11 Design Patterns for SaaS Apps

William Kahan (1933–) received the 1989 Turing Award for his fundamental contributions to numerical analysis. Kahan dedicated himself to "making the world safe for numerical computations."

Things are genuinely simple when you can think correctly about what's going on without having a lot of extraneous or confusing thoughts to impede you. Think of Einstein's maxim, "Everything should be made as simple as possible, but no simpler."

—"A Conversation with William Kahan," *Dr. Dobbs' Journal*, 1997

Prerequisites and Concepts

The big concept of this chapter is that ***design patterns*** can improve the quality of the classes. A design pattern captures proven solutions to problems by separating the things that change from those that don't.

Concepts:

Five object-oriented design principles identified by the acronym SOLID describe sound design of interactions among classes. An ***antipattern*** indicates poor class design, which a ***design smell*** can identify. Thus, using design smells to detect violations to the SOLID principles for good class design is analogous to using ***code smells*** to detect violations of the SOFA principles for good method design (Section 9.5).

The five letters of the SOLID acronym stand for:

1. Single Responsibility Principle: a class should have one and only one responsibility; that is, only one reason to change. The Lack of Cohesion Of Methods metric indicates the antipattern of too large a class.

2. ***Open/Closed Principle***: a class should be open for extension, but closed against modification. The Case Statement design smell suggests a violation.

3. ***Liskov Substitution Principle***: a method designed to work on an object of type T should also work on an object of any subtype of T. That is, all of T's subtypes should preserve T's "contract." The refused bequest design smell often indicates a violation.

4. Dependency Injection Principle: if two classes depend on each other but their implementations may change, it would be better for them to both depend on a separate abstract interface which is "injected" between them.

5. ***Demeter Principle***: a method can call other methods in its own class, and methods on the classes of its own instance variables; everything else is taboo. A design smell that indicates a violation is inappropriate intimacy.

For Agile, ***refactoring*** is the vehicle for improving the design of classes and methods; in some cases refactoring may allow you to apply an appropriate ***design pattern***. In contrast, for the Plan-and-Document lifecycles:

- The early design phase makes it easier to select a good initial software architecture and class designs.

- The specification is broken into problems and then into subproblems, where developers try to use patterns to solve them.

- As design precedes coding, ***design reviews*** can offer early feedback.

- One concern is whether the design must change once coding begins.

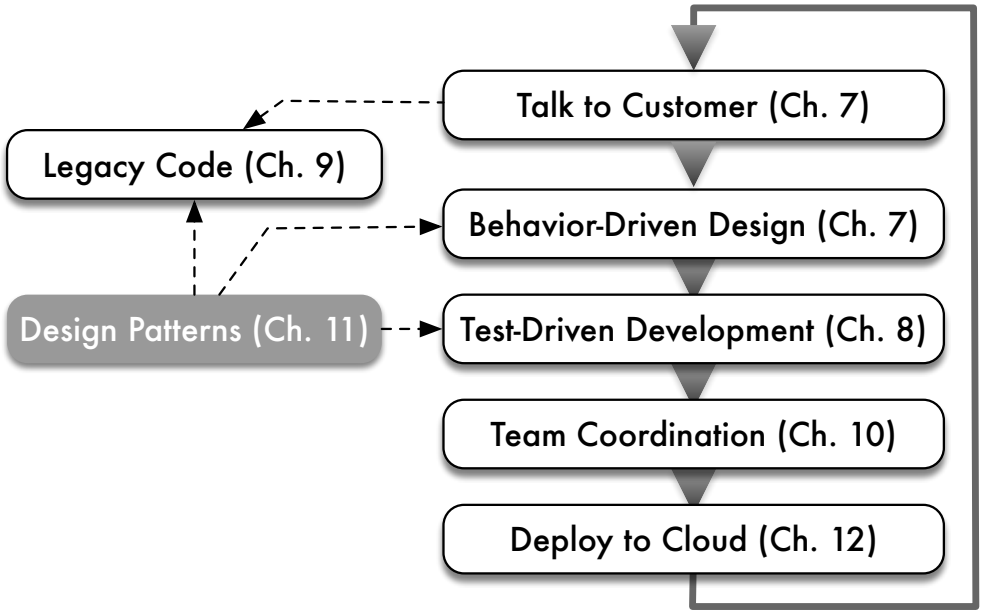

Figure 11.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers design patterns, which influence BDD and TDD for new apps *and* for enhancing legacy code.

11.1 Patterns, Antipatterns, and SOLID Class Architecture

In Chapter 3, we introduced the idea of a design pattern: a reusable structure, behavior, strategy, or technique that captures a proven solution to a collection of similar problems by *separating the things that change from those that stay the same*. Patterns play a major role in helping us achieve our goal throughout this book: producing code that is not only correct (TDD) and meets a customer need (BDD), but is also concise, readable, DRY, and generally beautiful. Figure 11.1 highlights the role of design patterns in the Agile lifecycle as covered in this chapter.

While we have already seen architectural patterns such as Client–Server and structural patterns such as Model–View–Controller, this chapter examines design patterns that apply to classes and class architecture. As Figure 11.2 shows, we will follow a similar approach as we did in Chapter 9. Rather than simply listing a catalog of design patterns, we'll motivate their use by starting from some guidelines about what makes a class architecture good or bad, identifying smells and metrics that indicate possible problem spots, and showing how some of these problems can be fixed by refactoring—both within classes and by moving code across classes—to eliminate the problems. In some cases, we can refactor to make the code match an existing and proven design pattern. In other cases, the refactoring doesn't necessarily result in major structural changes to the class architecture.

As with method-level refactoring, application of design patterns is best learned by doing, and the number of design patterns exceeds what we can cover in one chapter of one book. Indeed, there are entire books just on design patterns, including the seminal *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1994), whose authors became known as the "Gang of Four" or GoF, and their catalog known as the "***GoF design pat-***

Chapter 9	Chapter 11
Code smells warn of problems in methods of a class	Design smells warn of problems in relationships among classes
Many catalogs of code smells and refactorings; we use Fowler's as definitive	Many catalogs of design smells and design patterns; we use Ruby-specific versions of the Gang of Four (GoF) design patterns as definitive
ABC, Cyclomatic Complexity metrics complement code smells with quantitative warnings	LCOM (Lack of Cohesion of Methods) metric complements design smells with quantitative warnings
Refactoring by extracting methods and moving code within a class	Refactoring by extracting classes and moving code between classes
SOFA guidelines for good methods (**S**hort, do **O**ne thing, **F**ew arguments, single **A**bstraction level)	SOLID guidelines for good class architecture (**S**ingle responsibility, **O**pen/Closed, **L**iskov substitution, dependency **I**njection, **D**emeter)
Some code smells don't apply in Ruby	Some design smells don't apply in Ruby or SaaS

Figure 11.2: The parallels between the warning symptoms and remedies introduced for individual classes and methods in Chapter 9 and those introduced for inter-class relationships in this chapter. For reasons explained in the text, whereas most books use the I in SOLID for Interface Segregation (a smell that doesn't arise in Ruby) and D for injecting Dependencies, we instead use I for Injecting dependencies and D for the Demeter principle, which arises frequently in Ruby.

terns." The 23 GoF design patterns are divided into Creational, Structural, and Behavioral design patterns, as Figure 11.3 shows. As with Fowler's original book on refactoring, the GoF design patterns book gave rise to other books with examples tailored to specific languages including Ruby (Olsen 2007).

The GoF authors cite two overarching principles of good object-oriented design that inform most of the patterns:

- Prefer Composition and Delegation over Inheritance.
- Program to an Interface, not an Implementation.

We will learn what these catch-phrases mean as we explore some specific design patterns.

In an ideal world, all programmers would use design patterns tastefully, continuously refactoring their code as Chapter 9 suggests, and all code would be beautiful. Needless to say, this is not always the case. An *antipattern* is a piece of code that seems to want to be expressed in terms of a well-known design pattern, but isn't—often because the original (good) code has evolved to fill new needs without refactoring along the way. *Design smells*, similar to the code smells we saw in Chapter 9, are warning signs that your code may be headed towards an antipattern. In contrast to code smells, which typically apply to methods within a class, design smells apply to relationships between classes and how responsibilities are divided among them. Therefore, whereas refactoring a method involves moving code around *within* a class, refactoring a design involves moving code *between* classes, creating new classes or modules (perhaps by extracting commonality from existing ones), or removing classes that aren't pulling their weight.

Similar to SOFA in Chapter 9, the mnemonic SOLID (credited to Robert C. Martin) stands for a set of five design principles that clean code should respect. As in Chapter 9, design smells and quantitative metrics can tell us when we're in danger of violating one or more SOLID guidelines; the fix is often a refactoring that eliminates the problem by bringing the code in line with one or more design patterns.

Since the GoF design patterns evolved in the context of statically typed languages, some of them address problems that don't arise in Ruby. For example, patterns that eliminate type signature changes that would trigger recompilation are rarely used in Ruby, which isn't compiled and doesn't use types to enforce contracts.

"Uncle Bob" Martin, an American software engineer and consultant[1] since 1970, is a founder of Agile/XP and a leading member of the ***Software Craftsmanship*** movement, which encourages programmers to see themselves as creative professionals learning a disciplined craft in an apprenticeship model.

Creational patterns
***Abstract Factory*, Factory Method**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes **Singleton:** Ensure a class has only one instance, and provide a global point of access to it. **Prototype:** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. As we've seen in Chapter 6, prototype-based inheritance is part of the JavaScript language. **Builder:** Separate the construction of a complex object from its representation allowing the same construction process to create various representations
Structural patterns
Adapter, Proxy, Façade*, Bridge**: Convert the programming interface of a class into another (sometimes simpler) interface that clients expect, or decouple an abstraction's interface from its implementation, for dependency injection or performance ***Decorator: Attach additional responsibilities to an object dynamically, keeping the same interface. Helps with "Prefer composition or delegation over inheritance." **Composite**: Provide operations that work on both an individual object and a collection of that type of object **Flyweight:** Use sharing to support large numbers of similar objects efficiently
Behavioral patterns
Template Method, Strategy: Uniformly encapsulate multiple varying strategies for same task ***Observer:*** One or more entities need to be notified when something happens to an object ***Iterator, Visitor:*** Separate traversal of a data structure from operations performed on each element of the data structure ***Null Object:*** (Doesn't appear in GoF catalog) Provide an object with defined neutral behaviors that can be safely called, to take the place of conditionals guarding method calls **State:** Encapsulate an object whose behaviors (methods) differ depending on which of a small number of internal states the object is in **Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request, passing request up the chain until someone handles it **Mediator:** Define an object that encapsulates how a set of objects interact without those objects having to refer to each other explicitly, allowing decoupling **Interpreter:** Define a representation for a language along with an interpreter that executes the representation **Command:** Encapsulate an operation request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Figure 11.3: The 23 *GoF design patterns* spanning three categories, with italics showing a subset we'll encounter as we illustrate and fix SOLID violations and with closely-related patterns grouped into a single entry, as with Abstract Factory and Factory Method. Whenever we introduce a design pattern, we'll explain the pattern's goal, show a Unified Modeling Language representation (introduced in the next section) of the class architecture before and after refactoring to that pattern, and when possible, give an example of how the pattern is used "in the wild" in Rails itself or in a Ruby gem.

Principle	Meaning	Warning smells	Refactoring fix
Single Responsibility	A class should have one and only one reason to change	Large class, poor LCOM (Lack of Cohesion Of Methods) score, data clumps	Extract class, move methods
Open/Closed	Classes should be open for extension but closed for modification	Conditional complexity, `case`-based dispatcher	Use Strategy or Template Method, possibly combined with Abstract Factory pattern; use Decorator to avoid explosion of subclasses
Liskov Substitution	Substituting a subclass for a class should preserve correct program behavior	Refused bequest: subclass destructively overrides an inherited method	Replace inheritance with delegation
Injection of Dependencies	Collaborating classes whose implementation may vary at runtime should depend on an intermediate "injected" dependency	Unit tests that require ad hoc stubbing to create seams; constructors that hardwire a call to another class's constructor, rather than allowing runtime determination of *which* other class to use	Inject a dependency on a shared interface to isolate the classes; use Adapter, Façade, or Proxy patterns as needed to make the interface uniform across variants
Demeter Principle	Speak only to your friends; treat your friends' friends as strangers	Inappropriate intimacy, feature envy, mock trainwrecks	Delegate behaviors and call the delegate methods instead

Figure 11.4: The SOLID design guidelines and some smells that may suggest your code violates one or more of them. We diverge a little bit from standard usage of SOLID: we use I for Injecting dependencies and D for the Demeter principle, whereas most books use I for Interface Segregation (which doesn't apply in Ruby) and D for injecting Dependencies.

Figure 11.4 shows the SOLID mnemonics and what they tell us about good composition of classes. In our discussion of selected design patterns, we'll see violations of each one of these guidelines, and show how refactoring the bad code (in some cases, with the goal of applying a design pattern) can fix the violation. In general, the SOLID principles strive for a class architecture that avoids various problems that thwart productivity:

1. Viscosity: it's easier to fix a problem using a quick hack, even though you know that's not the right thing to do.

2. Immobility: it's hard to be DRY and because the functionality you want to reuse is wired into the app in a way that makes extraction difficult.

3. Needless repetition: possibly as a consequence of immobility, the app has similar functionality duplicated in multiple places. As a result, a change in one part of the app often ripples to many other parts of the app, so that a small change in functionality requires a lot of little changes to code and tests, a process sometimes called *shotgun surgery*.

4. Needless complexity: the app's design reflects generality that was inserted before it was needed.

As with refactoring and legacy code, seeking out design smells and addressing them by refactoring with judicious use of design patterns is a skill learned by doing. Therefore, rather than presenting "laundry lists" of design smells, refactorings, and design patterns, we focus our discussion around the SOLID principles and give a few representative examples of the overall process of identifying design smells and assessing the alternatives for addressing

them. As you tackle your own applications, perusing the more detailed resources listed in Section 11.11 is essential.

Summary of patterns, antipatterns and SOLID:

- Good code should accommodate evolutionary change gracefully. Design patterns are proven solutions to common problems that thwart this goal. They work by providing a clean way to separate the things that may change or evolve from those that stay the same and a clean way to accommodate those changes.
- Just as with individual methods, refactoring is the process of improving the structure of a class architecture to make the code more maintainable and evolvable by moving code across classes as well as refactoring within the class. In some cases, these refactorings lead us to one of the 23 "Gang of Four" (GoF) design patterns.
- Just as with individual methods, design smells and metrics can serve as early warnings of an *antipattern*—a piece of code that would be better structured if it followed a design pattern.

■ ***Elaboration: Other types of patterns***

As we've emphasized since the beginning of this book, judicious use of patterns pervades good software engineering. To complement class-level design patterns, others have developed catalogs of architectural patterns for enterprise applications[2] (we met some in Chapter 3), parallel programming patterns, computational patterns (to support specific algorithm families such as graph algorithms, linear algebra, circuits, grids, and so on), ***Concurrency patterns***, and user interface patterns[3].

Self-Check 11.1.1

True or false: one measure of the quality of a piece of software is the degree to which it uses design patterns.

◇ False: while design patterns provide proven solutions to some common problems, code that doesn't exhibit such problems may not need those patterns, but that doesn't make it poor code. The GoF authors specifically warn against measuring code quality in terms of design pattern usage. ■

Grady Booch (1955–), internationally recognized for his work in software engineering and collaborative development environments, developed UML with Ivar Jacobson and James Rumbaugh.

11.2 Just Enough UML

The ***Unified Modeling Language*** or UML is not a textual language, but a set of graphical notation techniques that provide a "standard way to visualize the design of a [software] system." UML evolved from 1995 to the present through the unification of previously-distinct modeling language standards and diagram types, which Figure 11.5 lists.

While this book focuses on more lightweight Agile modeling—indeed, UML-based modeling has been criticized as being too "bloated" and heavyweight—some types of UML diagrams are widely used even in Agile modeling. Figure 11.6 shows a UML ***class diagram***, which depicts each actual class in the app, its most important class and instance variables and methods, and its relationship to other classes, such as has-many or belongs-to associations.

Structure diagrams	
Class	Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
Component	Describes how a software system is split up into components and shows the dependencies among these components.
Composite structure	Describes the internal structure of a class and the collaborations that this structure makes possible.
Deployment	Describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
Object	Shows a complete or partial view of the structure of an example modeled system at a specific time.
Package	Describes how a system is split up into logical groupings by showing the dependencies among these groupings.
Profile	Describes reusable domain-specific "stereotype" objects from which specific object types can be derived for use in a particular application.
Interaction diagrams	
Communication	Shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
Interaction overview	Provides an overview in which the nodes represent communication diagrams.
Sequence	Shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
Timing	A specific type of interaction diagram where the focus is on timing constraints.
Behavior diagrams	
Activity	Describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
State machine	Describes the states and state transitions of the system.
Use Case	Describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Figure 11.5: The fourteen types of diagrams defined by UML 2.2 for describing a software system. These descriptions are based on the excellent Wikipedia summary of UML[5], which also shows an example of each diagram type. Use case diagrams are similar to Agile user stories, but lack the level of detail that allows tools like Cucumber to bridge the gap between user stories and integration/acceptance tests.

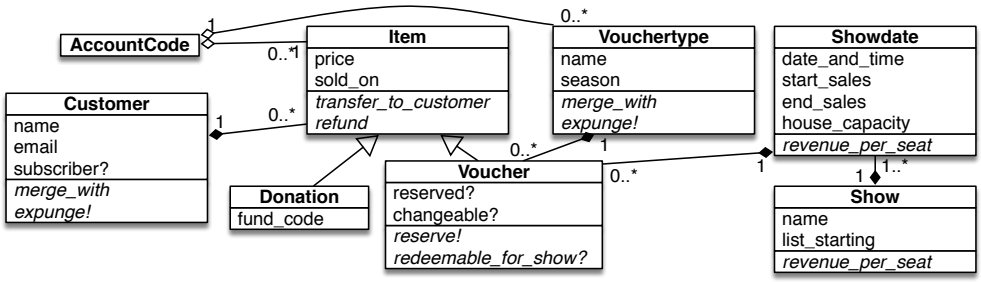

Figure 11.6: This UML *class diagram* shows a subset of the classes in the theater-ticketing app consistent with Figures 9.4 and 7.4. Each box represents a class with its most important methods and attributes (responsibilities). Inheritance is represented by an arrow. Classes with associations are connected by lines whose endpoints are annotated with a multiplicity and optionally a diamond—open for aggregations, filled for compositions, absent otherwise.

Each end of the line connecting two associated classes is annotated with the minimum and maximum number of instances that can participate in that "side" of the association, called the association's *multiplicity*, using the symbol $*$ for "unlimited". For example, a multiplicity `1..*` means "one or more", `0..*` means "zero or more", and `1` means "exactly one." UML distinguishes two kinds of "owning" (has-one or has-many) associations. In an *aggregation*, the owned objects survive destruction of the owning object. For example, *Course has many Students* is an aggregation because the students happily don't get destroyed when the course is over! In a *composition*, the owned objects are usually destroyed when the owning object is destroyed. For example, *Movie has many Reviews* is a composition since deleting a Movie should cause all of its reviews to be deleted.

Class diagrams are popular even among software engineers who don't use the other parts of UML. With this introduction to UML in hand, we can use class diagrams to illustrate "before and after" class architecture when we improve code using the SOLID guidelines and design patterns.

Summary of Unified Modeling Language (UML):

- UML comprises a family of diagram types to illustrate various aspects of a software design and implementation.
- UML class diagrams are widely used even by engineers who don't use other UML features. They show a class's name, its most important public and private methods and attributes, and its relationship to other classes.

■ ***Elaboration: When to use UML?***

While heavyweight, UML is useful for modeling very large applications divided into subsystems being worked on by widely-distributed teams. Also, since UML notation is language-neutral, it can be helpful for coordinating international teams. Because of UML's maturity, many tools support its use; the challenge is keeping the diagrams "in sync" with the code and the design, which is why most such tools try to go in both directions, synthesizing code skeletons from UML and extracting UML diagrams from code. One such tool useful for learning UML is UMPLE[6], a domain-specific language developed at the University of Ottawa for expressing class relationships. The Try Umple[7] web site can generate UML class diagrams from UMPLE code, generate UMPLE code from diagrams you draw yourself, or generate executable code in various programming languages corresponding to your UMPLE code or UML diagrams. It's a great tool for exploring UML and class diagrams, but we don't recommend using the Ruby code it generates, which is non-DRY and somewhat non-idiomatic.

Self-Check 11.2.1

In a UML class diagram depicting the relationship "University has many Departments," what multiplicities would be allowable on each side of the association?

◇ The University side has multiplicity `1`, because a Department must belong to exactly one University. The Department side has multiplicity `1..*`, because one or more Departments can belong to a University. ■

Competency 11.2.2

Determine whether an aggregation or composition is more appropriate for modeling a

LCOM variant	Scores	Interpretation
Revised Henderson-Sellers LCOM	0 (best) to 1 (worst)	0 means all instance methods access all instance variables. 1 means any given instance variable is used by only one instance method, that is, the instance methods are fairly independent of each other.
LCOM-4	1 (best) to n (worst)	Estimates number of responsibilities n in your class as the number of connected components in a graph in which related methods' nodes are connected by an edge. $n > 1$ suggests that up to $n - 1$ responsibilities could be extracted into their own classes.

Figure 11.7: The "recommended" lack of cohesion of methods (LCOM) score depends heavily on which LCOM variant is used. The table shows two of the most widely-used variants.

specific one-to-many relationship.

Competency 11.2.3

Given a simple UML class diagram showing entities and relationships such as has-many, identify the relationships depicted.

11.3 Single Responsibility Principle

The ***Single Responsibility Principle*** (SRP) of SOLID states that a class should have one and only one responsibility—that is, only one reason to change. For example, in Section 5.2, when we added single sign-on to RottenPotatoes, we created a new `SessionsController` to handle the sign-on interaction. An alternate strategy would be to augment `MoviegoersController`, since sign-on is an action associated with moviegoers. Indeed, before the single sign-on approach described in Chapter 5, this was the recommended way to implementing password-based authentication in earlier versions of Rails. But such a scheme would require changing the `Moviegoer` model and controller whenever we wanted to change the authentication strategy, even though the "essence" of a Moviegoer doesn't really depend on how they sign in. In MVC, each controller should specialize in dealing with one resource; an authenticated user session is a distinct resource from the user himself, and deserves its own RESTful actions and model methods. As a rule of thumb, if you cannot describe the responsibility of a class in 25 words or less, it may have more than one responsibility, and the new ones should be split out into their own classes.

In statically typed compiled languages, the cost of violating SRP is obvious: any change to a class requires recompilation and may also trigger recompilation or relinking of other classes that depend on it. Because we don't pay this price in interpreted dynamic languages, it's easy to let classes get too large and violate SRP. One tip-off is lack of ***cohesion***, which is the degree to which the elements of a single logical entity, in this case a class, are related. Two methods are related if they access the same subset of instance or class variables or if one calls the other. The *LCOM* metric, for *Lack of Cohesion Of Methods*, measures cohesion for a class: in particular, it warns you if the class consists of multiple "clusters" in which methods within a cluster are related, but methods in one cluster aren't strongly related to methods in other clusters. Figure 11.7 shows two of the most commonly used variants of the LCOM metric.

In Section 9.6, after successfully refactoring `convert`, `reek` reported "low cohesion" in the `TimeSetter` class because we used class variables rather than instance variables for maintaining what was actually instance state, as that section described.

The *Data Clumps* design smell is one warning sign that a good class is evolving toward the "multiple responsibilities" antipattern. A Data Clump is a group of variables or values that are always passed together as arguments to a method or returned together as a set of

https://gist.github.com/a100b89babb3e1dfc3f0fbbe98fcd820

```
class Moviegoer
  attr_accessor :name, :street, :phone_number, :zipcode
  validates :phone_number, # ...
  validates :zipcode, # ...
  def format_phone_number ; ... ; end
  def check_zipcode ; ... ; end
  def format_address(street, phone_number, zipcode) # data clump
    # do formatting, calling format_phone_number and check_zipcode
  end
end
# After applying Extract Class:
class Moviegoer
  attr_accessor :name
  has_one :address
end
class Address
  belongs_to :moviegoer
  attr_accessor :phone_number, :zipcode
  validates :phone_number, # ...
  validates :zipcode, # ...
  def format_address ; ... ; end # no arguments - operates on 'self'
  private  # no need to expose these now:
  def format_phone_number ; ... ; end
  def check_zipcode ; ... ; end
end
```

Figure 11.8: To perform Extract Class, we identify the group of methods that shares a responsibility distinct from that of the rest of the class, move those methods into a new class, make the "traveling together" data items on which they operate into instance variables of the class, and arrange to pass an instance of the class around rather than the individual items.

results from a method. This "traveling together" is a sign that the values might really need their own class. Another symptom is that something that used to be a "simple" data value acquires new behaviors. For example, suppose a `Moviegoer` has attributes `phone_number` and `zipcode`, and you want to add the ability to check the zip code for accuracy or canonicalize the formatting of the phone number. If you add these methods to `Moviegoer`, they will reduce its cohesion because they form a "clique" of methods that only deal with specific instance variables. The alternative is to use the *Extract Class* refactoring to put these methods into a new `Address` class, as Figure 11.8 shows.

Summary of Single Responsibility Principle:

- A class should have one and only one reason to change, that is, one responsibility.
- A poor LCOM (Lack of Cohesion Of Methods) score and the Data Clump design smell are both warnings of possible SRP violations. The Extract Class refactoring can help remove and encapsulate additional responsibilities in a separate class.

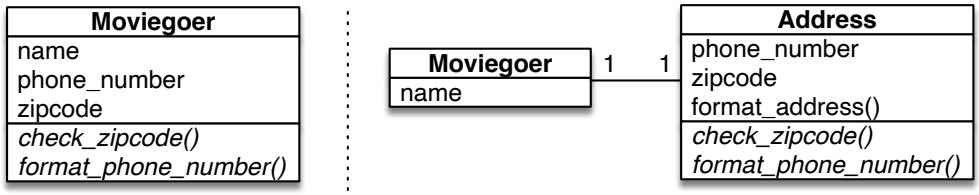

Figure 11.9: UML class diagrams before (left) and after (right) extracting the `Address` class from `Moviegoer`.

■ ***Elaboration: Interface Segregation Principle***

Related to SRP is the ***Interface Segregation Principle*** (ISP, and the original **I** in SOLID), which states that if a class's API is used by multiple quite different types of clients, the API probably should be segregated into subsets useful to each type of client. For example, the `Movie` class might provide both movie metadata (MPAA rating, release date, and so on) and an interface for searching TMDb, but it's unlikely that a client using one of those two sets of services would care about the other. The problem solved by ISP arises in compiled languages in which changes to an interface require recompiling the class, thereby triggering recompilation or relinking of classes that use that interface. While documenting separate interfaces for distinct sets of functionality is good style, ISP rarely arises in Ruby since there are no compiled classes, so we won't discuss it further.

Self-Check 11.3.1

Draw the UML class diagrams showing class architecture before and after the refactoring in Figure 11.8.

◇ Figure 11.9 shows the UML diagrams. ■

Competency 11.3.2

Draw UML class diagrams to represent the "before and after" states of a refactoring that changes class structure.

11.4 Open/Closed Principle

The ***Open/Closed Principle*** (OCP) of SOLID states that classes should be "open for extension, but closed against modification." That is, it should be possible to extend the behavior of classes without modifying existing code on which other classes or apps depend.

While adding subclasses that inherit from a base class is one way to extend existing classes, it's often not enough by itself. Figure 11.10 shows why the presence of `case`-based dispatching logic—one variant of the *Case Statement* design smell—suggests a possible OCP violation.

Depending on the specific case, various design patterns can help. One problem that the smelly code in Figure 11.10 is trying to solve is that the desired subclass of `Formatter` isn't known until runtime, when it is stored in the `@format` instance variable. The ***abstract factory pattern*** provides a common interface for instantiating an object whose subclass may not be known until runtime. Ruby's duck typing and metaprogramming enable a particularly elegant implementation of this pattern, as Figure 11.11 shows. (In statically-typed languages,

https://gist.github.com/ce84aab55f40bdc7566e84a7512ede27

```
class Report
  def output
    formatter =
      case @format
      when :html
        HtmlFormatter.new(self)
      when :pdf
        PdfFormatter.new(self)
        # ...etc
      end
  end
end
```

Figure 11.10: The `Report` class depends on a base class `Formatter` with subclasses `HtmlFormatter` and `PdfFormatter`. Because of the explicit dispatch on the report format, adding a new type of report output requires modifying `Report#output`, and probably requires changing other methods of `Report` that have similar logic—so-called *shotgun surgery*.

https://gist.github.com/4d15a866491b82e0b6f19f1e97d5601b

```
class Report
  def output
    formatter_class =
      begin
        @format.to_s.classify.constantize
      rescue NameError
        # ...handle 'invalid formatter type'
      end
    formatter = formatter_class.send(:new, self)
    # etc
  end
end
```

Figure 11.11: Ruby's metaprogramming and duck typing enable an elegant implementation of the abstract factory pattern. `classify` is provided by Rails to convert `snake_case` to `UpperCamelCase`. `constantize` is syntactic sugar provided by Rails that calls the Ruby introspection method `Object#const_get` on the receiver. We also handle the case of an invalid value of the formatter class, which the bad code doesn't.

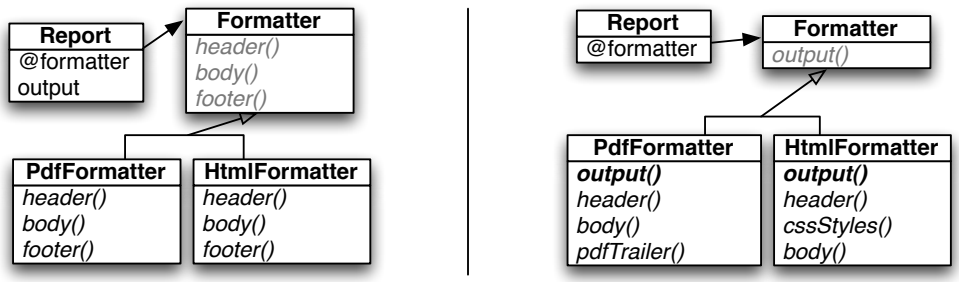

Figure 11.12: In Template Method (left), the extension points are `header`, `body`, and `footer`, since the `Report#output` method calls `@formatter.header`, `@formatter.body`, and so on, each of which delegates to a specialized counterpart in the appropriate subclass. (Light gray type indicates methods that just delegate to a subclass.) In Strategy (right), the extension point is the `output` method itself, which delegates the entire task to a subclass. Delegation is such a common ingredient of composition that some people refer to it as the *delegation pattern*.

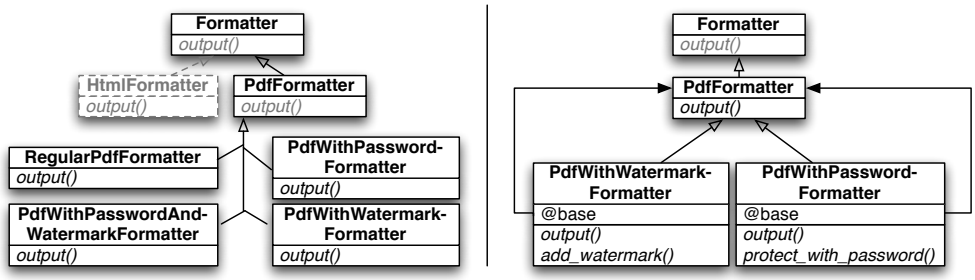

Figure 11.13: (Left) The multiplication of subclasses resulting from trying to solve the Formatter problem using inheritance shows why your class designs should "prefer composition over inheritance." (Right) A more elegant solution uses the Decorator design pattern.

to "work around" the type system, we have to create a factory method for each subclass and have them all implement a common interface—hence the name of the pattern.)

Another approach is to take advantage of the ***Strategy pattern*** or ***Template Method pattern***. Both support the case in which there is a general approach to doing a task but many possible variants. The difference between the two is the level at which commonality is captured. With Template Method, although the implementation of each step may differ, the set of steps is the same for all variants; hence it is usually implemented using inheritance. With Strategy, the overall task is the same, but the set of steps may be different in each variant; hence it is usually implemented using composition. Figure 11.12 shows how either pattern could be applied to the report formatter. If every kind of formatter followed the same high-level steps—for example, generate the header, generate the report body, and then generate the footer—we could use Template Method. On the other hand, if the steps themselves were quite different, it would make more sense to use Strategy.

An example of the Strategy pattern in the wild is OmniAuth (Section 5.2): many apps need third-party authentication, and the steps are quite different depending on the auth provider, but the API to all of them is the same. Indeed, OmniAuth even refers to its plug-ins as "strategies."

A different kind of OCP violation arises when we want to *add* behaviors to an existing class and discover that we cannot do so without modifying it. For example, PDF files can be generated with or without password protection and with or without a "Draft" watermark across the background. Both features amount to "tacking on" some extra behavior to what `PdfFormatter` already does. If you've done a lot of object-oriented programming, your first thought might therefore be to solve the problem using inheritance, as the UML diagram in Figure 11.13 (left) shows, but there are four permutations of features so you'd end up with four subclasses with duplication across them—hardly DRY. Fortunately, the ***decorator pattern*** can help: we "decorate" a class or method by wrapping it in an enhanced version that has the same API, allowing us to compose multiple decorations as needed. Figure 11.14 shows the code corresponding to the more elegant decorator-based design of the PDF formatter shown in Figure 11.13 (right).

Python's "decorators"[8] are, unfortunately, completely unrelated to the Decorator design pattern.

In the wild, the ActiveSupport module of Rails provides method-level decoration via `alias_method_chain`, which is very useful in conjunction with Ruby's open classes, as Figure 11.15 shows. A more interesting example of Decorator in the wild is the Rack application server we've been using since Chapter 3. The heart of Rack is a "middleware" module that receives an HTTP request and returns a three-element array consisting of an

https://gist.github.com/4f46456d38c0dfb5743481ae76acb83d

```
class PdfFormatter
  def initialize ; ... ; end
  def output ; ... ; end
end
class PdfWithPasswordFormatter < PdfFormatter
  def initialize(base) ; @base = base ; end
  def protect_with_password(original_output) ; ... ; end
  def output ; protect_with_password @base.output ; end
end
class PdfWithWatermarkFormatter < PdfFormatter
  def initialize(base) ; @base = base ; end
  def add_watermark(original_output) ; ... ; end
  def output ; add_watermark @base.output ; end
  end
end
# If we just want a plain PDF
formatter = PdfFormatter.new
# If we want a "draft" watermark
formatter = PdfWithWatermarkFormatter.new(PdfFormatter.new)
# Both password protection and watermark
formatter = PdfWithWatermarkFormatter.new(
  PdfWithPasswordFormatter.new(PdfFormatter.new))
```

Figure 11.14: To apply Decorator to a class, we "wrap" class by creating a subclass (to follow the Liskov Substitution Principle, as we'll learn in Section 11.5). The subclass delegates to the original method or class for functionality that isn't changed, and implements the extra methods that extend the functionality. We can then easily "build up" just the version of `PdfFormatter` we need by "stacking" decorators.

https://gist.github.com/d6e8a3aeb60516338d38ee95478be708

```
# reopen Mailer class and decorate its send_email method.
class Mailer
  alias_method_chain :send_email, :cc
  def send_email_with_cc(recipient,body) # this is our new method
    send_email_without_cc(recipient,body) # will call original method
    copy_sender(body)
  end
end
# now we have two methods:
send_email(...)            # calls send_email_with_cc
send_email_with_cc(...)    # same thing
send_email_without_cc(...) # call (renamed) original method
```

Figure 11.15: To decorate an existing method `Mailer#send_email`, we reopen its class and use `alias_method_chain` to decorate it. Without changing any classes that call `send_email`, all calls now use the decorated version that sends email and copies the sender.

HTTP response code, HTTP headers, and a response body. A Rack-based application specifies a "stack" of middleware components that all requests traverse: to add a behavior to an HTTP request (for example, to intercept certain requests as OmniAuth does to initiate an authentication flow), we decorate the basic HTTP request behavior. Additional decorators add support for SSL (Secure Sockets Layer), measuring app performance, and some types of HTTP caching.

Summary of Open/Closed Principle:

- To make a class open for extension but closed against modification, we need mechanisms that enable specific *extension points* at places we think extensions might be needed in the future. The *Case Statement* design smell is one symptom of a possible OCP violation.
- If the extension point takes the form of a task with varying implementations for the steps, the Strategy and Template Method patterns may apply. Both are often used in conjunction with the Abstract Factory pattern, since the variant to create may not be known until runtime.
- If the extension point takes the form of selecting different subsets of features that "add on" to existing class behaviors, the Decorator pattern may apply. The Rack application server is designed this way.

■ ***Elaboration: Closed against what?***

"Open for extension but closed against modification" presupposes that you know in advance what the useful extension points will be, so you can leave the class open for the "most likely" changes and strategically close it against changes that might break its dependents. In our example, since we already had more than one way to do something (format a report), it seemed reasonable to allow additional formatters to be added later, but you don't always know in advance what extension points you'll want. Make your best guess, and deal with change as it comes.

Self-Check 11.4.1

Here are two statements about delegation:

1. *A subclass delegates a behavior to an ancestor class*
2. *A class delegates a behavior to a descendant class*

Looking at the examples of the Template Method, Strategy, and Decorator patterns (Figures 11.12 and 11.13), which statement best describes how each pattern uses delegation?

◇ In Template Method and Strategy, the ancestor class provides the "basic game plan" which is customized by delegating specific behaviors to different subclasses. In Decorator, each subclass provides special functionality of its own, but delegates back to the ancestor class for the "basic" functionality. ■

Competency 11.4.2

Identify potential violations of OCP in a class structure, and propose one or more refac-

https://gist.github.com/cca588f67ed62d38e96094fff68a5418

```
class Rectangle
  attr_accessor :width, :height, :top_left
  def initialize(width,height,top_left) ... ; end
  def area ... ; end
  def perimeter ... ; end
end
# A square is just a special case of rectangle...right?
class Square < Rectangle
  # ooops...a square has to have width and height equal
  attr_reader :width, :height, :side
  def width=(w)  ; @width = @height = w ; end
  def height=(w) ; @width = @height = w ; end
  def side=(w)   ; @width = @height = w ; end
end
# But is a Square really a kind of Rectangle?
class Rectangle
  def make_twice_as_wide_as_high(dim)
    self.width = 2*dim
    self.height = dim           # doesn't work!
  end
end
```

Figure 11.16: Behaviorally, rectangles have some capabilities that squares don't have—for example, the ability to set the lengths of their sides independently, as in `Rectangle#make_twice_as_wide_as_high`.

> *torings that would resolve them, and specifically, whether the Template/Strategy pattern or Decorator pattern may apply.*

11.5 Liskov Substitution Principle

The ***Liskov Substitution Principle*** (LSP) is named for Turing Award winner Barbara Liskov, who did seminal work on subtypes that heavily influenced object-oriented programming. Informally, LSP states that a method designed to work on an object of type T should also work on an object of any subtype of T. That is, all of T's subtypes should preserve T's "contract."

This may seem like common sense, but it's subtly easy to get wrong. Consider the code in Figure 11.16, which suffers from an LSP violation. You might think a `Square` is just a special case of `Rectangle` and should therefore inherit from it. But *behaviorally,* a square is *not* like a rectangle when it comes to setting the length of a side! When you spot this problem, you might be tempted to override `Rectangle#make_twice_as_wide_as_high` within `Square`, perhaps raising an exception since this method doesn't make sense to call on a `Square`. But that would be a *refused bequest*—a design smell that often indicates an LSP violation. The symptom is that a subclass either destructively overrides a behavior inherited from its superclass or forces changes to the superclass to avoid the problem (which itself should indicate a possible OCP violation). The problem is that inheritance is all about implementation sharing, but if a subclass won't take advantage of its parent's implementations, it might not deserve to be a subclass at all.

The fix, therefore, is to again use composition and delegation rather than inheritance, as Figure 11.17 shows. Happily, because of Ruby's duck typing, this use of composition and delegation still allows us to pass an instance of `Square` to most places where a `Rectangle` would be expected, even though it's no longer a subclass; a statically-typed language would have to introduce an explicit interface capturing the operations common to both `Square` and

https://gist.github.com/8c1a10862521a34e274c3f1cfa5c24d8

```
1  # LSP-compliant solution: replace inheritance with delegation
2  # Ruby's duck typing still lets you use a square in most places where
3  #  rectangle would be used - but no longer a subclass in LSP sense.
4  class Square
5    attr_accessor :rect
6    def initialize(side, top_left)
7      @rect = Rectangle.new(side, side, top_left)
8    end
9    def area      ; rect.area      ; end
10   def perimeter ; rect.perimeter ; end
11   # A more concise way to delegate, if using ActiveSupport (see text):
12   #  delegate :area, :perimeter, :to => :rect
13   def side=(s) ; rect.width = rect.height = s ; end
14 end
```

Figure 11.17: As with some OCP violations, the problem arises from a misuse of inheritance. As Figure 11.18 shows, preferring composition and delegation to inheritance fixes the problem. Line 12 shows a concise syntax for delegation available to apps using ActiveSupport (and all Rails apps do); similar functionality for non-Rails Ruby apps is provided by the `Forwardable` module in Ruby's standard library.

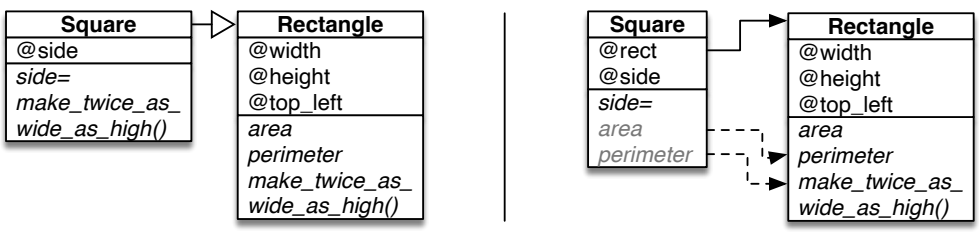

Figure 11.18: Left: The UML class diagram representing the original LSP-violating code in Figure 11.16, which destructively overrides `Rectangle#make_twice_as_wide_as_high`. Right: the class diagram for the refactored LSP-compliant code in Figure 11.17.

`Rectangle`.

Summary of the Liskov Substitution Principle:

- LSP states that a method that operates on objects of some class should also work correctly on objects of any subclass of that class. When a subclass differs behaviorally from one of its parents, an LSP violation can arise.
- The *refused bequest* design smell, in which a subclass destructively overrides a parent behavior or forces changes to the parent class so that the behavior is not inherited—often signals an LSP violation.
- Many LSP violations can be fixed by using composition of classes rather than inheritance, achieving reuse through delegation rather than through subclassing.

Self-Check 11.5.1

Why is `Forwardable` in the Ruby standard library provided as a module rather than a class?

◇ Modules allow the delegation mechanisms to be mixed in to any class that wants to use them, which would be awkward if `Forwardable` were a class. That is, `Forwardable` is itself an example of preferring composition to inheritance! ■

Competency 11.5.2

Identify a potential LSP violation in the implementation of a class and its subclass(es).

Competency 11.5.3

Resolve an LSP violation by replacing inheritance with composition, showing the UML class diagrams of the resulting refactoring.

11.6 Dependency Injection Principle

The dependency injection principle (DIP), sometimes also called dependency inversion, states that if two classes depend on each other but their implementations may change, it would be better for them to both depend on a separate abstract interface that is "injected" between them.

Suppose RottenPotatoes now adds email marketing—interested moviegoers can receive emails with discounts on their favorite movies. RottenPotatoes integrates with the external email marketing service MailerMonkey to do this job:

https://gist.github.com/c722647142f471b8bb7806055a8c4765

```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize
5      @mailer = MailerMonkey.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10   moviegoers = Moviegoer.interested_in params[:movie_id]
11   EmailList.new.send_email_to moviegoers
12 end
```

The feature is so successful that you decide to extend the mechanism so that moviegoers who are on the Amiko social network can opt to have these emails forwarded to their Amiko friends as well, using the new `Amiko` gem that wraps Amiko's RESTful API for messaging friends. There are two problems, however.

First, `EmailList#initialize` has a hardcoded dependency on `MailerMonkey`, but now we will sometimes need to use `Amiko` instead. This runtime variation is the problem solved by dependency injection—since we won't know until runtime which type of mailer we'll need, we modify `EmailList#initialize` so we can "inject" the correct value at runtime:

https://gist.github.com/c9a3235dbf7be72738dcb4911ccb18e9

```
1  class EmailList
2    attr_reader :mailer
3    delegate :send_email, :to => :mailer
4    def initialize(mailer_type)
5      @mailer = mailer_type.new
6    end
7  end
8  # in RottenPotatoes EmailListController:
9  def advertise_discount_for_movie
10   moviegoers = Moviegoer.interested_in params[:movie_id]
11   mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
12   EmailList.new(mailer).send_email_to moviegoers
13 end
```

You can think of DIP as injecting an additional seam between two classes, and indeed, in statically compiled languages DIP helps with testability. This benefit is less apparent in Ruby, since as we've seen we can create seams almost anywhere we want at runtime using mocking or stubbing in conjunction with Ruby's dynamic language features.

The second problem is that `Amiko` exposes a different and more complex API than the simple `send_email` method provided by `MailerMonkey` (to which `EmailList#send_email` delegates in line 3), yet our controller method is already set up to call `send_email` on the mailer object. The ***Adapter pattern*** can help us here: it's designed to convert an existing API into one that's compatible with an existing caller. In this case, we can define a new class `AmikoAdapter` that converts the more complex `Amiko` API into the simpler one that our controller expects, by providing the same `send_email` method that `MailerMonkey` provides:

ActiveRecord has been criticized for configuring the database at startup from `database.yml` rather than using DIP. Presumably the designers judged that the database wouldn't change while the app was running. While DIP-induced seams also help with stubbing and mocking, Chapter 8 shows that Ruby's open classes and metaprogramming let you insert test seams wherever needed.

https://gist.github.com/3533bdccbcc7bcd3fb36b7f18fc79862

```
class Config
  def self.email_enabled? ; ... ; end
  def self.emailer ; if has_amiko? then Amiko else MailerMonkey end ; end
end
def advertise_discount_for_movie
  if Config.email_enabled?
    moviegoers = Moviegoer.interested_in(params[:movie_id])
    EmailList.new(Config.emailer).send_email_to(moviegoers)
  end
end
```

https://gist.github.com/4b333ccb9cbf7f5de75c1ca9d94e7ad6

```
class Config
  def self.emailer
    if email_disabled? then NullMailer else
      if has_amiko? then Amiko else MailerMonkey end
    end
  end
end
class NullMailer
  def initialize ; end
  def send_email_to(*args) ; true ; end
end
def advertise_discount_for_movie
  moviegoers = Moviegoer.interested_in(params[:movie_id])
  EmailList.new(Config.emailer).send_email_to(moviegoers)
  end
end
```

Figure 11.19: Top: a naive way to disable a behavior is to "condition it out" wherever it occurs. Bottom: the Null Object pattern eliminates the conditionals by providing "dummy" methods that are safe to call but don't do anything.

https://gist.github.com/411c249b2ba9e9f04613df32bc8862c1

```
class AmikoAdapter
  def initialize ; @mailer = Amiko.new(...) ; end
  def send_email
    @mailer.authenticate(...)
    @mailer.send_message(...)
  end
end
# Change the controller method to use the adapter:
def advertise_discount_for_movie
  moviegoers = Moviegoer.interested_in params[:movie_id]
  mailer = if Config.has_amiko? then AmikoAdapter else MailerMonkey end
  EmailList.new(mailer).send_email_to moviegoers
end
```

When the Adapter pattern not only converts an existing API but also simplifies it—for example, the `Amiko` gem also provides many other Amiko functions unrelated to email, but `AmikoAdapter` only "adapts" the email-specific part of that API—it is sometimes called the ***Façade pattern***.

Lastly, even in cases where the email strategy is known when the app starts up, what if we want to disable email sending altogether from time to time? Figure 11.19 (top) shows a naive approach: we have moved the logic for determining which emailer to use into a new `Config` class, but we still have to "condition out" the email-sending logic in the controller method if email is disabled. But if there are other places in the app where a similar check must be performed, the same condition logic would have to be replicated there (shotgun surgery). A better alternative is the ***Null Object pattern***, in which we create a "dummy" object that has all the same behaviors as a real object but doesn't do anything when those behaviors are

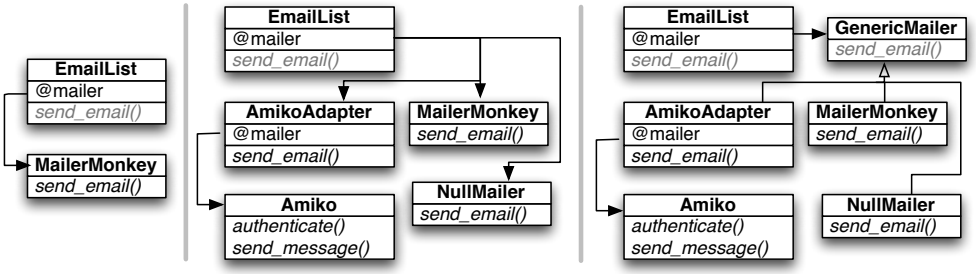

Figure 11.20: Left: Without dependency injection, `EmailList` depends directly on `MailerMonkey`. Center: With dependency injection, `@mailer` can be set at runtime to use any of `MailerMonkey`, `NullMailer` (which implements the Null Object pattern to disable email), or `AmikoAdapter` (which implements the Adapter/Façade pattern over `Amiko`), all of which have the same API. Right: In statically typed languages, the abstract superclass `GenericMailer` formalizes the fact that all three mailers have compatible APIs, but in Ruby this superclass is often omitted if it consists entirely of abstract methods (as is the case here), since abstract methods and classes aren't part of the language.

called. Figure 11.19 (bottom) applies the Null Object pattern to this example, avoiding the proliferation of conditionals throughout the code.

Figure 11.20 shows the UML class diagrams corresponding to the various versions of our DIP example.

An interesting relative of the Adapter and Façade patterns is the ***Proxy pattern***, in which one object "stands in" for another that has the same API. The client communicates with the proxy instead of the original object; the proxy may forward some requests directly to the original object (that is, delegate them) but may take other actions on different requests, perhaps for reasons of performance or efficiency.

Two classic examples of this pattern are found in ActiveRecord itself. First, the object returned by ActiveRecord's `all`, `where` and `find`-based methods quacks like a collection, but it's actually a proxy object that doesn't even do the query until you force the issue by asking for one of the collection's elements. That is why you can build up complex queries with multiple `where`s without paying the cost of doing the query each time. The second is when you use ActiveRecord's associations (Section 5.4): the result of evaluating `@movie.reviews` quacks like an enumerable collection, but it's actually a proxy object that responds to all the collection methods (`size`, `<<`, and so on), without querying the database except when it has to.

Another example of the proxy pattern in SaaS is the service worker of a progressive Web app, as Section 6.10 describes: the service worker transparently intercepts the main app's HTTP requests and can behave differently depending on whether the user's device is connected to the network.

Summary of Dependency Injection:

- Dependency injection inserts a seam between two classes by passing in (injecting) a dependency whose value may not be known until runtime, rather than hardwiring a dependency into the source code.
- Because dependency injection is often used to vary which of a collection of implementations is used at runtime, it's often seen together with the Adapter pattern, in which a class converts one API into another that a client expects to use.
- Variations on Adapter include Façade, in which the API is not only adapted but also simplified, and Proxy, in which the API is exactly imitated but the behaviors changed to accommodate different usage conditions without the client (caller of the API) having to change its behavior.
- The Null Object pattern is another mechanism for replacing unwieldy conditionals with safe "neutral" behaviors as a way of disabling a feature.

■ ***Elaboration: Did injecting a dependency violate the Open/Closed Principle?***

You might wonder whether our "fix" to add a second type of mailer service violates OCP, because adding support for a third mailer would then require modifying `advertise_discount_for_movie`. If you had reason to believe you might indeed need to add additional mailers later, you could combine this with the Abstract Factory pattern introduced in Section 11.4. This scenario is an example of making a judgment call about whether the possibility of handling additional mailers is an extension point you want to leave open, or a change you feel the app wouldn't accommodate well and should therefore be strategically closed against.

Self-Check 11.6.1

Why does proper use of DIP have higher impact in statically typed languages?

◇ In such languages, you cannot create a runtime seam to override a "hardwired" behavior as you can in dynamic languages like Ruby, so the seam must be provided in advance by injecting the dependency. ■

11.7 Demeter Principle

The name comes from the Demeter Project on adaptive and aspect-oriented programming, which in turn is named for the Greek goddess of agriculture to signify a "from the ground up" approach to programming.

The Demeter Principle or ***Law of Demeter*** states informally: "Talk to your friends—don't get intimate with strangers." Specifically, a method can call other methods in its own class, and methods on the classes of its own instance variables; everything else is taboo. Demeter isn't originally part of the SOLID guidelines, as Figure 11.4 explains, but we include it here since it is highly applicable to Ruby and SaaS, and we opportunistically hijack the **D** in SOLID to represent it.

The Demeter Principle is easily illustrated by example. Suppose RottenPotatoes has made deals with movie theaters so that moviegoers can buy movie tickets directly via RottenPotatoes by maintaining a credit balance (for example, by receiving movie theater gift cards).

Figure 11.21 shows an implementation of this behavior that contains a Demeter Principle

https://gist.github.com/d3ea5309672163c64674b21033c9106c

```
1  # This example is adapted from Dan Manges's blog, dcmanges.com
2  class Wallet ; attr_accessor :credit_balance ; end
3  class Moviegoer
4    attr_accessor :wallet
5    def initialize
6      # ...setup wallet attribute with correct credit balance
7    end
8  end
9  class MovieTheater
10   def collect_money(moviegoer, due_amount)
11     # VIOLATION OF DEMETER (see text)
12     if moviegoer.wallet.credit_balance < due_amount
13       raise InsufficientFundsError
14     else
15       moviegoer.wallet.credit_balance -= due_amount
16       @collected_amount += due_amount
17     end
18   end
19 end
20 # Imagine testing the above code:
21 describe MovieTheater do
22   describe "collecting money" do
23     it "should raise error if moviegoer can't pay" do
24       # "Mock trainwreck" is a warning of a Demeter violation
25       wallet = double('wallet', :credit_balance => 5.00)
26       moviegoer = double('moviegoer', :wallet => wallet)
27       expect { @theater.collect_money(moviegoer, 10.00) }.
28         to raise_error(...)
29     end
30   end
31 end
```

Figure 11.21: Line 12 contains a Demeter violation: while it's reasonable for `MovieTheater` to know about `Moviegoer`, it also knows about the implementation of `Wallet`, since it "reaches through" the `wallet` attribute to manipulate the wallet's `credit_balance`. Also, we're handling the problem of "not enough cash" in `MovieTheater`, even though logically it seems to belong in `Wallet`.

violation. A problem arises if we ever change the implementation of `Wallet`—for example, if we change `credit_balance` to `cash_balance`, or add `points_balance` to allow moviegoers to accumulate PotatoPoints by becoming top reviewers. All of a sudden, the `MovieTheater` class, which is "twice removed" from `Wallet`, would have to change.

Two design smells can tip us off to possible Demeter violations. One is ***inappropriate intimacy***: the `collect_money` method manipulates the `credit_balance` attribute of `Wallet` directly, even though managing that attribute is the `Wallet` class's responsibility. (When the same kind of inappropriate intimacy occurs repeatedly throughout a class, it's sometimes called ***feature envy***, because `Moviegoer` "wishes it had access to" the features managed by `Wallet`.) Another smell that arises in tests is the *mock trainwreck*, which occurs in lines 25–27 of Figure 11.21: to test code that violates Demeter, we find ourselves setting up a "chain" of mocks that will be used when we call the method under test.

Once again, delegation comes to the rescue. A simple improvement comes from delegating the `credit_balance` attribute, as Figure 11.22 (top) shows. But the best delegation is that in Figure 11.22 (bottom), since now the behavior of payment is entirely encapsulated within `Wallet`, as is the decision of when to raise an error for failed payments.

Inappropriate intimacy and Demeter violations can arise in any situation where you feel you are "reaching through" an interface to get some task done, thereby exposing yourself to dependency on implementation details of a class that should really be none of your business. Three design patterns address common scenarios that could otherwise lead to Demeter violations. One is the Visitor pattern, in which a data structure is traversed and you provide a callback method to execute for each member of the data structure, allowing you to "visit" each element while remaining ignorant of the way the data structure is organized. Indeed, the "data structure" could even be materialized lazily as you visit the different nodes, rather than existing statically all at once. An example of this pattern in the wild is the Nokogiri[9] gem, which supports traversal of HTML and XML documents organized as a tree: in addition to searching for a specific element in a document, you can have Nokogiri traverse the document and call a visitor method you provide at each document node.

A simple special case of Visitor is the ***Iterator pattern***, which is so pervasive in Ruby (you use it anytime you use `each`) that many Rubyists hardly think of it as a pattern. Iterator separates the implementation of traversing a collection from the behavior you want to apply to each collection element. Without iterators, the behavior would have to "reach into" the collection, thereby knowing inappropriately intimate details of how the collection is organized.

Observer was first implemented in the MVC framework of ***Smalltalk***, from which Ruby inherits its object model.

The last design pattern that can help with some cases of Demeter violations is the ***Observer pattern***, which is used when one class (the observer) wants to be kept aware of what another class is doing (the subject) without knowing the details of the subject's implementation. The Observer design pattern provides a canonical way for the subject to maintain a list of its observers and notify them automatically of any state changes in which they have indicated interest, using a narrow interface to separate the concept of observation from the specifics of what each observer does with the information.

While the Ruby standard library includes a mixin[10] called `Observable`, Rails' ActiveSupport provides a more concise Observer that lets you observe any model's ActiveRecord lifecycle hooks (`after_save` and so on), introduced in Section 5.1. Figure 11.23 shows how easy it is to add an `EmailList` class to RottenPotatoes that "subscribes" to two kinds of state changes:

1. When a new review is added, it emails all moviegoers who have already reviewed that

https://gist.github.com/6a636b98e53b84c9628eba411d3d4086

```
# Better: delegate credit_balance so MovieTheater only accesses Moviegoer
class Moviegoer
  def credit_balance
    self.wallet.credit_balance  # delegation
  end
end
class MovieTheater
  def collect_money(moviegoer,due_amount)
    if moviegoer.credit_balance >= due_amount
      moviegoer.credit_balance -= due_amount
      @collected_amount += due_amount
    else
      raise InsufficientFundsError
    end
  end
end
```

https://gist.github.com/8a6859cdd248ab77673f3eb5373e9a67

```
class Wallet
  attr_reader :credit_balance # no longer attr_accessor!
  def withdraw(amount)
     raise InsufficientFundsError if amount > @credit_balance
     @credit_balance -= amount
     amount
  end
end
class Moviegoer
  # behavior delegation
  def pay(amount)
    wallet.withdraw(amount)
  end
end
class MovieTheater
  def collect_money(moviegoer, amount)
    @collected_amount += moviegoer.pay(amount)
  end
end
```

Figure 11.22: (Top) If `Moviegoer` delegates `credit_balance` to its `wallet`, `MovieTheater` no longer has to know about the implementation of `Wallet`. However, it may still be undesirable that the payment *behavior* (subtract payment from credit balance) is exposed to `MovieTheater` when it should really be the responsibility of `Moviegoer` or `Wallet` only. (Bottom) Delegating the behavior of payment, rather than the attributes through which it's accomplished, solves the problem and eliminates the Demeter violation.

https://gist.github.com/7d58ab75f108b42105b6c198040f831b

```
class EmailList
  observe Review
  def after_create(review)
    moviegoers = review.moviegoers # from has_many :through, remember?
    self.email(moviegoers, "A new review for #{review.movie} is up.")
  end
  observe Moviegoer
  def after_create(moviegoer)
    self.email([moviegoer], "Welcome, #{moviegoer.name}!")
  end
  def self.email ; ... ; end
end
```

Figure 11.23: An email list subsystem observes other models so it can generate email in response to certain events. The Observer pattern is an ideal fit since it collects all the concerns about when to send email in one place.

same movie.

2. When a new moviegoer signs up, it sends her a "Welcome" email.

In addition to ActiveRecord lifecycle hooks, Rails caching, which we will encounter in Chapter 12, is another example of the Observer pattern in the wild: the cache for each type of ActiveRecord model observes the model instance in order to know when model instances become stale and should be removed from the cache. The observer doesn't have to know the implementation details of the observed class—it just gets called at the right time, like Iterator and Visitor.

To close out this section, it's worth pointing out an example that looks like it violates Demeter, but really doesn't. It's common in Rails views (say, for a `Review`) to see code such as:

https://gist.github.com/c5ddf39643f6763c5b7ec377a8cdd5e7

```
<p> Review of:  <%= @review.movie.title %>    </p>
<p> Written by: <%= @review.moviegoer.name %> </p>
```

Aren't these Demeter violations? It's a judgment call: strictly speaking, a `review` shouldn't know the implementation details of `movie`, but it's hard to argue that creating delegate methods `Review#movie_title` and `Review#moviegoer_name` would enhance readability in this case. The general opinion in the Rails community is that it's acceptable for views whose purpose is to display object relationships to also expose those relationships in the view code, so examples like this are usually allowed to stand.

Summary of Demeter Principle:

- The Demeter Principle states that a class shouldn't be aware of the details of collaborator classes from which it is further away than "once removed." That is, you can access instance methods in your own class and in the classes corresponding to your nearest collaborators, but not on *their* collaborators.
- The Inappropriate Intimacy design smell, which sometimes manifests as a Mock Trainwreck in unit tests, may signal a Demeter violation. If a class shows many instances of Inappropriate Intimacy with another class, it is sometimes said to have Feature Envy with respect to the other class.
- Delegation is the key mechanism for resolving these violations.
- Design patterns cover some common manipulations of classes without violating Demeter, including Iterator and Visitor (separating traversal of an aggregate from behavior) and Observer (separating notification of "interesting" events from the details of the class being observed).

■ ***Elaboration: Observers, Visitors, Iterators, and Mixins***

Because of duck typing and mixins, Ruby can express many design patterns with far less code than statically-typed languages, as the Wikipedia entries for ***Observer***, ***Iterator*** and ***Visitor*** clearly demonstrate by using Java-based examples. In contrast to Ruby's internal iterators based on `each`, statically-typed languages usually provide external iterators and visitors in which you set up the iterator over a collection and ask the iterator explicitly whether the collection has any more elements, sometimes requiring various contortions to work around the type system. Similarly, Observer usually requires modifying the subject class(es) so that they can implement an `Observable` interface, but Ruby's open classes allow us to skip that step, as Figure 11.23 showed: from the programmer's point of view, all of the logic is in the observing class, not the subjects.

Self-Check 11.7.1

Ben Bitdiddle is a purist about Demeter violations, and he objects to the expression `@movie.reviews.average_rating` in the movie details view, which shows a movie's average review score. How would you placate Ben and fix this Demeter violation?

https://gist.github.com/77f741c6c441b81be9c3220bf01a05ee

```
# naive way:
class Movie
  has_many :reviews
  def average_rating
    self.reviews.average_rating # delegate to Review#average_rating
  end
end
# Rails shortcut:
class Movie
  has_many :reviews
  delegate :average_rating, :to => :review
end
```

■

Self-Check 11.7.2

Notwithstanding that "delegation is the key mechanism" for resolving Demeter violations, why should you be concerned if you find yourself delegating many methods from class A to class B just to resolve Demeter violations present in class C?

◇ You might ask yourself whether there should be a direct relationship between class C and class B, or whether class A has "feature envy" for class B, indicating that the division of responsibilities between A and B might need to be reengineered. ■

11.8 The Plan-And-Document Perspective on Design Patterns

A strength of Plan-and-Document is that careful upfront planning can result in a product with a good software architecture that uses design patterns well. This preplanning is reflected in the alternative catch phrase for these processes of ***Big Design Up Front***, as Chapter 1 mentions.

A Plan-and-Document development team starts with the ***Software Requirements Specification*** (***SRS***) (see Section 7.10), which the team breaks into a series of problems. For each one, the team looks for one or more architecture patterns that might solve the problem. The team then goes down to the next level of subproblems, and looks for design patterns

that match them. The philosophy is to learn from the experience of others captured as patterns so as to avoid repeating the mistakes of your predecessors. Another way to get feedback from more experienced engineers is to hold a ***design review*** (see Section 10.7). Note that design reviews can be done before any code is written in Plan-and-Document processes.

Thus, compared to Agile, there is considerably more effort in starting with a good design in Plan-and-Document. As Martin Fowler points out in his article *Is Design Dead?*[11], a frequent critique of Agile is that it encourages developers to jump in and start coding without any design, and rely too much on refactoring to fix things later. As the critics sometimes say, you can build a doghouse by slapping stuff together and planning as you go, but you can't build a skyscraper that way.

Agile supporters counter that Plan-and-Document methods are just as bad: by disallowing any code until the design is complete, it's impossible to be confident that the design will be implementable or that it really captures the customer's needs. This critique especially holds when the architects/designers will not be writing the code or may be out of touch with current coding practices and tools. As a result, say Agile proponents, when coding starts, the design will have to change anyway.

Both sides have a point, but the critique can be phrased in a more nuanced way as "How much design makes sense up front?" For example, Agile developers plan for persistent storage as part of their SaaS apps, even though the first BDD and TDD tests they write will not touch the database. A more subtle example is horizontal scaling. As we alluded to in Chapter 3, and will discuss more fully in Chapter 12, designers of successful SaaS *must* think about horizontal scalability early on. Even though it may be months before scalability matters, design decisions early in the project can cripple scalability, and it may be difficult to change them without major rewriting and refactoring.

A possible solution to the conundrum is captured by a rule of thumb in Fowler's article. If you have previously done a project that has some design constraint or element, it's OK to plan for it in a new project that is similar, because your previous experience will likely lead to reasonable design decisions this time.

Summary: Plan-and-Document processes have an explicit design phase that is a natural fit to the use of design patterns in the software development process. One potential drawback is uncertainty as to whether the initial architecture and design patterns will need to change as the code is written and as the system evolves. In contrast, the Agile process relies on refactoring to incorporate design patterns as the code evolves, although experienced developers may lay plans for software architectures and design patterns that they expect to need based on previous, similar projects.

Self-Check 11.8.1

True or False: Agile design is an oxymoron.

◇ False. Although there is no separate design phase in Agile development, the refactoring that is the norm in Agile can incorporate design patterns. ■

11.9 6S: A Clean Code Checklist

A key message of this book is that when you write code, you're writing for other developers who will maintain it after you've moved on. The many tools and techniques we

introduced—software architectures such as model-view-controller and client-server, class-level and method-level design patterns, and tools for measuring and improving code quality through refactoring—are all there to help enhance code beauty and therefore maintainability.

While there is no fixed recipe for creating beautiful code (other than lots of practice), the information in this book on refactoring (Chapter 9) and design patterns (this chapter) can go a long way towards helping you produce beautiful code, and the frequent code reviews that are part of the pull request process (Chapter 10) can serve as a further cross-check. You have also learned in multiple contexts that while automated code quality tools are useful, they are no substitute for engineering judgment, a solid understanding of the codebase, and a team whose members trust and rely on each other for constructive criticism.

To help pull all of this advice together, in this section we present a "checklist" you and your team can use to evaluate your code. Since Agile is all about iterative refinement in the pursuit of higher code quality, we will call it the "6S List," which we hope you will remember because it almost sounds like "success list." Here is our proposed 6S list, ordered from highest to lowest level of abstraction, to use as a checklist before opening a pull request: Site, SOLID, SOFA, Smells, Style, and Sign-off.

The name is also a nod to ***Six Sigma***, a technique for manufacturing-process improvement that uses statistical methods to find and remediate process problems that harm output quality.

Site. If you're adding new code, is it in the right place architecturally? Does it properly belong directly in a model, view, controller, or helper? Or, at least as often, should it be separately modularized as a service object, form object, or other type of code (Section 5.8), with clear and explicit dependencies on other existing classes? Is the new code connected in the proper way to the rest of the app? For example, JavaScript code must be appropriately loaded by the asset pipeline, and may require DOM bindings (Section 6.4) on particular app pages.

SOLID. If you're adding new classes or adding new code to existing classes, really scrutizing your code against each of the SOLID principles (Sections 11.3–11.7) can reveal risk areas in which you are introducing unnecessary technical debt—changes that will make the code harder to understand, maintain, or modify later. This is the moment to identify possible refactorings, even simple ones such as extracting a class (Section 11.3). Automated tools such as CodeClimate can help identify trouble spots. Once technical debt is incurred and gets into the mainline codebase, it is *extremely resistant* to being paid back, since new features or other work always seem to take priority.

SOFA. Just as with SOLID, make an effort to critically apply each of the four SOFA criteria (Section 9.5) to any new or refactored methods within each class. Is each method **S**hort, because it does only **O**ne thing requiring **F**ew arguments while maintaining a single level of **A**bstraction? Just as one of the most frequent violations of SOLID is violating the Single Responsibility Principle, one of the most frequent violations of SOFA is violating "Methods should be **S**hort." Both violations are usually symptomatic of other violations likely lurking in the code.

Smells. Is the code relatively free of code and design smells (Chapters 9 and 11)? Tools like CodeClimate can find both language-independent smells (high cyclomatic complexity, deeply nested conditionals, meaningless variable names, arguments that travel around as a group) and language-specific or framework-specific problems (overly long controller actions, use of potentially confusing language idioms). Again, use automated tools in tandem with good engineering judgment, not as a substitute for it.

Style. Most languages and frameworks have low-level best practices regarding variable and function naming (Section 2.3), spacing and indentation, punctuation, and so on. Where

such practices are unambiguous, your code should follow them; where there is flexibility in the practices, your code should follow the existing conventions of the codebase. All modern editors allow customizing the rules for automatic indentation and spacing, use of spaces rather than tabs, and so on; configure your editor to match the codebase's conventions.

Sign-off. When you've checked all of the above, the last step is to open a pull request and invite team feedback until the code passes your team's review with "LGTM" ("looks good to me") or similar. Automated tools are wonderful, but the eyes and brains of your team can produce codebase-specific suggestions beyond what automated tools are able to do.

Summary:

- The six S's—site, SOLID, SOFA, smells, style, sign-off—can help you check your own work as you get more proficient at creating beautiful code.
- Automated tools can help with some of the S's, but in the end, there's no substitute for an informed team who trust and hold each other responsible for providing constructive feedback. Keeping the codebase clean is everyone's job!

■ ***Elaboration: Guidelines or Rules?***

In a 2013 presentation[12] at the Barcelona Ruby Conference, Rubyist Sandi Metz, the author of one of the books we recommend on object-oriented design (Metz 2012), proposes a set of rules based on a seventh S, *size*. Her proposed rules are:

- Classes must be no longer than 100 lines
- Methods must be no longer than 5 lines, and take no more than 4 arguments
- Controller actions may name at most 2 other classes, and may set at most 1 instance variable to be consumed by the view

The rules may only be broken if you can convince your pair programming partner that it makes sense to do so in a given situation. Metz notes that while these thresholds are arbitrary, at least they force developers to think about *why* they might need to break the rules or whether they should refactor. We largely agree with her rules (even if the thresholds are a bit tight), and we hope the 6S approach will help identify *why* the rules are about to be broken and provide guidance for how to improve.

11.10 Fallacies and Pitfalls

Pitfall: **Over-reliance or under-reliance on patterns.**

As with every tool and methodology we've seen, slavishly following design patterns is a pitfall: they can help point the way when your problem could take advantage of a proven solution, but they cannot by themselves ensure beautiful code. In fact, the GoF authors specifically warn *against* trying to evaluate the soundness of a design based on the number of patterns it uses. In addition, if you apply design patterns too early in your design cycle, you may try to implement a pattern in its full generality even though you may not need that generality for solving the current problem. That will complicate your design because most

design patterns call for *more* classes, methods, and levels of indirection than the same code would require without this level of generality. In contrast, if you apply design patterns too late, you risk falling into antipatterns and extensive refactoring.

What to do? Develop taste and judgment through learning by doing. You will make some mistakes as you go, but your judgment on how to deliver working and maintainable code will quickly improve.

Pitfall: **Over-reliance on UML or other diagrams.**

A diagram's purpose is communication of intent. Reading UML diagrams is not necessarily easier than reading user stories or well-factored TDD tests. Create a diagram when it helps to clarify a class architecture; don't rely on them as a crutch.

Fallacy: **SOLID principles aren't needed in dynamic languages.**

As we saw in this chapter, some of the problems addressed by SOLID don't really arise in dynamically-typed languages like Ruby. Nonetheless, the SOLID guidelines still represent good design; in static languages, there is simply a much more tangible up-front cost to ignoring them. In dynamic languages, while the opportunity exists to use dynamic features to make your code more elegant and DRY without the extra machinery required by some of the SOLID guidelines, the corresponding risk is that it's easier to fall into sloth and end up with ugly antipattern code.

Pitfall: **Lots of private methods in a class.**

You may have already discovered that methods declared `private` are hard to test, because by definition they can only be called from within an instance method of that class—meaning they cannot be called directly from an RSpec test. Although you can use a hack to temporarily make the method public (`MyClass.send(:public,:some_private_method)`), private methods complex enough to need their own tests should be considered a smell: the methods themselves may be too long, violating the **S**hort guideline of SOFA, and the class containing these methods may be violating the ***Single Responsibility Principle***. In this case, consider extracting a collaborator class whose methods are public (and therefore easy to test and easy to shorten by refactoring) but are only called from the original class, thereby improving maintainability and testability.

Pitfall: **Using `initialize` to implement factory patterns.**

In Section 11.4, we showed an example of Abstract Factory pattern in which the correct subclass constructor is called directly. Another common scenario is one in which you have a class `A` with subclasses `A1` and `A2`, and you want calls to `A`'s constructor to return a new object of the correct subclass. You usually cannot put the factory logic into the `initialize` method of `A`, because that method must by definition return an instance of class `A`. Instead, give the factory method a different name such as `create`, make it a class method, and call it from `A`'s constructor:

https://gist.github.com/f6cd62078436064577a5773617fcccd4

```
class A
  def self.create(subclass, *args) # subclass must be either 'A1' or 'A2'
    return Object.const_get(subclass).send(:new, *args)
  end
end
```

11.11 Concluding Remarks: Frameworks Capture Design Patterns

> *The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.*
>
> —Donald Knuth

The idea of design patterns is inspired by Christopher Alexander's 1977 book *A Pattern Language: Towns, Buildings, Construction* describing design patterns for civil architecture. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the "Gang Of Four" or GOF) published the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1995 (Gamma et al. 1994). It described what are now called the 23 GoF Design Patterns focusing on class-level structures and behaviors. The original 23 design patterns from the Gang of Four have been expanded dramatically since their book appeared. There are numerous repositories of design patterns (Cunningham 2013; Noble and Johnson 2013), with some tailored to specific problem areas such as user interfaces (Griffiths 2013; Toxboe 2013).

Despite design patterns' popularity as a tool, they have been the subject of some critique; for example, Peter Norvig, currently Google's Director of Research, has argued that some design patterns just compensate for deficiencies in statically-typed programming languages such as C++ and Java, and that the need for them disappears in dynamic languages such as Lisp or Ruby. Notwithstanding some controversy, patterns of many kinds remain a valuable way for software engineers to identify structure in their work and bring proven solutions to bear on recurring problems.

A problem for novice developers is that even if you read the Gang of Four book or study these repositories, it is hard to know which pattern to apply. If you don't have previous experience with a given design pattern, and you try to design for it in an anticipatory manner, you're more likely to get it wrong, so you should instead wait to add it later when and if it's really needed.

The good news is that frameworks like Rails encapsulate others' design experience to provide abstractions and design constraints that have been proven through reuse. For example, it may not occur to you to design your app's actions around REST, but it turns out that doing so results in a design that is more consistent with the scalability success stories of the Web. While the Gang of Four went out of their way to differentiate design patterns from frameworks to try to make it clear what design patterns are—more abstract, narrower in focus, and not targeted to a problem domain—today frameworks are a great way for a novice to get started with design patterns. By examining the patterns in a framework that are instantiated as code, you can gain experience on how to create your own code based on design patterns.

Design Patterns (Gamma et al. 1994) is the classic Gang of Four text on design patterns. While canonical, it's a bit slower reading than some other sources, and the examples are heavily oriented to C++. *Design Patterns in Ruby* (Olsen 2007) treats a subset of the GoF

Smell	Description	Fix
Comment deodorant, inappropriate name	Obfuscated variable or method names make lots of comments necessary	Reduce need for comments through descriptive names and (as necessary) by addressing other smells within the offending code
Lazy class, data class	A class does too little, for example, providing nothing but getters and setters for some object but no other logic	Merge methods that encapsulate the data object into another class
Duplicated code, combinatorial explosion	Nearly the same code repeated with subtle changes in multiple methods, in same class	Extract common parts using DRY mechanisms like blocks and `yield` (Section 2.3), extracting helper methods (Section 9.6), using Template or Strategy design pattern (Section 11.4)
Parallel inheritance hierarchy	Nearly the same code repeated with subtle changes in different classes that inherit from different ancestors; for example, numerous pieces of code using slightly different combinations of data or behavior	Extract commonality into its own class and delegate to that class (Section 11.7). If classes with different ancestors need the functionality, try extracting it into a module that can be mixed in

Figure 11.24: Some smells are relatively easily fixed by a local modification. These are excerpted from Fowler's *Refactoring, Ruby Edition* (Fields et al. 2009).

patterns in detail showing Ruby examples. It also discusses patterns made unnecessary by Ruby language features. *Clean Code* (Martin 2008) has a more thorough exposition of both the SOFA and SOLID guidelines that motivate the use of design patterns.

Rather than presenting a "laundry list" of patterns, we tried to motivate a subset of patterns by showing the design smells they fix. *Rails Antipatterns* (Pytel and Saleh 2010) gives great examples of how real-life code that starts with a good design can become cluttered over time, and how to beautify and streamline it by refactoring, often using one or more of the appropriate design patterns. Figure 11.24 shows a few examples of those refactorings, largely drawn from Martin Fowler's online catalog of refactorings[13] and comprehensive book (Fields et al. 2009).

W. Cunningham. Portland pattern repository, 2013. URL `http://c2.com/ppr/`.

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508.

E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN 0201633612.

R. Griffiths. HCI design patterns, 2013. URL `http://www.hcipatterns.org/patterns`.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer (Addison-Wesley Professional Ruby)*. Addison-Wesley Professional, 2012. ISBN 0321721330. URL `http://poodr.com`.

J. Noble and R. Johnson. Design patterns library, 2013. URL `http://hillside.net/patterns`.

R. Olsen. *Design Patterns in Ruby*. Addison-Wesley Professional, 2007. ISBN 9780321490452.

C. Pytel and T. Saleh. *Rails AntiPatterns: Best Practice Ruby on Rails Refactoring (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2010. ISBN 9780321604811.

A. Toxboe. UI patterns, 2013. URL `http://ui-patterns.com/`.

Notes

[1] `http://cleancoder.com`
[2] `http://martinfowler.com/eaaCatalog`
[3] `http://ui-patterns.com`
[4] `http://en.wikipedia.org/wiki/Unified_Modeling_Language`
[5] `http://en.wikipedia.org/wiki/Unified_Modeling_Language`
[6] `http://cruise.site.uottawa.ca/umple/`
[7] `http://try.umple.org`
[8] `http://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators`
[9] `http://nokogiri.org`
[10] `http://www.ruby-doc.org/stdlib-1.9.3/libdoc/observer/rdoc/Observable.html`
[11] `http://www.martinfowler.com/articles/designDead.html`
[12] `https://youtu.be/npOGOmkxuio`
[13] `http://martinfowler.com/refactoring/catalog`