of knowledge embedded in the application. The problem is that it's easy to duplicate knowledge in the specifications,

1. To paraphrase the old Arlen/Mercer song…
2. Or, perhaps, to keep your sanity, every 10th time…

processes, and programs that we develop, and when we do so, we invite a maintenance nightmare—one that starts well before the application ships.

We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY principle:

*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

Why do we call it DRY?

**Tip 15** DRY—Don't Repeat Yourself

The alternative is to have the same thing expressed in two or more places. If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction. It isn't a question of whether you'll remember: it's a question of when you'll forget.

You'll find the DRY principle popping up time and time again throughout this book, often in contexts that have nothing to do with coding. We feel that it is one of the most important tools in the Pragmatic Programmer's tool box.

In this section we'll outline the problems of duplication and suggest general strategies for dealing with it.

## DRY Is More Than Code

Let's get something out of the way up-front. In the first edition of this book we did a poor job of explaining just what we meant by *Don't Repeat Yourself*. Many people took it to refer to code only: they thought that DRY means "don't copy-and-paste lines of source."

That *is* part of DRY, but it's a tiny and fairly trivial part.

DRY is about the duplication of *knowledge*, of *intent*. It's about expressing the same thing in two different places, possibly in two totally different ways.

Here's the acid test: when some single facet of the code has to change, do you find yourself making that change in multiple places, and in multiple different formats? Do you have to change code and documentation, or a database schema and a structure that holds it, or…? If so, your code isn't DRY.

So let's look at some typical examples of duplication.

## Duplication in Code

It may be trivial, but code duplication is oh, so common. Here's an example:

```
def print_balance(account)
  printf "Debits:  %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  if account.fees < 0
    printf "Fees:    %10.2f-\n", -account.fees
  else
    printf "Fees:    %10.2f\n", account.fees
  end
  printf "         ———-\n"
  if account.balance < 0
    printf "Balance: %10.2f-\n", -account.balance
  else
    printf "Balance: %10.2f\n", account.balance
  end
end
```

For now ignore the implication that we're committing the newbie mistake of storing currencies in floats. Instead see if you can spot duplications in this code. (We can see at least three things, but you might see more.)

What did you find? Here's our list.

First, there's clearly a copy-and-paste duplication of handling the negative numbers. We can fix that by adding another function:

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_balance(account)
  printf "Debits:  %10.2f\n", account.debits
  printf "Credits: %10.2f\n", account.credits
  printf "Fees:    %s\n",     format_amount(account.fees)
  printf "         ———-\n"
  printf "Balance: %s\n",     format_amount(account.balance)
end
```

Another duplication is the repetition of the field width in all the printf calls. We *could* fix this by introducing a constant and passing it to each call, but why not just use the existing function?

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_balance(account)
  printf "Debits:  %s\n", format_amount(account.debits)
  printf "Credits: %s\n", format_amount(account.credits)
  printf "Fees:    %s\n", format_amount(account.fees)
  printf "         ———-\n"
  printf "Balance: %s\n", format_amount(account.balance)
end
```

Anything more? Well, what if the client asks for an extra space between the labels and the numbers? We'd have to change five lines. Let's remove that duplication:

```
def format_amount(value)
  result = sprintf("%10.2f", value.abs)
  if value < 0
    result + "-"
  else
    result + " "
  end
end

def print_line(label, value)
  printf "%-9s%s\n", label, value
end

def report_line(label, amount)
  print_line(label + ":", format_amount(amount))
end

def print_balance(account)
  report_line("Debits",  account.debits)
  report_line("Credits", account.credits)
  report_line("Fees",    account.fees)
  print_line("",         "———-")
  report_line("Balance", account.balance)
end
```

If we have to change the formatting of amounts, we change format_amount. If we want to change the label format, we change report_line.

There's still an implicit DRY violation: the number of hyphens in the separator line is related to the width of the amount field. But it isn't an exact match: it's currently one character shorter, so any trailing minus signs extend beyond

the column. This is the customer's intent, and it's a different intent to the actual formatting of amounts.

### Not All Code Duplication Is Knowledge Duplication

As part of your online wine ordering application you're capturing and validating your user's age, along with the quantity they're ordering. According to the site owner, they should both be numbers, and both greater than zero. So you code up the validations:

```
def validate_age(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)

def validate_quantity(value):
    validate_type(value, :integer)
    validate_min_integer(value, 0)
```

During code review, the resident know-it-all bounces this code, claiming it's a DRY violation: both function bodies are the same.

They are wrong. The code is the same, but the knowledge they represent is different. The two functions validate two separate things that just happen to have the same rules. That's a coincidence, not a duplication.

## Duplication in Documentation

Somehow the myth was born that you should comment all your functions. Those who believe in this insanity then produce something such as this:

```
# Calculate the fees for this account.
#
# * Each returned check costs $20
# * If the account is in overdraft for more than 3 days,
#   charge $10 for each day
# * If the average account balance is greater that $2,000
#   reduce the fees by 50%

def fees(a)
  f = 0
  if a.returned_check_count > 0
    f += 20 * a.returned_check_count
  end
  if a.overdraft_days > 3
    f += 10*a.overdraft_days
  end
  if a.average_balance > 2_000
    f /= 2
  end
  f
end
```

The intent of this function is given twice: once in the comment and again in the code. The customer changes a fee, and we have to update both. Given time, we can pretty much guarantee the comment and the code will get out of step.

Ask yourself what the comment adds to the code. From our point of view, it simply compensates for some bad naming and layout. How about just this:

```
def calculate_account_fees(account)
  fees  = 20 * account.returned_check_count
  fees += 10 * account.overdraft_days  if account.overdraft_days > 3
  fees /= 2                            if account.average_balance > 2_000
  fees
end
```

The name says what it does, and if someone needs details, they're laid out in the source. That's DRY!

### DRY Violations in Data

Our data structures represent knowledge, and they can fall afoul of the DRY principle. Let's look at a class representing a line:

```
class Line {
  Point  start;
  Point  end;
  double length;
};
```

At first sight, this class might appear reasonable. A line clearly has a start and end, and will always have a length (even if it's zero). But we have duplication. The length is defined by the start and end points: change one of the points and the length changes. It's better to make the length a calculated field:

```
class Line {
  Point  start;
  Point  end;
  double length() { return start.distanceTo(end); }
};
```

Later on in the development process, you may choose to violate the DRY principle for performance reasons. Frequently this occurs when you need to cache data to avoid repeating expensive operations. The trick is to localize the impact. The violation is not exposed to the outside world: only the methods within the class have to worry about keeping things straight:

```
class Line {
  private double length;
  private Point  start;
  private Point  end;

  public Line(Point start, Point end) {
    this.start = start;
    this.end   = end;
    calculateLength();
  }

  // public
  void setStart(Point p) { this.start = p; calculateLength(); }
  void setEnd(Point p)   { this.end   = p; calculateLength(); }

  Point getStart()       { return start; }
  Point getEnd()         { return end;   }

  double getLength()     { return length; }

  private void calculateLength() {
    this.length  = start.distanceTo(end);
  }
};
```

This example also illustrates an important issue: whenever a module exposes a data structure, you're coupling all the code that uses that structure to the implementation of that module. Where possible, always use accessor functions to read and write the attributes of objects. It will make it easier to add functionality in the future.

This use of accessor functions ties in with Meyer's *Uniform Access principle*, described in *Object-Oriented Software Construction [Mey97]*, which states that

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

## Representational Duplication

Your code interfaces to the outside world: other libraries via APIs, other services via remote calls, data in external sources, and so on. And pretty much each time you do, you introduce some kind of DRY violation: your code has to have knowledge that is also present in the external *thing*. It needs to know the API, or the schema, or the meaning of error codes, or whatever. The duplication here is that two things (your code and the external entity) have to have knowledge of the representation of their interface. Change it at one end, and the other end breaks.

This duplication is inevitable, but can be mitigated. Here are some strategies.

### Duplication Across Internal APIs

For internal APIs, look for tools that let you specify the API in some kind of neutral format. These tools will typically generate documentation, mock APIs, functional tests, and API clients, the latter in a number of different languages. Ideally the tool will store all your APIs in a central repository, allowing them to be shared across teams.

### Duplication Across External APIs

Increasingly, you'll find that public APIs are documented formally using something like OpenAPI.[3] This allows you to import the API spec into your local API tools and integrate more reliably with the service.

If you can't find such a specification, consider creating one and publishing it. Not only will others find it useful; you may even get help maintaining it.

### Duplication with Data Sources

Many data sources allow you to introspect on their data schema. This can be used to remove much of the duplication between them and your code. Rather than manually creating the code to contain this stored data, you can generate the containers directly from the schema. Many persistence frameworks will do this heavy lifting for you.

There's another option, and one we often prefer. Rather than writing code that represents external data in a fixed structure (an instance of a struct or class, for example), just stick it into a key/value data structure (your language might call it a map, hash, dictionary, or even object).

On its own this is risky: you lose a lot of the security of knowing just what data you're working with. So we recommend adding a second layer to this solution: a simple table-driven validation suite that verifies that the map you've created contains at least the data you need, in the format you need it. Your API documentation tool might be able to generate this.

## Interdeveloper Duplication

Perhaps the hardest type of duplication to detect and handle occurs between different developers on a project. Entire sets of functionality may be inadvertently duplicated, and that duplication could go undetected for years, leading to maintenance problems. We heard firsthand of a U.S. state whose governmental computer systems were surveyed for Y2K compliance. The audit turned

3. https://github.com/OAI/OpenAPI-Specification

up more than 10,000 programs that each contained a different version of Social Security Number validation code.

At a high level, deal with the problem by building a strong, tight-knit team with good communications.

However, at the module level, the problem is more insidious. Commonly needed functionality or data that doesn't fall into an obvious area of responsibility can get implemented many times over.

We feel that the best way to deal with this is to encourage active and frequent communication between developers.

Maybe run a daily scrum standup meeting. Set up forums (such as Slack channels) to discuss common problems. This provides a nonintrusive way of communicating—even across multiple sites—while retaining a permanent history of everything said.

Appoint a team member as the project librarian, whose job is to facilitate the exchange of knowledge. Have a central place in the source tree where utility routines and scripts can be deposited. And make a point of reading other people's source code and documentation, either informally or during code reviews. You're not snooping—you're learning from them. And remember, the access is reciprocal—don't get twisted about other people poring (pawing?) through *your* code, either.

**Tip 16** Make It Easy to Reuse

What you're trying to do is foster an environment where it's easier to find and reuse existing stuff than to write it yourself. *If it isn't easy, people won't do it.* And if you fail to reuse, you risk duplicating knowledge.

## Related Sections Include

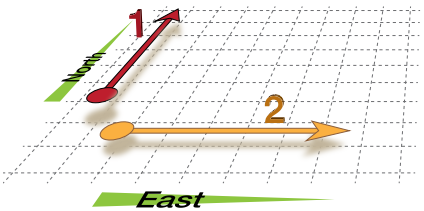

- Topic 8, *The Essence of Good Design*, on page 28
- Topic 28, *Decoupling*, on page 130
- Topic 32, *Configuration*, on page 166
- Topic 38, *Programming by Coincidence*, on page 197
- Topic 40, *Refactoring*, on page 209


# Orthogonality

Orthogonality is a critical concept if you want to produce systems that are easy to design, build, test, and extend. However, the concept of orthogonality is rarely taught directly. Often it is an implicit feature of various other methods and techniques you learn. This is a mistake. Once you learn to apply the principle of orthogonality directly, you'll notice an immediate improvement in the quality of systems you produce.

## What Is Orthogonality?

"Orthogonality" is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are *independent*. As the number 1 on the diagram moves north, it doesn't change how far east or west it is. The number 2 moves east, but not north or south.




In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.

Before we look at the benefits of orthogonal systems, let's first look at a system that isn't orthogonal.

### A Nonorthogonal System

You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch, suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground.

As luck would have it, you had read a Wikipedia page about helicopters the previous night. You know that helicopters have four basic controls. The *cyclic* is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the *collective pitch lever*. Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the *throttle*. Finally you have two *foot pedals*, which vary the amount of tail rotor thrust and so help turn the helicopter.

"Easy!," you think. "Gently lower the collective pitch lever and you'll descend gracefully to the ground, a hero." However, when you try it, you discover that life isn't that simple. The helicopter's nose drops, and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.

Helicopter controls are decidedly not orthogonal.

## Benefits of Orthogonality

As the helicopter example illustrates, nonorthogonal systems are inherently more complex to change and control. When components of any system are highly interdependent, there is no such thing as a local fix.

**Tip 17** Eliminate Effects Between Unrelated Things

We want to design components that are self-contained: independent, and with a single, well-defined purpose (what Yourdon and Constantine call *cohesion* in *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design [YC79]*). When components are isolated from one another, you know that you can change one without having to worry about the rest. As long as you don't change that component's external interfaces, you can be confident that you won't cause problems that ripple through the entire system.

You get two major benefits if you write orthogonal systems: increased productivity and reduced risk.

### Gain Productivity

- Changes are localized, so development time and testing time are reduced. It is easier to write relatively small, self-contained components than a single large block of code. Simple components can be designed, coded, tested, and then forgotten—there is no need to keep changing existing code as you add new code.

- An orthogonal approach also promotes reuse. If components have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned by their original implementors. The more loosely coupled your systems, the easier they are to reconfigure and reengineer.
- There is a fairly subtle gain in productivity when you combine orthogonal components. Assume that one component does $M$ distinct things and another does $N$ things. If they are orthogonal and you combine them, the result does $M \times N$ things. However, if the two components are not orthogonal, there will be overlap, and the result will do less. You get more functionality per unit effort by combining orthogonal components.

### Reduce Risk

An orthogonal approach reduces the risks inherent in any development.

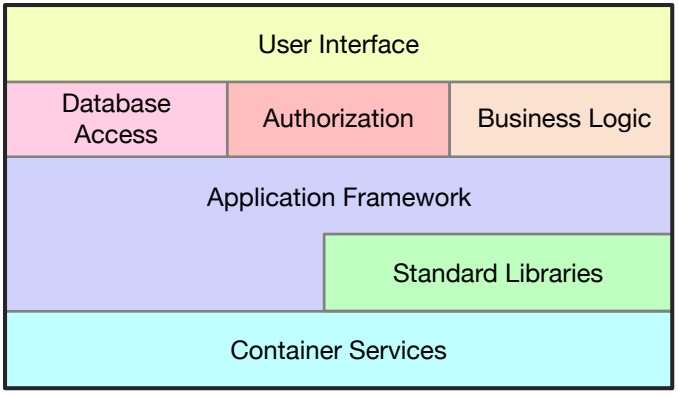
- Diseased sections of code are isolated. If a module is sick, it is less likely to spread the symptoms around the rest of the system. It is also easier to slice it out and transplant in something new and healthy.
- The resulting system is less fragile. Make small changes and fixes to a particular area, and any problems you generate will be restricted to that area.
- An orthogonal system will probably be better tested, because it will be easier to design and run tests on its components.
- You will not be as tightly tied to a particular vendor, product, or platform, because the interfaces to these third-party components will be isolated to smaller parts of the overall development.

Let's look at some of the ways you can apply the principle of orthogonality to your work.

## Design

Most developers are familiar with the need to design orthogonal systems, although they may use words such as *modular*, *component-based*, and *layered* to describe the process. Systems should be composed of a set of cooperating modules, each of which implements functionality independent of the others. Sometimes these components are organized into layers, each providing a level of abstraction. This layered approach is a powerful way to design orthogonal systems. Because each layer uses only the abstractions provided by the layers below it, you have great flexibility in changing underlying

implementations without affecting code. Layering also reduces the risk of runaway dependencies between modules. You'll often see layering expressed in diagrams:




There is an easy test for orthogonal design. Once you have your components mapped out, ask yourself: *If I dramatically change the requirements behind a particular function, how many modules are affected?* In an orthogonal system, the answer should be "one."[4] Moving a button on a GUI panel should not require a change in the database schema. Adding context-sensitive help should not change the billing subsystem.

Let's consider a complex system for monitoring and controlling a heating plant. The original requirement called for a graphical user interface, but the requirements were changed to add a mobile interface that lets engineers monitor key values. In an orthogonally designed system, you would need to change only those modules associated with the user interface to handle this: the underlying logic of controlling the plant would remain unchanged. In fact, if you structure your system carefully, you should be able to support both interfaces with the same underlying code base.

Also ask yourself how decoupled your design is from changes in the real world. Are you using a telephone number as a customer identifier? What happens when the phone company reassigns area codes? Postal codes, Social Security Numbers or government IDs, email addresses, and domains are all external identifiers that you have no control over, and could change at any time for any reason. *Don't rely on the properties of things you can't control.*

4. In reality, this is naive. Unless you are remarkably lucky, most real-world requirements changes will affect multiple functions in the system. However, if you analyze the change in terms of functions, each functional change should still ideally affect just one module.

## Toolkits and Libraries

Be careful to preserve the orthogonality of your system as you introduce third-party toolkits and libraries. Choose your technologies wisely.

When you bring in a toolkit (or even a library from other members of your team), ask yourself whether it imposes changes on your code that shouldn't be there. If an object persistence scheme is transparent, then it's orthogonal. If it requires you to create or access objects in a special way, then it's not. Keeping such details isolated from your code has the added benefit of making it easier to change vendors in the future.

The Enterprise Java Beans (EJB) system is an interesting example of orthogonality. In most transaction-oriented systems, the application code has to delineate the start and end of each transaction. With EJB, this information is expressed declaratively as annotations, outside the methods that do the work. The same application code can run in different EJB transaction environments with no change.

In a way, EJB is an example of the Decorator Pattern: adding functionality to things without changing them. This style of programming can be used in just about every programming language, and doesn't necessarily require a framework or library. It just takes a little discipline when programming.

## Coding

Every time you write code you run the risk of reducing the orthogonality of your application. Unless you constantly monitor not just what you are doing but also the larger context of the application, you might unintentionally duplicate functionality in some other module, or express existing knowledge twice.

There are several techniques you can use to maintain orthogonality:

*Keep your code decoupled*

Write shy code—modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations. Try the Law of Demeter, which we discuss in Topic 28, *Decoupling*, on page 130. If you need to change an object's state, get the object to do it for you. This way your code remains isolated from the other code's implementation and increases the chances that you'll remain orthogonal.

*Avoid global data*

Every time your code references global data, it ties itself into the other components that share that data. Even globals that you intend only to

read can lead to trouble (for example, if you suddenly need to change your code to be multithreaded). In general, your code is easier to understand and maintain if you explicitly pass any required context into your modules. In object-oriented applications, context is often passed as parameters to objects' constructors. In other code, you can create structures containing the context and pass around references to them.

The Singleton pattern in *Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]* is a way of ensuring that there is only one instance of an object of a particular class. Many people use these singleton objects as a kind of global variable (particularly in languages, such as Java, that otherwise do not support the concept of globals). Be careful with singletons—they can also lead to unnecessary linkage.

*Avoid similar functions*

Often you'll come across a set of functions that all look similar—maybe they share common code at the start and end, but each has a different central algorithm. Duplicate code is a symptom of structural problems. Have a look at the Strategy pattern in *Design Patterns* for a better implementation.

Get into the habit of being constantly critical of your code. Look for any opportunities to reorganize it to improve its structure and orthogonality. This process is called *refactoring*, and it's so important that we've dedicated a section to it (see Topic 40, *Refactoring*, on page 209).

## Testing

An orthogonally designed and implemented system is easier to test. Because the interactions between the system's components are formalized and limited, more of the system testing can be performed at the individual module level. This is good news, because module level (or unit) testing is considerably easier to specify and perform than integration testing. In fact, we suggest that these tests be performed automatically as part of the regular build process (see Topic 41, *Test to Code*, on page 214).

Writing unit tests is itself an interesting test of orthogonality. What does it take to get a unit test to build and run? Do you have to import a large percentage of the rest of the system's code? If so, you've found a module that is not well decoupled from the rest of the system.

Bug fixing is also a good time to assess the orthogonality of the system as a whole. When you come across a problem, assess how localized the fix is. Do you change just one module, or are the changes scattered throughout the

entire system? When you make a change, does it fix everything, or do other problems mysteriously arise? This is a good opportunity to bring automation to bear. If you use a version control system (and you will after reading Topic 19, *Version Control*, on page 84), tag bug fixes when you check the code back in after testing. You can then run monthly reports analyzing trends in the number of source files affected by each bug fix.

## Documentation

Perhaps surprisingly, orthogonality also applies to documentation. The axes are content and presentation. With truly orthogonal documentation, you should be able to change the appearance dramatically without changing the content. Word processors provide style sheets and macros that help. We personally prefer using a markup system such as Markdown: when writing we focus only on the content, and leave the presentation to whichever tool we use to render it.[5]

## Living with Orthogonality

Orthogonality is closely related to the DRY principle on page 30. With DRY, you're looking to minimize duplication within a system, whereas with orthogonality you reduce the interdependency among the system's components. It may be a clumsy word, but if you use the principle of orthogonality, combined closely with the DRY principle, you'll find that the systems you develop are more flexible, more understandable, and easier to debug, test, and maintain.

If you're brought into a project where people are desperately struggling to make changes, and where every change seems to cause four other things to go wrong, remember the nightmare with the helicopter. The project probably is not orthogonally designed and coded. It's time to refactor.

And, if you're a helicopter pilot, don't eat the fish....

## Related Sections Include


- Topic 3, *Software Entropy*, on page 6
- Topic 8, *The Essence of Good Design*, on page 28
- Topic 11, *Reversibility*, on page 47
- Topic 28, *Decoupling*, on page 130
- Topic 31, *Inheritance Tax*, on page 158


5. In fact, this book is written in Markdown, and typeset directly from the Markdown source.

- Topic 33, *Breaking Temporal Coupling*, on page 170
- Topic 34, *Shared State Is Incorrect State*, on page 174
- Topic 36, *Blackboards*, on page 187

### Challenges

- Consider the difference between tools which have a graphical user interface and small but combinable command-line utilities used at shell prompts. Which set is more orthogonal, and why? Which is easier to use for exactly the purpose for which it was intended? Which set is easier to combine with other tools to meet new challenges? Which set is easier to learn?
- C++ supports multiple inheritance, and Java allows a class to implement multiple interfaces. Ruby has mixins. What impact does using these facilities have on orthogonality? Is there a difference in impact between using multiple inheritance and multiple interfaces? Is there a difference between using delegation and using inheritance?

### Exercises

**Exercise 1** (possible answer on page 293)

You're asked to read a file a line at a time. For each line, you have to split it into fields. Which of the following sets of pseudo class definitions is likely to be more orthogonal?

```
class Split1 {
  constructor(fileName)    # opens the file for reading
  def readNextLine()       # moves to the next line
  def getField(n)          # returns nth field in current line
}
```

or

```
class Split2 {
  constructor(line)        # splits a line
  def getField(n)          # returns nth field in current line
}
```

**Exercise 2** (possible answer on page 293)

What are the differences in orthogonality between object-oriented and functional languages? Are these differences inherent in the languages themselves, or just in the way people use them?


# Reversibility

*Nothing is more dangerous than an idea if it's the only one you have.*

➤ *Emil-Auguste Chartier (Alain), Propos sur la religion, 1938*

Engineers prefer simple, singular solutions to problems. Math tests that allow you to proclaim with great confidence that $x = 2$ are much more comfortable than fuzzy, warm essays about the myriad causes of the French Revolution. Management tends to agree with the engineers: singular, easy answers fit nicely on spreadsheets and project plans.

If only the real world would cooperate! Unfortunately, while $x$ is 2 today, it may need to be 5 tomorrow, and 3 next week. Nothing is forever—and if you rely heavily on some fact, you can almost guarantee that it *will* change.

There is always more than one way to implement something, and there is usually more than one vendor available to provide a third-party product. If you go into a project hampered by the myopic notion that there is only *one* way to do it, you may be in for an unpleasant surprise. Many project teams have their eyes forcibly opened as the future unfolds:

> "But you said we'd use database XYZ! We are 85% done coding the project, we can't change now!" the programmer protested. "Sorry, but our company decided to standardize on database PDQ instead—for all projects. It's out of my hands. We'll just have to recode. All of you will be working weekends until further notice."

Changes don't have to be that Draconian, or even that immediate. But as time goes by, and your project progresses, you may find yourself stuck in an untenable position. With every critical decision, the project team commits to a smaller target—a narrower version of reality that has fewer options.

By the time many critical decisions have been made, the target becomes so small that if it moves, or the wind changes direction, or a butterfly in Tokyo flaps its wings, you miss.[6] And you may miss by a huge amount.

The problem is that critical decisions aren't easily reversible.

6. Take a nonlinear, or chaotic, system and apply a small change to one of its inputs. You may get a large and often unpredictable result. The clichéd butterfly flapping its wings in Tokyo could be the start of a chain of events that ends up generating a tornado in Texas. Does this sound like any projects you know?

Once you decide to use this vendor's database, or that architectural pattern, or a certain deployment model, you are committed to a course of action that cannot be undone, except at great expense.

## Reversibility

Many of the topics in this book are geared to producing flexible, adaptable software. By sticking to their recommendations—especially the DRY principle on page 30, decoupling on page 130, and use of external configuration on page 166—we don't have to make as many critical, irreversible decisions. This is a good thing, because we don't always make the best decisions the first time around. We commit to a certain technology only to discover we can't hire enough people with the necessary skills. We lock in a certain third-party vendor just before they get bought out by their competitor. Requirements, users, and hardware change faster than we can get the software developed.

Suppose you decide, early in the project, to use a relational database from vendor A. Much later, during performance testing, you discover that the database is simply too slow, but that the document database from vendor B is faster. With most conventional projects, you'd be out of luck. Most of the time, calls to third-party products are entangled throughout the code. But if you *really* abstracted the idea of a database out—to the point where it simply provides persistence as a service—then you have the flexibility to change horses in midstream.

Similarly, suppose the project begins as a browser-based application, but then, late in the game, marketing decides that what they really want is a mobile app. How hard would that be for you? In an ideal world, it shouldn't impact you too much, at least on the server side. You'd be stripping out some HTML rendering and replacing it with an API.

The mistake lies in assuming that any decision is cast in stone—and in not preparing for the contingencies that might arise. Instead of carving decisions in stone, think of them more as being written in the sand at the beach. A big wave can come along and wipe them out at any time.

**Tip 18** There Are No Final Decisions

## Flexible Architecture

While many people try to keep their *code* flexible, you also need to think about maintaining flexibility in the areas of architecture, deployment, and vendor integration.

We're writing this in 2019. Since the turn of the century we've seen the following "best practice" server-side architectures:

- Big hunk of iron
- Federations of big iron
- Load-balanced clusters of commodity hardware
- Cloud-based virtual machines running applications
- Cloud-based virtual machines running services
- Containerized versions of the above
- Cloud-supported serverless applications
- And, inevitably, an apparent move back to big hunks of iron for some tasks

Go ahead and add the very latest and greatest fads to this list, and then regard it with awe: it's a miracle that anything ever worked.

How can you plan for this kind of architectural volatility? You can't.

What you can do is make it easy to change. Hide third-party APIs behind your own abstraction layers. Break your code into components: even if you end up deploying them on a single massive server, this approach is a lot easier than taking a monolithic application and splitting it. (We have the scars to prove it.)

And, although this isn't particularly a reversibility issue, one final piece of advice.

**Tip 19** Forgo Following Fads

No one knows what the future may hold, especially not us! So enable your code to rock-n-roll: to "rock on" when it can, to roll with the punches when it must.

## Related Sections Include


- Topic 8, *The Essence of Good Design*, on page 28
- Topic 10, *Orthogonality*, on page 39
- Topic 19, *Version Control*, on page 84
- Topic 28, *Decoupling*, on page 130
- Topic 45, *The Requirements Pit*, on page 244
- Topic 51, *Pragmatic Starter Kit*, on page 273

### Challenges

- Time for a little quantum mechanics with Schrödinger's cat.

  Suppose you have a cat in a closed box, along with a radioactive particle. The particle has exactly a 50% chance of fissioning into two. If it does, the cat will be killed. If it doesn't, the cat will be okay. So, is the cat dead or alive? According to Schrödinger, the correct answer is *both* (at least while the box remains closed). Every time a subnuclear reaction takes place that has two possible outcomes, the universe is cloned. In one, the event occurred, in the other it didn't. The cat's alive in one universe, dead in another. Only when you open the box do you know which universe *you* are in.

  No wonder coding for the future is difficult.

  But think of code evolution along the same lines as a box full of Schrödinger's cats: every decision results in a different version of the future. How many possible futures can your code support? Which ones are more likely? How hard will it be to support them when the time comes?

  Dare you open the box?

---

12

## Tracer Bullets

*Ready, fire, aim...*

➤ *Anon*

We often talk about hitting targets when we develop software. We're not actually firing anything at the shooting range, but it's still a useful and very visual metaphor. In particular, it's interesting to consider *how* to hit a target in a complex and shifting world.

The answer, of course, depends on the nature of the device you're aiming with. With many you only get one chance to aim, and then get to see if you hit the bullseye or not. But there's a better way.

You know all those movies, TV shows, and video games where people are shooting machine guns? In these scenes, you'll often see the path of bullets as bright streaks in the air. These streaks come from tracer bullets.

Tracer bullets are loaded at intervals alongside regular ammunition. When they're fired, their phosphorus ignites and leaves a pyrotechnic trail from the gun to whatever they hit. If the tracers are hitting the target, then so are the

regular bullets. Soldiers use these tracer rounds to refine their aim: it's pragmatic, real-time feedback under actual conditions.

That same principle applies to projects, particularly when you're building something that hasn't been built before. We use the term *tracer bullet development* to visually illustrate the need for immediate feedback under actual conditions with a moving goal.

Like the gunners, you're trying to hit a target in the dark. Because your users have never seen a system like this before, their requirements may be vague. Because you may be using algorithms, techniques, languages, or libraries you aren't familiar with, you face a large number of unknowns. And because projects take time to complete, you can pretty much guarantee the environment you're working in will change before you're done.

The classic response is to specify the system to death. Produce reams of paper itemizing every requirement, tying down every unknown, and constraining the environment. Fire the gun using dead reckoning. One big calculation up front, then shoot and hope.

Pragmatic Programmers, however, tend to prefer using the software equivalent of tracer bullets.

## Code That Glows in the Dark

Tracer bullets work because they operate in the same environment and under the same constraints as the real bullets. They get to the target fast, so the gunner gets immediate feedback. And from a practical standpoint they're a relatively cheap solution.

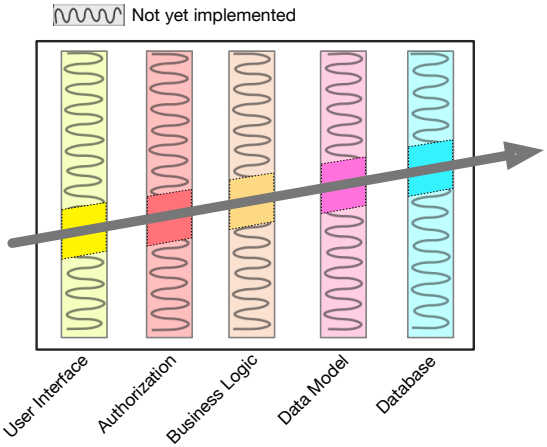
To get the same effect in code, we look for something that gets us from a requirement to some aspect of the final system quickly, visibly, and repeatably.

Look for the important requirements, the ones that define the system. Look for the areas where you have doubts, and where you see the biggest risks. Then prioritize your development so that these are the first areas you code.

**Tip 20** Use Tracer Bullets to Find the Target

In fact, given the complexity of today's project setup, with swarms of external dependencies and tools, tracer bullets become even more important. For us, the very first tracer bullet is simply *create the project, add a "hello world!," and make sure it compiles and runs.* Then we look for areas of uncertainty in the overall application and add the skeleton needed to make it work.

Have a look at the following diagram. This system has five architectural layers. We have some concerns about how they'd integrate, so we look for a simple feature that lets us exercise them together. The diagonal line shows the path that feature takes through the code. To make it work, we just have to implement the solidly shaded areas in each layer: the stuff with the squiggles will be done later.




We once undertook a complex client-server database marketing project. Part of its requirement was the ability to specify and execute temporal queries. The servers were a range of relational and specialized databases. The client UI, written in random language A, used a set of libraries written in a different language to provide an interface to the servers. The user's query was stored on the server in a Lisp-like notation before being converted to optimized SQL just prior to execution. There were many unknowns and many different environments, and no one was too sure how the UI should behave.

This was a great opportunity to use tracer code. We developed the framework for the front end, libraries for representing the queries, and a structure for converting a stored query into a database-specific query. Then we put it all together and checked that it worked. For that initial build, all we could do was submit a query that listed all the rows in a table, but it proved that the UI could talk to the libraries, the libraries could serialize and unserialize a query, and the server could generate SQL from the result. Over the following months we gradually fleshed out this basic structure, adding new functionality by augmenting each component of the tracer code in parallel. When the UI added a new query type, the library grew and the SQL generation was made more sophisticated.

Tracer code is not disposable: you write it for keeps. It contains all the error checking, structuring, documentation, and self-checking that any piece of production code has. It simply is not fully functional. However, once you have achieved an end-to-end connection among the components of your system, you can check how close to the target you are, adjusting if necessary. Once you're on target, adding functionality is easy.

Tracer development is consistent with the idea that a project is never finished: there will always be changes required and functions to add. It is an incremental approach.

The conventional alternative is a kind of heavy engineering approach: code is divided into modules, which are coded in a vacuum. Modules are combined into subassemblies, which are then further combined, until one day you have a complete application. Only then can the application as a whole be presented to the user and tested.

The tracer code approach has many advantages:

*Users get to see something working early*

If you have successfully communicated what you are doing (see Topic 52, *Delight Your Users*, on page 280), your users will know they are seeing something immature. They won't be disappointed by a lack of functionality; they'll be ecstatic to see some visible progress toward their system. They also get to contribute as the project progresses, increasing their buy-in. These same users will likely be the people who'll tell you how close to the target each iteration is.

*Developers build a structure to work in*

The most daunting piece of paper is the one with nothing written on it. If you have worked out all the end-to-end interactions of your application, and have embodied them in code, then your team won't need to pull as much out of thin air. This makes everyone more productive, and encourages consistency.

*You have an integration platform*

As the system is connected end-to-end, you have an environment to which you can add new pieces of code once they have been unit-tested. Rather than attempting a big-bang integration, you'll be integrating every day (often many times a day). The impact of each new change is more apparent, and the interactions are more limited, so debugging and testing are faster and more accurate.

*You have something to demonstrate*

Project sponsors and top brass have a tendency to want to see demos at the most inconvenient times. With tracer code, you'll always have something to show them.

*You have a better feel for progress*

In a tracer code development, developers tackle use cases one by one. When one is done, they move to the next. It is far easier to measure performance and to demonstrate progress to your user. Because each individual development is smaller, you avoid creating those monolithic blocks of code that are reported as 95% complete week after week.

## Tracer Bullets Don't Always Hit Their Target

Tracer bullets show what you're hitting. This may not always be the target. You then adjust your aim until they're on target. That's the point.

It's the same with tracer code. You use the technique in situations where you're not 100% certain of where you're going. You shouldn't be surprised if your first couple of attempts miss: the user says "that's not what I meant," or data you need isn't available when you need it, or performance problems seem likely. So change what you've got to bring it nearer the target, and be thankful that you've used a lean development methodology; a small body of code has low inertia—it is easy and quick to change. You'll be able to gather feedback on your application and generate a new, more accurate version quickly and cheaply. And because every major application component is represented in your tracer code, your users can be confident that what they're seeing is based on reality, not just a paper specification.

## Tracer Code versus Prototyping

You might think that this tracer code concept is nothing more than prototyping under an aggressive name. There is a difference. With a prototype, you're aiming to explore specific aspects of the final system. With a true prototype, you will throw away whatever you lashed together when trying out the concept, and recode it properly using the lessons you've learned.

For example, say you're producing an application that helps shippers determine how to pack odd-sized boxes into containers. Among other problems, the user interface needs to be intuitive and the algorithms you use to determine optimal packing are very complex.

You could prototype a user interface for your end users in a UI tool. You code only enough to make the interface responsive to user actions. Once they've

agreed to the layout, you might throw it away and recode it, this time with the business logic behind it, using the target language. Similarly, you might want to prototype a number of algorithms that perform the actual packing. You might code functional tests in a high-level, forgiving language such as Python, and code low-level performance tests in something closer to the machine. In any case, once you'd made your decision, you'd start again and code the algorithms in their final environment, interfacing to the real world. This is *prototyping*, and it is very useful.

The tracer code approach addresses a different problem. You need to know how the application as a whole hangs together. You want to show your users how the interactions will work in practice, and you want to give your developers an architectural skeleton on which to hang code. In this case, you might construct a tracer consisting of a trivial implementation of the container packing algorithm (maybe something like first-come, first-served) and a simple but working user interface. Once you have all the components in the application plumbed together, you have a framework to show your users and your developers. Over time, you add to this framework with new functionality, completing stubbed routines. But the framework stays intact, and you know the system will continue to behave the way it did when your first tracer code was completed.

The distinction is important enough to warrant repeating. Prototyping generates disposable code. Tracer code is lean but complete, and forms part of the skeleton of the final system. Think of prototyping as the reconnaissance and intelligence gathering that takes place before a single tracer bullet is fired.

## Related Sections Include


- Topic 13, *Prototypes and Post-it Notes*, on page 56
- Topic 27, *Don't Outrun Your Headlights*, on page 125
- Topic 40, *Refactoring*, on page 209
- Topic 49, *Pragmatic Teams*, on page 264
- Topic 50, *Coconuts Don't Cut It*, on page 270
- Topic 51, *Pragmatic Starter Kit*, on page 273
- Topic 52, *Delight Your Users*, on page 280

13

## Prototypes and Post-it Notes

Many industries use prototypes to try out specific ideas; prototyping is much cheaper than full-scale production. Car makers, for example, may build many different prototypes of a new car design. Each one is designed to test a specific aspect of the car—the aerodynamics, styling, structural characteristics, and so on. Old school folks might use a clay model for wind tunnel testing, maybe a balsa wood and duct tape model will do for the art department, and so on. The less romantic will do their modeling on a computer screen or in virtual reality, reducing costs even further. In this way, risky or uncertain elements can be tried out without committing to building the real item.

We build software prototypes in the same fashion, and for the same reasons—to analyze and expose risk, and to offer chances for correction at a greatly reduced cost. Like the car makers, we can target a prototype to test one or more specific aspects of a project.

We tend to think of prototypes as code-based, but they don't always have to be. Like the car makers, we can build prototypes out of different materials. Post-it notes are great for prototyping dynamic things such as workflow and application logic. A user interface can be prototyped as a drawing on a whiteboard, as a nonfunctional mock-up drawn with a paint program, or with an interface builder.

Prototypes are designed to answer just a few questions, so they are much cheaper and faster to develop than applications that go into production. The code can ignore unimportant details—unimportant to you at the moment, but probably very important to the user later on. If you are prototyping a UI, for instance, you can get away with incorrect results or data. On the other hand, if you're just investigating computational or performance aspects, you can get away with a pretty poor UI, or perhaps even no UI at all.

But if you find yourself in an environment where you *cannot* give up the details, then you need to ask yourself if you are really building a prototype at all. Perhaps a tracer bullet style of development would be more appropriate in this case (see Topic 12, *Tracer Bullets*, on page 50).

### Things to Prototype

What sorts of things might you choose to investigate with a prototype? Anything that carries risk. Anything that hasn't been tried before, or that is

absolutely critical to the final system. Anything unproven, experimental, or doubtful. Anything you aren't comfortable with. You can prototype:

- Architecture
- New functionality in an existing system
- Structure or contents of external data
- Third-party tools or components
- Performance issues
- User interface design

Prototyping is a learning experience. Its value lies not in the code produced, but in the lessons learned. That's really the point of prototyping.

**Tip 21** Prototype to Learn

## How to Use Prototypes

When building a prototype, what details can you ignore?

*Correctness*
: You may be able to use dummy data where appropriate.

*Completeness*
: The prototype may function only in a very limited sense, perhaps with only one preselected piece of input data and one menu item.

*Robustness*
: Error checking is likely to be incomplete or missing entirely. If you stray from the predefined path, the prototype may crash and burn in a glorious display of pyrotechnics. That's okay.

*Style*
: Prototype code shouldn't have much in the way of comments or documentation (although you may produce reams of documentation as a result of your experience with the prototype).

Prototypes gloss over details, and focus in on specific aspects of the system being considered, so you may want to implement them using a high-level scripting language—higher than the rest of the project (maybe a language such as Python or Ruby), as these languages can get out of your way. You may choose to continue to develop in the language used for the prototype, or you can switch; after all, you're going to throw the prototype away anyway.

To prototype user interfaces, use a tool that lets you focus on the appearance and/or interactions without worrying about code or markup.

Scripting languages also work well as the "glue" to combine low-level pieces into new combinations. Using this approach, you can rapidly assemble existing components into new configurations to see how things work.

## Prototyping Architecture

Many prototypes are constructed to model the entire system under consideration. As opposed to tracer bullets, none of the individual modules in the prototype system need to be particularly functional. In fact, you may not even need to code in order to prototype architecture—you can prototype on a whiteboard, with Post-it notes or index cards. What you are looking for is how the system hangs together as a whole, again deferring details. Here are some specific areas you may want to look for in the architectural prototype:

- Are the responsibilities of the major areas well defined and appropriate?
- Are the collaborations between major components well defined?
- Is coupling minimized?
- Can you identify potential sources of duplication?
- Are interface definitions and constraints acceptable?
- Does every module have an access path to the data it needs during execution? Does it have that access *when* it needs it?

This last item tends to generate the most surprises and the most valuable results from the prototyping experience.

## How *Not* to Use Prototypes

Before you embark on any code-based prototyping, make sure that everyone understands that you are writing disposable code. Prototypes can be deceptively attractive to people who don't know that they are just prototypes. You must make it *very* clear that this code is disposable, incomplete, and unable to be completed.

It's easy to become misled by the apparent completeness of a demonstrated prototype, and project sponsors or management may insist on deploying the prototype (or its progeny) if you don't set the right expectations. Remind them that you can build a great prototype of a new car out of balsa wood and duct tape, but you wouldn't try to drive it in rush-hour traffic!

If you feel there is a strong possibility in your environment or culture that the purpose of prototype code may be misinterpreted, you may be better off with the tracer bullet approach. You'll end up with a solid framework on which to base future development.

Properly used prototypes can save you huge amounts of time, money, and pain by identifying and correcting potential problem spots early in the development cycle—the time when fixing mistakes is both cheap and easy.

### Related Sections Include

- Topic 12, *Tracer Bullets*, on page 50
- Topic 14, *Domain Languages*, on page 59
- Topic 17, *Shell Games*, on page 78
- Topic 27, *Don't Outrun Your Headlights*, on page 125
- Topic 37, *Listen to Your Lizard Brain*, on page 192
- Topic 45, *The Requirements Pit*, on page 244
- Topic 52, *Delight Your Users*, on page 280

### Exercises

**Exercise 3** (possible answer on page 294)

Marketing would like to sit down and brainstorm a few web page designs with you. They are thinking of clickable image maps to take you to other pages, and so on. But they can't decide on a model for the image—maybe it's a car, or a phone, or a house. You have a list of target pages and content; they'd like to see a few prototypes. Oh, by the way, you have 15 minutes. What tools might you use?

---


## Domain Languages

*The limits of language are the limits of one's world.*

➤ *Ludwig Wittgenstein*

Computer languages influence *how* you think about a problem, and how you think about communicating. Every language comes with a list of features: buzzwords such as static versus dynamic typing, early versus late binding, functional versus OO, inheritance models, mixins, macros—all of which may suggest or obscure certain solutions. Designing a solution with C++ in mind will produce different results than a solution based on Haskell-style thinking, and vice versa. Conversely, and we think more importantly, the language of the problem domain may also suggest a programming solution.

We always try to write code using the vocabulary of the application domain (see *Maintain a Glossary*, on page 251). In some cases, Pragmatic Programmers can go to the next level and actually program using the vocabulary, syntax, and semantics—the language—of the domain.

Tip 22 Program Close to the Problem Domain

## Some Real-World Domain Languages

Let's look at a few examples where folks have done just that.

### RSpec

RSpec[7] is a testing library for Ruby. It inspired versions for most other modern languages. A test in RSpec is intended to reflect the behavior you expect from your code.

```
describe BowlingScore do
  it "totals 12 if you score 3 four times" do
    score = BowlingScore.new
    4.times { score.add_pins(3) }
    expect(score.total).to eq(12)
  end
end
```

### Cucumber

Cucumber[8] is programming-language neutral way of specifying tests. You run the tests using a version of Cucumber appropriate to the language you're using. In order to support the natural-language like syntax, you also have to write specific matchers that recognize phrases and extract parameters for the tests.

```
Feature: Scoring

Background:
  Given an empty scorecard

Scenario: bowling a lot of 3s
  Given I throw a 3
  And I throw a 3
  And I throw a 3
  And I throw a 3
  Then the score should be 12
```

Cucumber tests were intended to be read by the customers of the software (although that happens fairly rarely in practice; the following aside considers why that might be).

7. https://rspec.info
8. https://cucumber.io/

### Why Don't Many Business Users Read Cucumber Features?

One of the reasons that the classic *gather requirements, design, code, ship* approach doesn't work is that it is anchored by the concept that we know what the requirements are. But we rarely do. Your business users will have a vague idea of what they want to achieve, but they neither know nor care about the details. That's part of our value: we intuit intent and convert it to code.

So when you force a business person to sign off on a requirements document, or get them to agree to a set of Cucumber features, you're doing the equivalent of getting them to check the spelling in an essay written in Sumerian. They'll make some random changes to save face and sign it off to get you out of their office.

Give them code that runs, however, and they can play with it. That's where their real needs will surface.

### Phoenix Routes

Many web frameworks have a routing facility, mapping incoming HTTP requests onto handler functions in the code. Here's an example from Phoenix.[9]

```
scope "/", HelloPhoenix do
  pipe_through :browser # Use the default browser stack

  get "/", PageController, :index
  resources "/users", UserController
end
```

This says that requests starting "/" will be run through a series of filters appropriate for browsers. A request to "/" itself will be handled by the index function in the PageController module. The UsersController implements the functions needed to manage a resource accessible via the url /users.

### Ansible

Ansible[10] is a tool that configures software, typically on a bunch of remote servers. It does this by reading a specification that you provide, then doing whatever is needed on the servers to make them mirror that spec. The specification can be written in YAML,[11] a language that builds data structures from text descriptions:

9. https://phoenixframework.org/
10. https://www.ansible.com/
11. https://yaml.org/

```
---
- name: install nginx
  apt: name=nginx state=latest

- name: ensure nginx is running (and enable it at boot)
  service: name=nginx state=started enabled=yes

- name: write the nginx config file
  template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf
  notify:
  - restart nginx
```

This example ensures that the latest version of nginx is installed on my servers, that it is started by default, and that it uses a configuration file that you've provided.

## Characteristics of Domain Languages

Let's look at these examples more closely.

RSpec and the Phoenix router are written in their host languages (Ruby and Elixir). They employ some fairly devious code, including metaprogramming and macros, but ultimately they are compiled and run as regular code.

Cucumber tests and Ansible configurations are written in their own languages. A Cucumber test is converted into code to be run or into a datastructure, whereas Ansible specs are always converted into a data structure that is run by Ansible itself.

As a result, RSpec and the router code are embedded into the code you run: they are true extensions to your code's vocabulary. Cucumber and Ansible are *read* by code and converted into some form the code can use.

We call RSpec and the router examples of *internal* domain languages, while Cucumber and Ansible use *external* languages.

## Trade-Offs Between Internal and External Languages

In general, an internal domain language can take advantage of the features of its host language: the domain language you create is more powerful, and that power comes for free. For example, you could use some Ruby code to create a bunch of RSpec tests automatically. In this case we can test scores where there are no spares or strikes:

```
describe BowlingScore do
  (0..4).each do |pins|
    (1..20).each do |throws|
      target = pins * throws

      it "totals #{target} if you score #{pins} #{throws} times" do
        score = BowlingScore.new
        throws.times { score.add_pins(pins) }
        expect(score.total).to eq(target)
      end
    end
  end
end
```

That's 100 tests you just wrote. Take the rest of the day off.

The downside of internal domain languages is that you're bound by the syntax and semantics of that language. Although some languages are remarkably flexible in this regards, you're still forced to compromise between the language you want and the language you can implement.

Ultimately, whatever you come up with must still be valid syntax in your target language. Languages with macros (such as Elixir, Clojure, and Crystal) gives you a little more flexibility, but ultimately syntax is syntax.

External languages have no such restrictions. As long as you can write a parser for the language, you're good to go. Sometimes you can use someone else's parser (as Ansible did by using YAML), but then you're back to making a compromise.

Writing a parser probably means adding new libraries and possibly tools to your application. And writing a good parser is not a trivial job. But, if you're feeling stout of heart, you could look at parser generators such as bison or ANTLR, and parsing frameworks such as the many PEG parsers out there.

Our suggestion is fairly simple: don't spend more effort than you save. Writing a domain language adds some cost to your project, and you'll need to be convinced that there are offsetting savings (potentially in the long term).

In general, use off-the-shelf external languages (such as YAML, JSON, or CSV) if you can. If not, look at internal languages. We'd recommend using external languages only in cases where your language will be written by the users of your application.

## An Internal Domain Language on the Cheap

Finally, there's a cheat for creating internal domain languages if you don't mind the host language syntax leaking through. Don't do a bunch of

metaprogramming. Instead, just write functions to do the work. In fact, this is pretty much what RSpec does:

```
describe BowlingScore do
  it "totals 12 if you score 3 four times" do
    score = BowlingScore.new
    4.times { score.add_pins(3) }
    expect(score.total).to eq(12)
  end
end
```

In this code, describe, it, expect, to, and eq are just Ruby methods. There's a little plumbing behind the scenes in terms of how objects are passed around, but it's all just code. We'll explore that a little in the exercises.

## Related Sections Include

- Topic 8, *The Essence of Good Design*, on page 28
- Topic 13, *Prototypes and Post-it Notes*, on page 56
- Topic 32, *Configuration*, on page 166

## Challenges

- Could some of the requirements of your current project be expressed in a domain-specific language? Would it be possible to write a compiler or translator that could generate most of the code required?
- If you decide to adopt mini-languages as a way of programming closer to the problem domain, you're accepting that some effort will be required to implement them. Can you see ways in which the framework you develop for one project can be reused in others?

## Exercises

**Exercise 4** (possible answer on page 294)

We want to implement a mini-language to control a simple turtle-graphics system. The language consists of single-letter commands, some followed by a single number. For example, the following input would draw a rectangle:

```
P 2  # select pen 2
D    # pen down
W 2  # draw west 2cm
N 1  # then north 1
E 2  # then east 2
S 1  # then back south
U    # pen up
```

Implement the code that parses this language. It should be designed so that it is simple to add new commands.

**Exercise 5** (possible answer on page 295)

In the previous exercise we implemented a parser for the drawing language—it was an external domain language. Now implement it again as an internal language. Don't do anything clever: just write a function for each of the commands. You may have to change the names of the commands to lower case, and maybe to wrap them inside something to provide some context.

**Exercise 6** (possible answer on page 295)

Design a BNF grammar to parse a time specification. All of the following examples should be accepted:

```
4pm, 7:38pm, 23:42, 3:16, 3:16am
```

**Exercise 7** (possible answer on page 296)

Implement a parser for the BNF grammar in the previous exercise using a PEG parser generator in the language of your choice. The output should be an integer containing the number of minutes past midnight.

**Exercise 8** (possible answer on page 297)

Implement the time parser using a scripting language and regular expressions.

---


# Estimating

The Library of Congress in Washington, DC, currently has about 75 terabytes of digital information online. Quick! How long will it take to send all that information over a 1Gbps network? How much storage will you need for a million names and addresses? How long does it take to compress 100Mb of text? How many months will it take to deliver your project?

At one level, these are all meaningless questions—they are all missing information. And yet they can all be answered, as long as you are comfortable estimating. And, in the process of producing an estimate, you'll come to understand more about the world your programs inhabit.

By learning to estimate, and by developing this skill to the point where you have an intuitive feel for the magnitudes of things, you will be able to show an apparent magical ability to determine their feasibility. When someone says "we'll send the backup over a network connection to S3," you'll be able to

know intuitively whether this is practical. When you're coding, you'll be able to know which subsystems need optimizing and which ones can be left alone.

**Tip 23** Estimate to Avoid Surprises

As a bonus, at the end of this section we'll reveal the single correct answer to give whenever anyone asks you for an estimate.

## How Accurate Is Accurate Enough?

To some extent, all answers are estimates. It's just that some are more accurate than others. So the first question you have to ask yourself when someone asks you for an estimate is the context in which your answer will be taken. Do they need high accuracy, or are they looking for a ballpark figure?

One of the interesting things about estimating is that the units you use make a difference in the interpretation of the result. If you say that something will take about 130 working days, then people will be expecting it to come in pretty close. However, if you say "Oh, about six months," then they know to look for it any time between five and seven months from now. Both numbers represent the same duration, but "130 days" probably implies a higher degree of accuracy than you feel. We recommend that you scale time estimates as follows:

| Duration | Quote estimate in |
|---|---|
| 1–15 days | Days |
| 3–6 weeks | Weeks |
| 8–20 weeks | Months |
| 20+ weeks | Think hard before giving an estimate |

So, if after doing all the necessary work, you decide that a project will take 125 working days (25 weeks), you might want to deliver an estimate of "about six months."

The same concepts apply to estimates of any quantity: choose the units of your answer to reflect the accuracy you intend to convey.

## Where Do Estimates Come From?

All estimates are based on models of the problem. But before we get too deeply into the techniques of building models, we have to mention a basic estimating trick that always gives good answers: ask someone who's already done it. Before you get too committed to model building, cast around for someone

who's been in a similar situation in the past. See how their problem got solved. It's unlikely you'll ever find an exact match, but you'd be surprised how many times you can successfully draw on others' experiences.

### Understand What's Being Asked

The first part of any estimation exercise is building an understanding of what's being asked. As well as the accuracy issues discussed above, you need to have a grasp of the scope of the domain. Often this is implicit in the question, but you need to make it a habit to think about the scope before starting to guess. Often, the scope you choose will form part of the answer you give: "Assuming there are no traffic accidents and there's gas in the car, I should be there in 20 minutes."

### Build a Model of the System

This is the fun part of estimating. From your understanding of the question being asked, build a rough-and-ready bare-bones mental model. If you're estimating response times, your model may involve a server and some kind of arriving traffic. For a project, the model may be the steps that your organization uses during development, along with a very rough picture of how the system might be implemented.

Model building can be both creative and useful in the long term. Often, the process of building the model leads to discoveries of underlying patterns and processes that weren't apparent on the surface. You may even want to reexamine the original question: "You asked for an estimate to do *X*. However, it looks like *Y*, a variant of *X*, could be done in about half the time, and you lose only one feature."

Building the model introduces inaccuracies into the estimating process. This is inevitable, and also beneficial. You are trading off model simplicity for accuracy. Doubling the effort on the model may give you only a slight increase in accuracy. Your experience will tell you when to stop refining.

### Break the Model into Components

Once you have a model, you can decompose it into components. You'll need to discover the mathematical rules that describe how these components interact. Sometimes a component contributes a single value that is added into the result. Some components may supply multiplying factors, while others may be more complicated (such as those that simulate the arrival of traffic at a node).

You'll find that each component will typically have parameters that affect how it contributes to the overall model. At this stage, simply identify each parameter.

### Give Each Parameter a Value

Once you have the parameters broken out, you can go through and assign each one a value. You expect to introduce some errors in this step. The trick is to work out which parameters have the most impact on the result, and concentrate on getting them about right. Typically, parameters whose values are added into a result are less significant than those that are multiplied or divided. Doubling a line speed may double the amount of data received in an hour, while adding a 5ms transit delay will have no noticeable effect.

You should have a justifiable way of calculating these critical parameters. For the queuing example, you might want to measure the actual transaction arrival rate of the existing system, or find a similar system to measure. Similarly, you could measure the current time taken to serve a request, or come up with an estimate using the techniques described in this section. In fact, you'll often find yourself basing an estimate on other subestimates. This is where your largest errors will creep in.

### Calculate the Answers

Only in the simplest of cases will an estimate have a single answer. You might be happy to say "I can walk five cross-town blocks in 15 minutes." However, as the systems get more complex, you'll want to hedge your answers. Run multiple calculations, varying the values of the critical parameters, until you work out which ones really drive the model. A spreadsheet can be a big help. Then couch your answer in terms of these parameters. "The response time is roughly three quarters of a second if the system has SSDs and 32GB of memory, and one second with 16GB memory." (Notice how "three quarters of a second" conveys a different feeling of accuracy than 750ms.)

During the calculation phase, you get answers that seem strange. Don't be too quick to dismiss them. If your arithmetic is correct, your understanding of the problem or your model is probably wrong. This is valuable information.

### Keep Track of Your Estimating Prowess

We think it's a great idea to record your estimates so you can see how close you were. If an overall estimate involved calculating subestimates, keep track of these as well. Often you'll find your estimates are pretty good—in fact, after a while, you'll come to expect this.

When an estimate turns out wrong, don't just shrug and walk away—find out why. Maybe you chose some parameters that didn't match the reality of the problem. Maybe your model was wrong. Whatever the reason, take some time to uncover what happened. If you do, your next estimate will be better.

## Estimating Project Schedules

Normally you'll be asked to estimate how long something will take. If that "something" is complex, the estimate can be very difficult to produce. In this section, we'll look at two techniques for reducing that uncertainty.

### Painting the Missile

*"How long will it take to paint the house?"*

*"Well, if everything goes right, and this paint has the coverage they claim, it might be as few as 10 hours. But that's unlikely: I'd guess a more realistic figure is closer to 18 hours. And, of course, if the weather turns bad, that could push it out to 30 or more."*

That's how people estimate in the real world. Not with a single number (unless you force them to give you one) but with a range of scenarios.

When the U.S. Navy needed to plan the Polaris submarine project, they adopted this style of estimating with a methodology they called the *Program Evaluation Review Technique*, or PERT.

Every PERT task has an *optimistic*, a *most likely*, and a *pessimistic estimate*. The tasks are arranged into a dependency network, and then you use some simple statistics to identify likely best and worst times for the overall project.

Using a range of values like this is a great way to avoid one of the most common causes of estimation error: padding a number because you're unsure. Instead, the statistics behind PERT spreads the uncertainty out for you, giving you better estimations of the whole project.

However, we're not big fans of this. People tend to produce wall-sized charts of all the tasks in a project, and implicitly believe that, just because they used a *formula*, they have an accurate estimate. The chances are they don't, because they have never done this before.

### Eating the Elephant

We find that often the only way to determine the timetable for a project is by gaining experience on that same project. This needn't be a paradox if you

practice incremental development, repeating the following steps with very thin slices of functionality:

- Check requirements
- Analyze risk (and prioritize riskiest items earlier)
- Design, implement, integrate
- Validate with the users

Initially, you may have only a vague idea of how many iterations will be required, or how long they may be. Some methods require you to nail this down as part of the initial plan; however, for all but the most trivial of projects this is a mistake. Unless you are doing an application similar to a previous one, with the same team and the same technology, you'd just be guessing.

So you complete the coding and testing of the initial functionality and mark this as the end of the first iteration. Based on that experience, you can refine your initial guess on the number of iterations and what can be included in each. The refinement gets better and better each time, and confidence in the schedule grows along with it. This kind of estimating is often done during the team's review at the end of each iterative cycle.

That's also how the old joke says to eat an elephant: one bite at a time.

**Tip 24** Iterate the Schedule with the Code

This may not be popular with management, who typically want a single, hard-and-fast number before the project even starts. You'll have to help them understand that the team, their productivity, and the environment will determine the schedule. By formalizing this, and refining the schedule as part of each iteration, you'll be giving them the most accurate scheduling estimates you can.

## What to Say When Asked for an Estimate

You say *"I'll get back to you."*

You almost always get better results if you slow the process down and spend some time going through the steps we describe in this section. Estimates given at the coffee machine will (like the coffee) come back to haunt you.

## Related Sections Include


- Topic 7, *Communicate!*, on page 19
- Topic 39, *Algorithm Speed*, on page 203

## Challenges

- Start keeping a log of your estimates. For each, track how accurate you turned out to be. If your error was greater than 50%, try to find out where your estimate went wrong.

## Exercises

**Exercise 9** (possible answer on page 297)

You are asked "Which has a higher bandwidth: a 1Gbps net connection or a person walking between two computers with a full 1TB of storage device in their pocket?" What constraints will you put on your answer to ensure that the scope of your response is correct? (For example, you might say that the time taken to access the storage device is ignored.)

**Exercise 10** (possible answer on page 298)

So, which has the higher bandwidth?