Chapter 1

What Is the Point of Test-Driven Development?

One must learn by doing the thing; for though you think you know it, you have no certainty, until you try.

—Sophocles

Software Development as a Learning Process

Almost all software projects are attempting something that nobody has done before (or at least that nobody in the organization has done before). That *something* may refer to the people involved, the application domain, the technology being used, or (most likely) a combination of these. In spite of the best efforts of our discipline, all but the most routine projects have elements of surprise. Interesting projects—those likely to provide the most benefit—usually have a lot of surprises.

Developers often don't completely understand the technologies they're using. They have to learn how the components work *whilst* completing the project. Even if they have a good understanding of the technologies, new applications can force them into unfamiliar corners. A system that combines many significant components (which means most of what a professional programmer works on) will be too complex for any individual to understand all of its possibilities.

For customers and end users, the experience is worse. The process of building a system forces them to look at their organization more closely than they have before. They're often left to negotiate and codify processes that, until now, have been based on convention and experience.

Everyone involved in a software project has to learn as it progresses. For the project to succeed, the people involved have to work together just to understand what they're supposed to achieve, and to identify and resolve misunderstandings along the way. They all know there will be changes, they just don't know *what* changes. They need a process that will help them cope with uncertainty as their experience grows—to *anticipate unanticipated changes*.

Feedback Is the Fundamental Tool

We think that the best approach a team can take is to use *empirical feedback* to learn about the system and its use, and then apply that learning back to the system. A team needs repeated cycles of activity. In each cycle it adds new features and gets feedback about the quantity and quality of the work already done. The team members split the work into *time boxes*, within which they analyze, design, implement, and deploy as many features as they can.

Deploying completed work to some kind of environment at each cycle is critical. Every time a team deploys, its members have an opportunity to check their assumptions against reality. They can measure how much progress they're really making, detect and correct any errors, and adapt the current plan in response to what they've learned. Without deployment, the feedback is not complete.

In our work, we apply feedback cycles at every level of development, organizing projects as a system of nested loops ranging from seconds to months, such as: pair programming, unit tests, acceptance tests, daily meetings, iterations, releases, and so on. Each loop exposes the team's output to empirical feedback so that the team can discover and correct any errors or misconceptions. The nested feedback loops reinforce each other; if a discrepancy slips through an inner loop, there is a good chance an outer loop will catch it.

Each feedback loop addresses different aspects of the system and development process. The inner loops are more focused on the technical detail: what a unit of code does, whether it integrates with the rest of the system. The outer loops are more focused on the organization and the team: whether the application serves its users' needs, whether the team is as effective as it could be.

The sooner we can get feedback about any aspect of the project, the better. Many teams in large organizations can release every few weeks. Some teams release every few days, or even hours, which gives them an order of magnitude increase in opportunities to receive and respond to feedback from real users.

Incremental and Iterative Development

In a project organized as a set of nested feedback loops, development is *incremental* and *iterative*.

Incremental development builds a system feature by feature, instead of building all the layers and components and integrating them at the end. Each feature is implemented as an end-to-end "slice" through all the relevant parts of the system. The system is always integrated and ready for deployment.

Iterative development progressively refines the implementation of features in response to feedback until they are good enough.

Practices That Support Change

We've found that we need two technical foundations if we want to grow a system reliably and to cope with the *unanticipated* changes that always happen. First, we need constant testing to catch regression errors, so we can add new features without breaking existing ones. For systems of any interesting size, frequent manual testing is just impractical, so we must automate testing as much as we can to reduce the costs of building, deploying, and modifying versions of the system.

Second, we need to keep the code as simple as possible, so it's easier to understand and modify. Developers spend far more time reading code than writing it, so that's what we should optimize for.[1] Simplicity takes effort, so we constantly *refactor* [Fowler99] our code as we work with it—to improve and simplify its design, to remove duplication, and to ensure that it clearly expresses what it does. The test suites in the feedback loops protect us against our own mistakes as we improve (and therefore change) the code.

The catch is that few developers enjoy testing their code. In many development groups, writing automated tests is seen as not "real" work compared to adding features, and boring as well. Most people do not do as well as they should at work they find uninspiring.

Test-Driven Development (TDD) turns this situation on its head. We write our tests *before* we write the code. Instead of just using testing to verify our work after it's done, TDD turns testing into a *design* activity. We use the tests to clarify our ideas about *what* we want the code to do. As Kent Beck described it to us, "I was finally able to separate logical from physical design. I'd always been told to do that but no one ever explained how." We find that the effort of writing a test first also gives us rapid feedback about the quality of our design ideas—that making code accessible for testing often drives it towards being cleaner and more modular.

If we write tests all the way through the development process, we can build up a safety net of automated regression tests that give us the confidence to make changes.

"... you have nothing to lose but your bugs"

We cannot emphasize strongly enough how liberating it is to work on test-driven code that has thorough test coverage. We find that we can concentrate on the task in hand, confident that we're doing the right work and that it's actually quite hard to break the system—as long as we follow the practices.

1. Begel and Simon [Begel08] showed that new graduates at Microsoft spend most of their first year just reading code.

Test-Driven Development in a Nutshell

The cycle at the heart of TDD is: write a test; write some code to get it working; *refactor* the code to be as simple an implementation of the tested features as possible. Repeat.

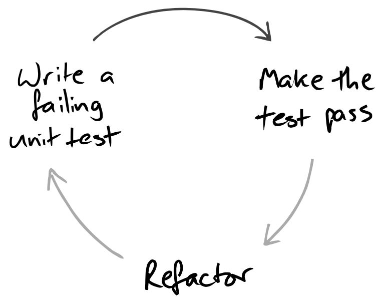

Figure 1.1 *The fundamental TDD cycle*

As we develop the system, we use TDD to give us feedback on the quality of both its *implementation* ("Does it work?") and *design* ("Is it well structured?"). Developing test-first, we find we benefit twice from the effort. *Writing* tests:

- makes us clarify the acceptance criteria for the next piece of work—we have to ask ourselves how we can tell when we're done (design);
- encourages us to write loosely coupled components, so they can easily be tested in isolation and, at higher levels, combined together (design);
- adds an executable description of what the code does (design); and,
- adds to a complete regression suite (implementation);

whereas *running* tests:

- detects errors while the context is fresh in our mind (implementation); and,
- lets us know when we've done enough, discouraging "gold plating" and unnecessary features (design).

This feedback cycle can be summed up by the Golden Rule of TDD:

The Golden Rule of Test-Driven Development

Never write new functionality without a failing test.

Refactoring. Think Local, Act Local

Refactoring means changing the internal structure of an existing body of code without changing its behavior. The point is to improve the code so that it's a better representation of the features it implements, making it more maintainable.

Refactoring is a disciplined technique where the programmer applies a series of transformations (or "refactorings") that do not change the code's behavior. Each refactoring is small enough to be easy to understand and "safe"; for example, a programmer might pull a block of code into a helper method to make the original method shorter and easier to understand. The programmer makes sure that the system is still working after each refactoring step, minimizing the risk of getting stranded by a change; in test-driven code, we can do that by running the tests.

Refactoring is a "microtechnique" that is driven by finding small-scale improvements. Our experience is that, applied rigorously and consistently, its many small steps can lead to significant structural improvements. Refactoring is not the same activity as *redesign*, where the programmers take a conscious decision to change a large-scale structure. That said, having taken a redesign decision, a team can use refactoring techniques to get to the new design incrementally and safely.

You'll see quite a lot of refactoring in our example in Part III. The standard text on the concept is Fowler's [Fowler99].

The Bigger Picture

It is tempting to start the TDD process by writing unit tests for classes in the application. This is better than having no tests at all and can catch those basic programming errors that we all know but find so hard to avoid: fencepost errors, incorrect boolean expressions, and the like. But a project with only unit tests is missing out on critical benefits of the TDD process. We've seen projects with high-quality, well unit-tested code that turned out not to be called from anywhere, or that could not be integrated with the rest of the system and had to be rewritten.

How do we know where to start writing code? More importantly, how do we know when to *stop* writing code? The golden rule tells us what we need to do: *Write a failing test.*

When we're implementing a feature, we start by writing an *acceptance test*, which exercises the functionality we want to build. While it's failing, an acceptance test demonstrates that the system does not yet implement that feature; when it passes, we're done. When working on a feature, we use its acceptance test to guide us as to whether we actually need the code we're about to write—we only write code that's directly relevant. Underneath the acceptance test, we follow the *unit level* test/implement/refactor cycle to develop the feature; the whole cycle looks like Figure 1.2.

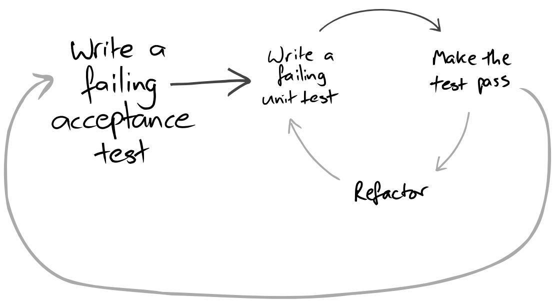

Figure 1.2 *Inner and outer feedback loops in TDD*

The outer test loop is a measure of demonstrable progress, and the growing suite of tests protects us against regression failures when we change the system. Acceptance tests often take a while to make pass, certainly more than one check-in episode, so we usually distinguish between acceptance tests we're working on (which are not yet included in the build) and acceptance tests for the features that have been finished (which are included in the build and must always pass).

The inner loop supports the developers. The unit tests help us maintain the quality of the code and should pass soon after they've been written. Failing unit tests should never be committed to the source repository.

Testing End-to-End

Wherever possible, an acceptance test should exercise the system end-to-end without directly calling its internal code. An *end-to-end test* interacts with the system only from the outside: through its user interface, by sending messages as if from third-party systems, by invoking its web services, by parsing reports, and so on. As we discuss in Chapter 10, the whole behavior of the system includes its interaction with its external environment. This is often the riskiest and most difficult aspect; we ignore it at our peril. We try to avoid acceptance tests that just exercise the internal objects of the system, unless we really need the speed-up and already have a stable set of end-to-end tests to provide cover.

The Importance of End-to-End Testing: A Horror Story

Nat was once brought onto a project that had been using TDD since its inception. The team had been writing acceptance tests to capture requirements and show progress to their customer representatives. They had been writing unit tests for the classes of the system, and the internals were clean and easy to change. They had been making great progress, and the customer representatives had signed off all the implemented features on the basis of the passing acceptance tests.

But the acceptance tests did not run end-to-end—they instantiated the system's internal objects and directly invoked their methods. The application actually did nothing at all. Its entry point contained only a single comment:

```
// TODO implement this
```

Additional feedback loops, such as regular show-and-tell sessions, should have been in place and would have caught this problem.

For us, "end-to-end" means more than just interacting with the system from the outside—that might be better called "edge-to-edge" testing. We prefer to have the end-to-end tests exercise both the system *and* the process by which it's built and deployed. An automated build, usually triggered by someone checking code into the source repository, will: check out the latest version; compile and unit-test the code; integrate and package the system; perform a production-like deployment into a realistic environment; and, finally, exercise the system through its external access points. This sounds like a lot of effort (it is), but has to be done anyway repeatedly during the software's lifetime. Many of the steps might be fiddly and error-prone, so the end-to-end build cycle is an ideal candidate for automation. You'll see in Chapter 10 how early in a project we get this working.

A system is *deployable* when the acceptance tests all pass, because they should give us enough confidence that everything works. There's still, however, a final step of deploying to production. In many organizations, especially large or heavily regulated ones, building a deployable system is only the start of a release process. The rest, before the new features are finally available to the end users, might involve different kinds of testing, handing over to operations and data groups, and coordinating with other teams' releases. There may also be additional, nontechnical costs involved with a release, such as training, marketing, or an impact on service agreements for downtime. The result is a more difficult release cycle than we would like, so we have to understand our whole technical and organizational environment.

Levels of Testing

We build a hierarchy of tests that correspond to some of the nested feedback loops we described above:

Acceptance: Does the whole system work?

Integration: Does our code work against code we can't change?

Unit: Do our objects do the right thing, are they convenient to work with?

There's been a lot of discussion in the TDD world over the terminology for what we're calling *acceptance tests*: "functional tests," "customer tests," "system tests." Worse, our definitions are often not the same as those used by professional software testers. The important thing is to be clear about our intentions. We use "acceptance tests" to help us, with the domain experts, understand and agree on what we are going to build next. We also use them to make sure that we haven't broken any existing features as we continue developing.

Our preferred implementation of the "role" of acceptance testing is to write *end-to-end tests* which, as we just noted, should be as end-to-end as possible; our bias often leads us to use these terms interchangeably although, in some cases, acceptance tests might not be end-to-end.

We use the term *integration tests* to refer to the tests that check how some of our code works with code from outside the team that we can't change. It might be a public framework, such as a persistence mapper, or a library from another team within our organization. The distinction is that integration tests make sure that any abstractions we build over third-party code work as we expect. In a small system, such as the one we develop in Part III, acceptance tests might be enough. In most professional development, however, we'll want integration tests to help tease out configuration issues with the external packages, and to give quicker feedback than the (inevitably) slower acceptance tests.

We won't write much more about techniques for acceptance and integration testing, since both depend on the technologies involved and even the culture of the organization. You'll see some examples in Part III which we hope give a sense of the motivation for acceptance tests and show how they fit in the development cycle. Unit testing techniques, however, are specific to a style of programming, and so are common across all systems that take that approach—in our case, are object-oriented.

External and Internal Quality

There's another way of looking at what the tests can tell us about a system. We can make a distinction between external and internal quality: *External* quality is how well the system meets the needs of its customers and users (is it functional, reliable, available, responsive, etc.), and *internal* quality is how well it meets the needs of its developers and administrators (is it easy to understand, easy to change, etc.). Everyone can understand the point of external quality; it's usually part of the contract to build. The case for internal quality is equally important but is often harder to make. Internal quality is what lets us cope with continual and unanticipated change which, as we saw at the beginning of this chapter, is a fact of working with software. The point of maintaining internal quality is to allow us to modify the system's behavior safely and predictably, because it minimizes the risk that a change will force major rework.

Running end-to-end tests tells us about the external quality of our system, and *writing* them tells us something about how well we (the whole team) understand the domain, but end-to-end tests don't tell us how well we've written the code. *Writing* unit tests gives us a lot of feedback about the quality of our code, and *running* them tells us that we haven't broken any classes—but, again, unit tests don't give us enough confidence that the system as a whole works. Integration tests fall somewhere in the middle, as in Figure 1.3.

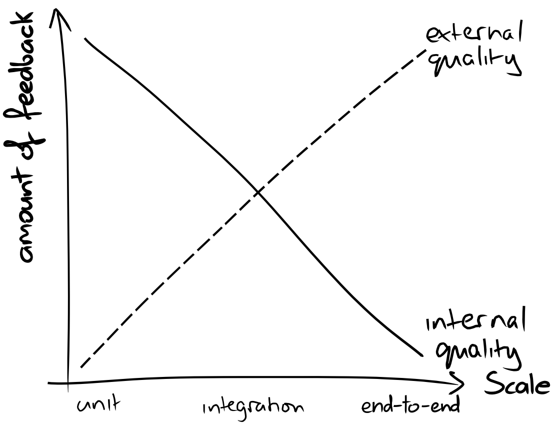

Figure 1.3 *Feedback from tests*

Thorough unit testing helps us improve the internal quality because, to be tested, a unit has to be structured to run outside the system in a test fixture. A unit test for an object needs to create the object, provide its dependencies, interact with it, and check that it behaved as expected. So, for a class to be easy to unit-test, the class must have explicit dependencies that can easily be substituted and clear responsibilities that can easily be invoked and verified. In software engineering terms, that means that the code must be *loosely coupled* and *highly cohesive*—in other words, well-designed.

When we've got this wrong—when a class, for example, is tightly coupled to distant parts of the system, has implicit dependencies, or has too many or unclear responsibilities—we find unit tests difficult to write or understand, so writing a test first gives us valuable, immediate feedback about our design. Like everyone, we're tempted not to write tests when our code makes it difficult, but we try to resist. We use such difficulties as an opportunity to investigate why the test is hard to write and refactor the code to improve its structure. We call this "listening to the tests," and we'll work through some common patterns in Chapter 20.

Coupling and Cohesion

Coupling and *cohesion* are metrics that (roughly) describe how easy it will be to change the behavior of some code. They were described by Larry Constantine in [Yourdon79].

Elements are *coupled* if a change in one forces a change in the other. For example, if two classes inherit from a common parent, then a change in one class might require a change in the other. Think of a combo audio system: It's tightly coupled because if we want to change from analog to digital radio, we must rebuild the whole system. If we assemble a system from separates, it would have low coupling and we could just swap out the receiver. "Loosely" coupled features (i.e., those with low coupling) are easier to maintain.

An element's *cohesion* is a measure of whether its responsibilities form a meaningful unit. For example, a class that parses both dates and URLs is not coherent, because they're unrelated concepts. Think of a machine that washes both clothes and dishes—it's unlikely to do both well.[2] At the other extreme, a class that parses only the punctuation in a URL is unlikely to be coherent, because it doesn't represent a whole concept. To get anything done, the programmer will have to find other parsers for protocol, host, resource, and so on. Features with "high" coherence are easier to maintain.

2. Actually, there was a combined clothes and dishwasher. The "Thor Automagic" was manufactured in the 1940s, but the idea hasn't survived.

Chapter 2

Test-Driven Development with Objects

Music is the space between the notes.

—Claude Debussy

A Web of Objects

Object-oriented design focuses more on the communication between objects than on the objects themselves. As Alan Kay [Kay98] wrote:

> The big idea is "messaging" [...] The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

An object communicates by messages: It receives messages from other objects and reacts by sending messages to other objects as well as, perhaps, returning a value or exception to the original sender. An object has a *method* of handling every type of message that it understands and, in most cases, encapsulates some internal state that it uses to coordinate its communication with other objects.

An object-oriented system is a web of collaborating objects. A system is built by creating objects and plugging them together so that they can send messages to one another. The behavior of the system is an emergent property of the composition of the objects—the choice of objects and how they are connected (Figure 2.1).

This lets us change the behavior of the system by changing the composition of its objects—adding and removing instances, plugging different combinations together—rather than writing procedural code. The code we write to manage this composition is a *declarative* definition of the how the web of objects will behave. It's easier to change the system's behavior because we can focus on *what* we want it to do, not *how*.

Values and Objects

When designing a system, it's important to distinguish between *values* that model unchanging quantities or measurements, and *objects* that have an identity, might change state over time, and model *computational processes*. In the

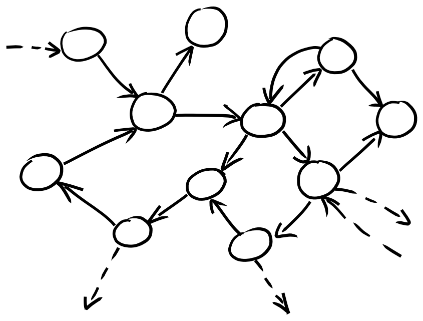

Figure 2.1 *A web of objects*

object-oriented languages that most of us use, the confusion is that both concepts are implemented by the same language construct: classes.

Values are immutable instances that model fixed quantities. They have no individual identity, so two value instances are effectively the same if they have the same state. This means that it makes no sense to compare the identity of two values; doing so can cause some subtle bugs—think of the different ways of comparing two copies of `new Integer(999)`. That's why we're taught to use `string1.equals(string2)` in Java rather than `string1 == string2`.

Objects, on the other hand, use mutable state to model their behavior over time. Two objects of the same type have separate identities even if they have exactly the same state now, because their states can diverge if they receive different messages in the future.

In practice, this means that we split our system into two "worlds": values, which are treated functionally, and objects, which implement the stateful behavior of the system. In Part III, you'll see how our coding style varies depending on which world we're working in.

In this book, we will use the term *object* to refer only to instances with identity, state, and processing—not values. There doesn't appear to be another accepted term that isn't overloaded with other meanings (such as *entity* and *process*).

Follow the Messages

We can benefit from this high-level, declarative approach only if our objects are designed to be easily pluggable. In practice, this means that they follow common *communication patterns* and that the dependencies between them are made explicit. A communication pattern is a set of rules that govern how a group of objects talk to each other: the roles they play, what messages they can send and when, and so on. In languages like Java, we identify object roles with (abstract) interfaces, rather than (concrete) classes—although interfaces don't define everything we need to say.

In our view, *the domain model is in these communication patterns*, because they are what gives meaning to the universe of possible relationships between the objects. Thinking of a system in terms of its dynamic, communication structure is a significant mental shift from the static classification that most of us learn when being introduced to objects. The domain model isn't even obviously visible because the communication patterns are not explicitly represented in the programming languages we get to work with. We hope to show, in this book, how tests and mock objects help us see the communication between our objects more clearly.

Here's a small example of how focusing on the communication between objects guides design.

In a video game, the objects in play might include: *actors*, such as the player and the enemies; *scenery*, which the player flies over; *obstacles*, which the player can crash into; and *effects*, such as explosions and smoke. There are also *scripts* spawning objects behind the scenes as the game progresses.

This is a good classification of the game objects from the players' point of view because it supports the decisions they need to make when playing the game—when interacting with the game from *outside*. This is not, however, a useful classification for the implementers of the game. The game engine has to display objects that are *visible*, tell objects that are *animated* about the passing of time, detect collisions between objects that are *physical*, and delegate decisions about what to do when physical objects collide to *collision resolvers*.

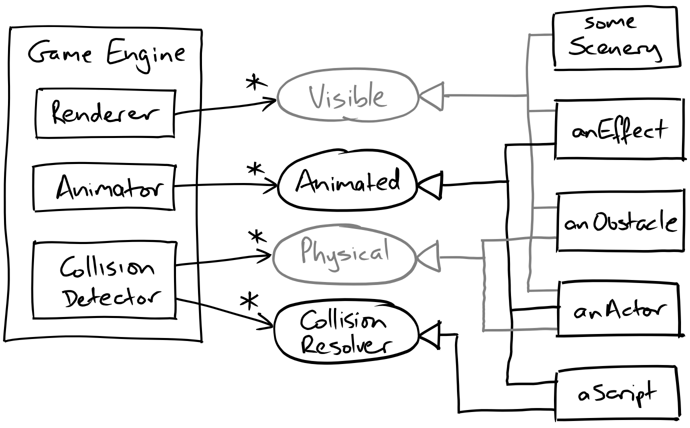

Figure 2.2 *Roles and objects in a video game*

As you can see in Figure 2.2, the two views, one from the game engine and one from the implementation of the in-play objects, are not the same. An Obstacle, for example, is *Visible* and *Physical*, while a Script is a *Collision Resolver* and *Animated* but not *Visible*. The objects in the game play different roles depending on what the engine needs from them at the time. This mismatch between static classification and dynamic communication means that we're unlikely to come up with a tidy class hierarchy for the game objects that will also suit the needs of the engine.

At best, a class hierarchy represents one dimension of an application, providing a mechanism for sharing implementation details between objects; for example, we might have a base class to implement the common features of frame-based animation. At worst, we've seen too many codebases (including our own) that suffer complexity and duplication from using one mechanism to represent multiple concepts.

Roles, Responsibilities, Collaborators

We try to think about objects in terms of roles, responsibilities, and collaborators, as best described by Wirfs-Brock and McKean in [Wirfs-Brock03]. An object is an implementation of one or more *roles*; a role is a set of related *responsibilities*; and a responsibility is an obligation to perform a task or know information. A *collaboration* is an interaction of objects or roles (or both).

Sometimes we step away from the keyboard and use an informal design technique that Wirfs-Brock and McKean describe, called *CRC cards* (Candidates, Responsibilities, Collaborators). The idea is to use low-tech index cards to explore the potential object structure of an application, or a part of it. These index cards allow us to experiment with structure without getting stuck in detail or becoming too attached to an early solution.

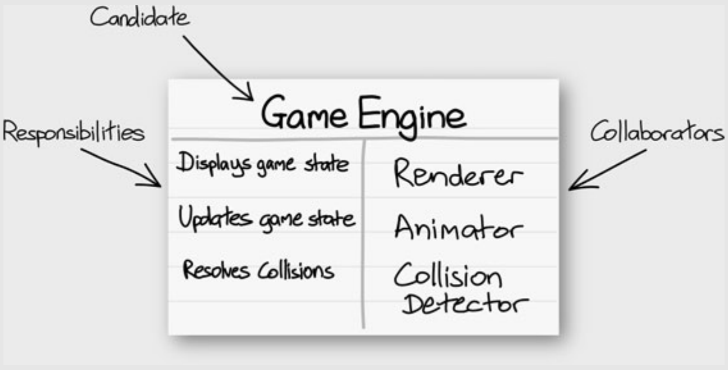

Figure 2.3 *CRC card for a video game*

Tell, Don't Ask

We have objects sending each other messages, so what do they say? Our experience is that the calling object should describe what it wants *in terms of the role* that its neighbor plays, and let the called object decide how to make that happen. This is commonly known as the "*Tell, Don't Ask*" style or, more formally, the *Law of Demeter*. Objects make their decisions based only on the information they hold internally or that which came with the triggering message; they avoid navigating to other objects to make things happen. Followed consistently, this style produces more flexible code because it's easy to swap objects that play the same role. The caller sees nothing of their internal structure or the structure of the rest of the system behind the role interface.

When we don't follow the style, we can end up with what's known as "*train wreck*" code, where a series of getters is chained together like the carriages in a train. Here's one case we found on the Internet:

```
((EditSaveCustomizer) master.getModelisable()
  .getDockablePanel()
    .getCustomizer())
      .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

After some head scratching, we realized what this fragment was *meant* to say:

```
master.allowSavingOfCustomisations();
```

This wraps all that implementation detail up behind a single call. The client of `master` no longer needs to know anything about the types in the chain. We've reduced the risk that a design change might cause ripples in remote parts of the codebase.

As well as hiding information, there's a more subtle benefit from "Tell, Don't Ask." It forces us to make explicit and so name the interactions between objects, rather than leaving them implicit in the chain of getters. The shorter version above is much clearer about *what* it's for, not just *how* it happens to be implemented.

But Sometimes Ask

Of course we don't "tell" everything;[1] we "ask" when getting information from values and collections, or when using a factory to create new objects. Occasionally, we also ask objects about their state when searching or filtering, but we still want to maintain expressiveness and avoid "train wrecks."

For example (to continue with the metaphor), if we naively wanted to spread reserved seats out across the whole of a train, we might start with something like:

1. Although that's an interesting exercise to try, to stretch your technique.

```
public class Train {
  private final List<Carriage> carriages […]
  private int percentReservedBarrier = 70;

  public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
      if (carriage.getSeats().getPercentReserved() < percentReservedBarrier) {
        request.reserveSeatsIn(carriage);
        return;
      }
    }
    request.cannotFindSeats();
  }
}
```

We shouldn't expose the internal structure of `Carriage` to implement this, not least because there may be different types of carriages within a train. Instead, we should ask the question we really want answered, instead of asking for the information to help us figure out the answer ourselves:

```
public void reserveSeats(ReservationRequest request) {
  for (Carriage carriage : carriages) {
    if (carriage.hasSeatsAvailableWithin(percentReservedBarrier)) {
      request.reserveSeatsIn(carriage);
      return;
    }
  }
  request.cannotFindSeats();
}
```

Adding a query method moves the behavior to the most appropriate object, gives it an explanatory name, and makes it easier to test.

We try to be sparing with queries on objects (as opposed to values) because they can allow information to "leak" out of the object, making the system a little bit more rigid. At a minimum, we make a point of writing queries that describe the intention of the calling object, not just the implementation.

Unit-Testing the Collaborating Objects

We appear to have painted ourselves into a corner. We're insisting on focused objects that send commands to each other and don't expose any way to query their state, so it looks like we have nothing available to assert in a unit test. For example, in Figure 2.4, the circled object will send messages to one or more of its three neighbors when invoked. How can we test that it does so correctly without exposing any of its internal state?

One option is to replace the target object's neighbors in a test with substitutes, or *mock objects*, as in Figure 2.5. We can specify how we expect the target object to communicate with its mock neighbors for a triggering event; we call these specifications *expectations*. During the test, the mock objects assert that they

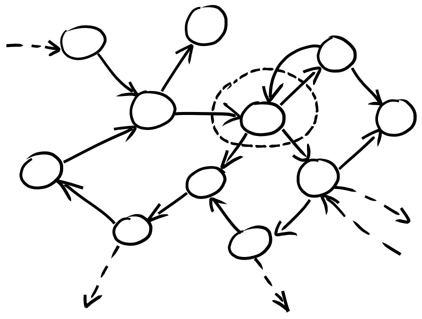

Figure 2.4 *Unit-testing an object in isolation*

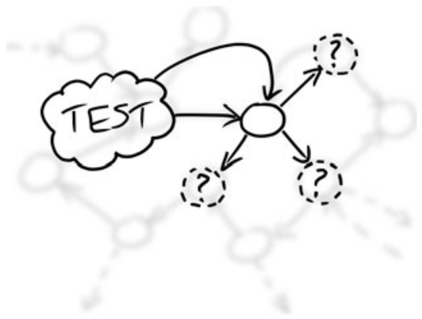

Figure 2.5 *Testing an object with mock objects*

have been called as expected; they also implement any stubbed behavior needed to make the rest of the test work.

With this infrastructure in place, we can change the way we approach TDD. Figure 2.5 implies that we're just trying to test the target object and that we already know what its neighbors look like. In practice, however, those collaborators don't need to exist when we're writing a unit test. We can use the test to help us tease out the supporting roles our object needs, defined as Java interfaces, and fill in real implementations as we develop the rest of the system. We call this *interface discovery*; you'll see an example when we extract an `AuctionEventListener` in Chapter 12.

Support for TDD with Mock Objects

To support this style of test-driven programming, we need to create mock instances of the neighboring objects, define expectations on how they're called and then check them, and implement any stub behavior we need to get through the test. In practice, the runtime structure of a test with mock objects usually looks like Figure 2.6.

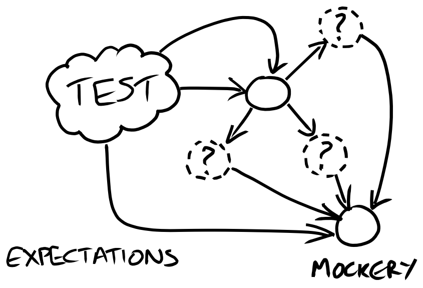

Figure 2.6 *Testing an object with mock objects*

We use the term *mockery*[2] for the object that holds the context of a test, creates mock objects, and manages expectations and stubbing for the test. We'll show the practice throughout Part III, so we'll just touch on the basics here. The essential structure of a test is:

- Create any required mock objects.
- Create any real objects, including the target object.
- Specify how you *expect* the mock objects to be called by the target object.
- Call the triggering method(s) on the target object.
- Assert that any resulting values are valid and that all the expected calls have been made.

The unit test makes explicit the relationship between the target object and its environment. It creates all the objects in the cluster and makes assertions about the interactions between the target object and its collaborators. We can code this infrastructure by hand or, these days, use one of the multiple mock object frameworks that are available in many languages. The important point, as we stress repeatedly throughout this book, is to make clear the intention of every test, distinguishing between the tested functionality, the supporting infrastructure, and the object structure.

2. This is a pun by Ivan Moore that we adopted in a fit of whimsy.