Chapter 2. Reading Code

Code is read much more often than it is written.

Guido van Rossum, creator of Python

Despite the way coding is taught, software engineers spend far more time reading code than writing it. In most beginner coding courses, you jump immediately into writing code, focusing on core language concepts and idioms without acknowledging that you'd never learn Polish or Portuguese in a similar manner. And while most academic projects start from a blank slate, practicing developers are almost always working within the confines of code that has taken years to arrive at its current state, something you can explore in depth in Chapter 6.

With the advent of agentic or chat-oriented programming,[1] reading code will become even *more* important for software engineers. While it may not be your first choice, you will work with code you did not write. Take heart, there *are* techniques to help you orient yourself when you encounter unfamiliar code. This chapter will go over why reading code can be challenging, and we'll give some tips to make the process simpler.

The Challenge of Working with Existing Code

Regardless of how you learned to code, you probably spent much of your time in the blissful space known as *greenfield development*, where you experience the job of starting from scratch, unencumbered by the baggage of prior work. Yet, in your professional life, you've likely had vanishingly few opportunities to build an application from a blank editor. As a practicing software engineer, much of your time will be spent on *brownfield development*, working within the limits of an existing codebase dealing day in and day out with legacy code.

Legacy Code by Any Other Name…

Reading old code is often a task most developers prefer to avoid. We often use the term *legacy code* to describe it, and this term is rarely meant as a compliment. There are any number of definitions of legacy code: code that was written yesterday, code without adequate test coverage or too much, or just code you didn't write.

However, you shouldn't disparage the success of an existing application. If a product has delivered business value for years and has justified continued investment, that is worthy of a pat on the back. We prefer a more positive frame, such as *heritage code* or *existing code*. For a more in-depth discussion of the topic, see Chapter 6.

When asked to solve a problem that has to do with existing code, you actually have *four* problems to solve.

First and foremost, you need to understand the business problem you are trying to solve. And the domains that developers work in are very demanding! Software is eating the world, meaning software engineers are tasked with increasingly more complex business challenges; much of the proverbial low-hanging fruit has already been picked.

Second, you must see the problem through the eyes of the developer who came before you, and that is often the most challenging aspect of software development. Different developers solve problems in different ways; everyone has their habits and tendencies. Perhaps your predecessor had a predilection for a more functional style of programming that you aren't as familiar with. Maybe they used a pattern that is new to you or leveraged a library that you haven't used before. Whether you agree with the approach taken or not, there are many ways to solve a given problem.

Third, it's possible the existing code doesn't have the right level of abstraction. Perhaps it is modeled in a way that is too generic or fails to capture the proper nuance of the domain. Maybe the previous developer conflated two concepts or forced a favored pattern when another would have been more appropriate. Regardless, a larger refactoring may be required to make the code more understandable.

Fourth, much like an archaeologist, you are often peeling back layer after layer of technical debt, old technologies and approaches that may now be considered anti-patterns. Over time, languages and best practices evolve,[2] and you will have to see the code through the lens of when it was written. You may even be able to "carbon date" the code simply by noticing what frameworks or language features are (or are not) used! Frameworks like Spring have evolved over the course of many years, often supporting multiple approaches to a given problem. What is considered the "right" way to do something changes over time.

It also doesn't help that you almost always have to deal with patches on top of patches. Maybe the last developer didn't have a full understanding of the problem, or they weren't up to speed on some new language feature that could greatly simplify the job at hand. Maybe the last developer had a premium license to an AI coding assistant and generated 20,000 lines of code per day. Add in the typical demands of fix-it-fast, and you could spend an afternoon deciphering a single method.

Working with existing code presents technical challenges, making it understandable why it's often one of a developer's least favorite tasks. However, the aversion extends beyond technical issues to include cognitive biases as well.

Cognitive Biases

When reading existing code, you might compare it unfavorably to your own work. Of course, *you* don't write bad code, do you? On more than one occasion, we, your humble authors, have struggled with some code, uttering less polite variations of "What idiot wrote this?" only to discover that it was written by none other than ourselves. And frankly, if you read code you wrote a few years ago, you *should* be a little disappointed—that's a sign of growth; you know more today than you did then. That is a good thing!

You also have a couple of cognitive biases working against you when you work with existing code. First is the IKEA effect, which is when you place a higher value on things you create. One study found people would pay 63% more for a product they successfully assembled themselves versus the identical product put together by someone else.[3] If you've ever gone to a pick-your-own apple orchard, you are often charged a premium to, well, do some of the work yourself. You'll do it, however, for the experience and the chance to select the very best fruit right off the trees. In software, developers often have strong opinions about the "right" way to do things and tend to prefer their own code and approach.

Additionally, there is the mere-exposure effect: you tend to prefer the things you are already familiar with. This leads to the typical dogmatism many developers have around programming languages. Developers tend to think time began with whatever language they learned first. When Java first introduced Lambda expressions, someone on a language-specific mailing list asked why Java needed these "new-fangled Lambdas," not realizing Lambdas are not a new concept in programming languages and were part of the original plan for Java itself!

Developers can be provincial around their preferred tools, which is something Paul Graham touches on in his essay "Beating the Averages". Graham says programming languages exist on a power continuum, but you often can't recognize *why* a language is more powerful than another. To demonstrate his point, he introduced the hypothetical Blub language and a productive Blub programmer. When the Blub programmer looks down the power continuum, all they see are languages that lack features they use every day, and they can't understand why anyone would choose such an inferior tool. When they look up the power continuum, all they see are a bunch of weird features they don't have in Blub, and they can't imagine why anyone would need those to be productive since they aren't in Blub.

As you work with code, as well as with your fellow developers, keep these biases in mind. If you aren't sure why a colleague is so adamant about a certain tool or approach, ask if it might be an instance of the IKEA effect or the Blub paradox. Of course, you should also reflect on your own assumptions to ensure you aren't exhibiting one of the predispositions yourself.

Approaching Unfamiliar Code

As much as you may wish you could spend all your work hours focused on crafting new code, you will encounter existing codebases throughout your career. How can you get up to speed on a new project without losing your mind? First, start with your teammates. A basic project overview should be part of any onboarding experience.

Spend some time with the documentation. Many projects have a README file that will help you get your bearings, while others have wikis or websites designed to give you a concise overview.[4] Projects may also have architecture decision records (ADRs).[5] ADRs provide invaluable context and the all-important "why" that often vanishes in the rush of the latest defect or outage. You could learn more in a few minutes with the docs than in hours with the debugger. Reading the project's coding standards will prepare you for the patterns you will encounter as you wade through the codebase.

If the documentation for the code you're reading is out-of-date, update it as you learn; if it is nonexistent, consider building your own as you go. Apply the Golden Rule (discussed in Chapter 1). Creating the documentation will help you learn the project, and it will also serve the developers who come after you.

There are any number of things you could document. As you read, write your documentation to favor lightweight, low-ceremony approaches seeking to answer common questions such as these:

- What does your service do?
- How does it work?
- What does it depend on?
- How do you run the application?

Wait, we can hear you now: documentation may be (and often is) out of sync with the code. But believe us, that doesn't have to be the case. Documentation can evolve with the code. The best way to ensure that it does is to use tests as documentation. Tests written with behavior-driven styles, if written properly, can produce executable documentation, a topic discussed at greater length in Chapter 5.

Metrics Can Mislead

Code coverage (how much of the codebase is executed when the tests are run) can be a useful metric on a project. However, there are no silver bullets in software, and it is possible to fail even with 100% code coverage. A friend of ours joined a project that was having regressions with every release. As he was getting up to speed on the code, he asked the tech lead if there were any tests. The tech lead proudly said, "Yes, we have right around 92% code coverage." Impressed, our friend was somewhat surprised they had so many regressions, but he continued his analysis.

Looking at the test code, he found some startling patterns. At first, he thought these were isolated, but eventually he discovered they were endemic to the codebase. He went back to the tech lead and said, "I couldn't help but notice your tests don't have any asserts." The tech lead responded by repeating the code coverage statistic.

The meta lesson is: be wary of any metric, because they can mislead. But don't lose sight of the value and purpose of a practice. If it is just about ceremony, you are unlikely to get the benefit you expect. Project teams should regularly challenge themselves and their approach; don't be afraid to change course when warranted.

Software Archeology

Once you've surveyed the team and familiarized yourself with any existing documentation, it is time to open your editor of choice and practice some software archeology. Roll up your sleeves and root around in the codebase! To paraphrase Sir Issac Newton, look for smoother pebbles and prettier shells. Look at the code structure—how is the code organized? Some languages have first-class constructs for packaging code; others rely on conventions. How does the code fit together? Is this a monolith or a distributed architecture with dozens or hundreds of services? What domain concepts are expressed in the code? Read the tests—what do they tell you about the functionality? With that information, do you understand the *intent* of this class?

If the intent isn't clear, dig further. Modern editors can make it trivially simple to see who calls a given function, allowing you to work your way backward. Callers should help you determine what a given class does and how it is used. Your backtracking may take you all the way to a service endpoint like an HTTP call, but eventually you should find the connection between a given user action and the code.

Once you have your bearings, run the application. What does it do? Find a specific element, be it something on a user interface or a parameter to a service call, and map that back to the code. Look at the issue list; see if focusing on a single feature or bug allows you to follow the coding path. Hunt for a landmark; if you know a given action results in an update to the datastore, find that in the code. Use your debugger to walk through the code—did it work the way you anticipated? Did you end up on a vastly different code path? Ultimately, you are building a mental model of the code; you are loading the application into your brain.

What might this look like in practice? Let's use the Spring PetClinic app as an example. Even if you aren't an expert in Java or Spring, navigating the app should be fairly straightforward, plus it has excellent documentation. Once you've cloned and run the application, you'll see that it has the ability to search for owners (see Figure 2-1). If you explore the owner templates, you'll see one helpfully named *findOwners.html*,[6] which references an `/owners` endpoint. Searching the project for `/owners` returns ample results, but a little intuition might lead you to the `@GetMapping("/owners")` annotation on the `processFindForm` method in the `OwnerController` file.

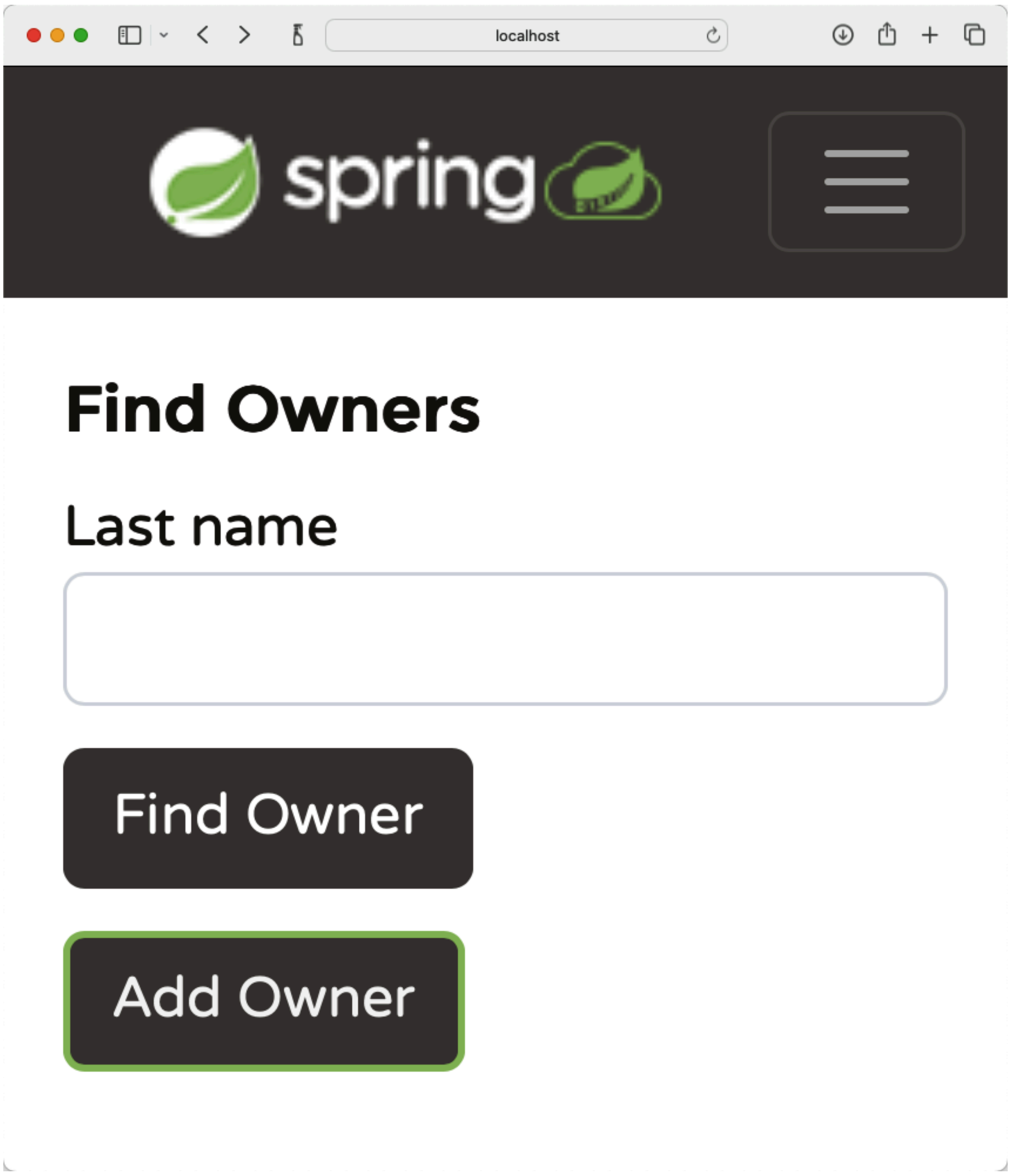
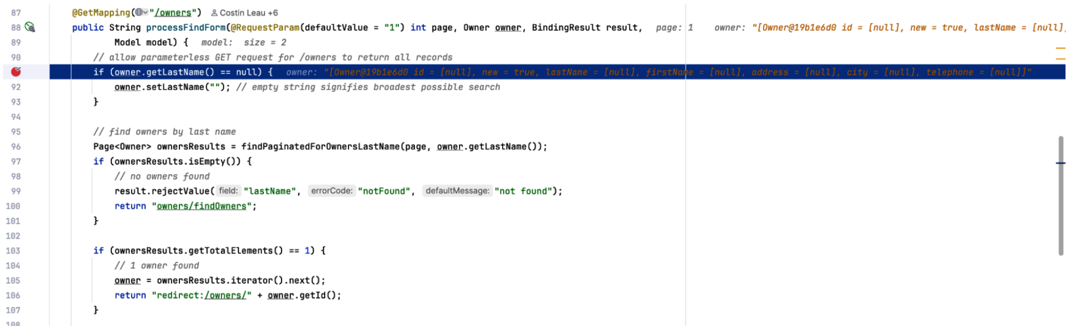


Figure 2-1. Spring PetClinic Find Owners page

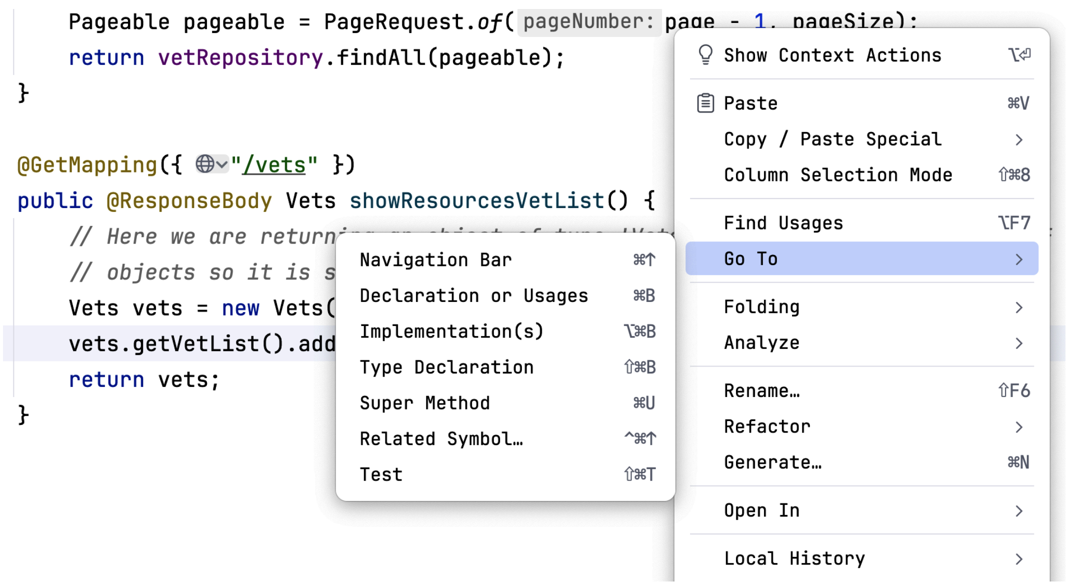
Put in a breakpoint, execute the search from the browser, and see what happens! Sure enough, your debugger should look something like Figure 2-2. If your intuition was wrong? Repeat the previous steps. Eventually, you will make the connection, allowing you to walk your way through the code and build your understanding as you go.



```
@GetMapping("/owners")  Costin Leau +6
public String processFindForm(@RequestParam(defaultValue = "1") int page, Owner owner, BindingResult result,   page: 1    owner: "[Owner@19b1e6d0 id = [null], new = true, lastName = [null],
        Model model) {   model:  size = 2
    // allow parameterless GET request for /owners to return all records
    if (owner.getLastName() == null) {   owner: "[Owner@19b1e6d0 id = [null], new = true, lastName = [null], firstName = [null], address = [null], city = [null], telephone = [null]]"
        owner.setLastName(""); // empty string signifies broadest possible search
    }

    // find owners by last name
    Page<Owner> ownersResults = findPaginatedForOwnersLastName(page, owner.getLastName());
    if (ownersResults.isEmpty()) {
        // no owners found
        result.rejectValue( field: "lastName",  errorCode: "notFound",  defaultMessage: "not found");
        return "owners/findOwners";
    }

    if (ownersResults.getTotalElements() == 1) {
        // 1 owner found
        owner = ownersResults.iterator().next();
        return "redirect:/owners/" + owner.getId();
    }
```

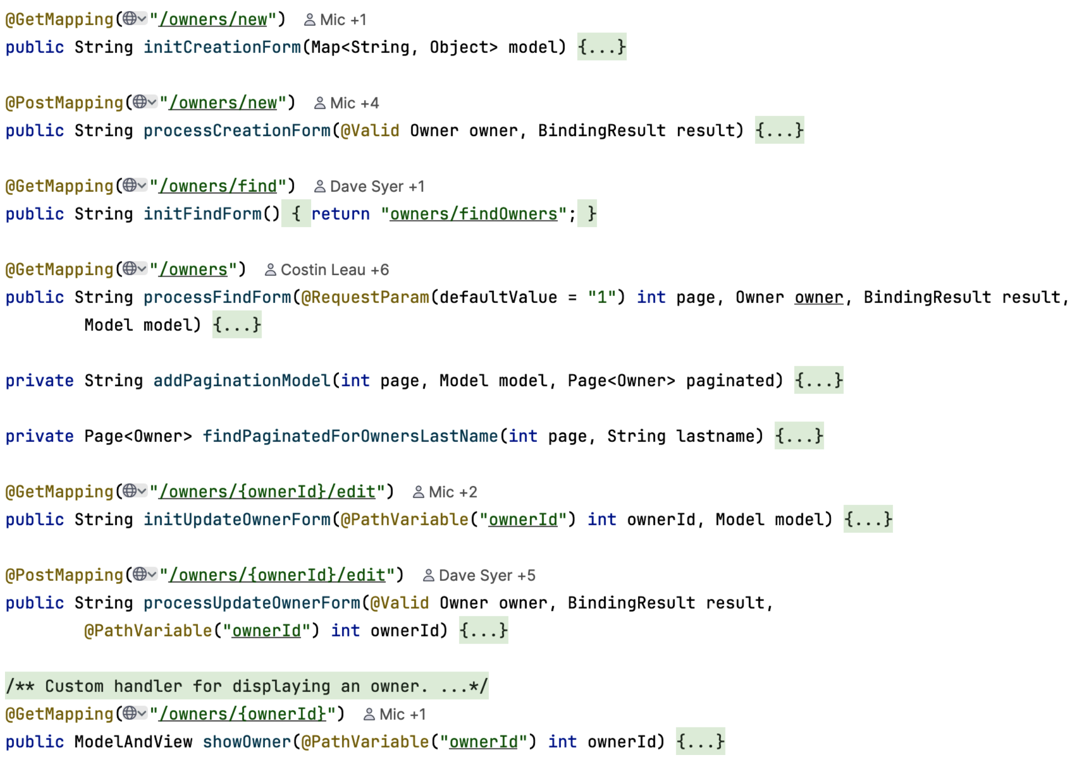
Figure 2-2. A breakpoint in `OwnerController` allows you to inspect the current state of the application

Use your integrated development environment (IDE) to navigate the code. Many IDEs make it easy to jump to methods in other classes (see Figure 2-3) as you work your way through the code. Consider collapsing all the method bodies to give you a smaller surface area to peruse (see Figure 2-4). Alternatively, many IDEs can show you an outline of a given class, providing a higher-level view of the code. Read the method names. What does that tell you about the purpose of the module?



```
    Pageable pageable = PageRequest.of( pageNumber: page - 1, pageSize);
    return vetRepository.findAll(pageable);
}

@GetMapping({ "/vets" })
public @ResponseBody Vets showResourcesVetList() {
    // Here we are returning an object of type 'Vets'
    // objects so it is s
    Vets vets = new Vets(
    vets.getVetList().add
    return vets;
}
```

Figure 2-3. Use your IDE to help you quickly navigate code



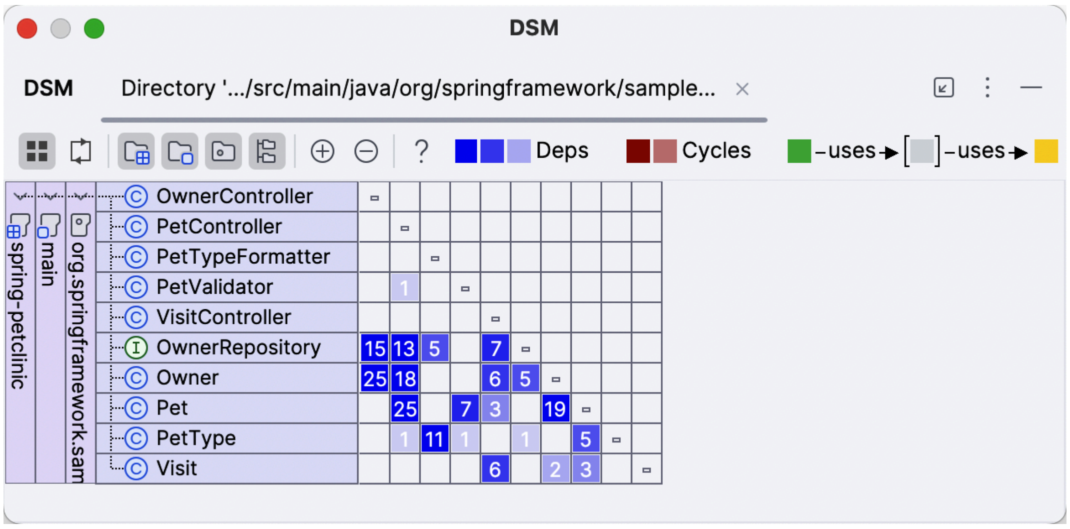
```
@GetMapping("/owners/new")   Mic +1
public String initCreationForm(Map<String, Object> model) {...}

@PostMapping("/owners/new")   Mic +4
public String processCreationForm(@Valid Owner owner, BindingResult result) {...}

@GetMapping("/owners/find")   Dave Syer +1
public String initFindForm() { return "owners/findOwners"; }

@GetMapping("/owners")   Costin Leau +6
public String processFindForm(@RequestParam(defaultValue = "1") int page, Owner owner, BindingResult result,
        Model model) {...}

private String addPaginationModel(int page, Model model, Page<Owner> paginated) {...}

private Page<Owner> findPaginatedForOwnersLastName(int page, String lastname) {...}

@GetMapping("/owners/{ownerId}/edit")   Mic +2
public String initUpdateOwnerForm(@PathVariable("ownerId") int ownerId, Model model) {...}

@PostMapping("/owners/{ownerId}/edit")   Dave Syer +5
public String processUpdateOwnerForm(@Valid Owner owner, BindingResult result,
        @PathVariable("ownerId") int ownerId) {...}

/** Custom handler for displaying an owner. ...*/
@GetMapping("/owners/{ownerId}")   Mic +1
public ModelAndView showOwner(@PathVariable("ownerId") int ownerId) {...}
```

Figure 2-4. Collapse methods to orient yourself in the codebase

Do not assume the code does what the name implies. As code evolves, variable and method names may no longer reflect reality. Don't rush; confirm your hunches. Resist the temptation to cut corners and potentially create more work later on. In other words, take the time it takes, so it takes less time.

Exceptions can mislead. More than once, we have encountered exceptions that made incorrect assumptions about possible error conditions. Pay extra care to situations that catch very high-level exceptions; while expedient for the author, they tend to obfuscate the possible problems.

Your IDE may also include tools or plug-ins that help you analyze the code. For example, IntelliJ IDEA can quickly show you dependencies, giving you a sense of how the code works together (see Figure 2-5). Modern developer tools are powerful; let them help you understand the code.


DSM

DSM Directory '.../src/main/java/org/springframework/sample...

🟦🟦🟦 Deps 🟥🟥 Cycles 🟩–uses→[ ]–uses→🟨

spring-petclinic / main / org.springframework.sam

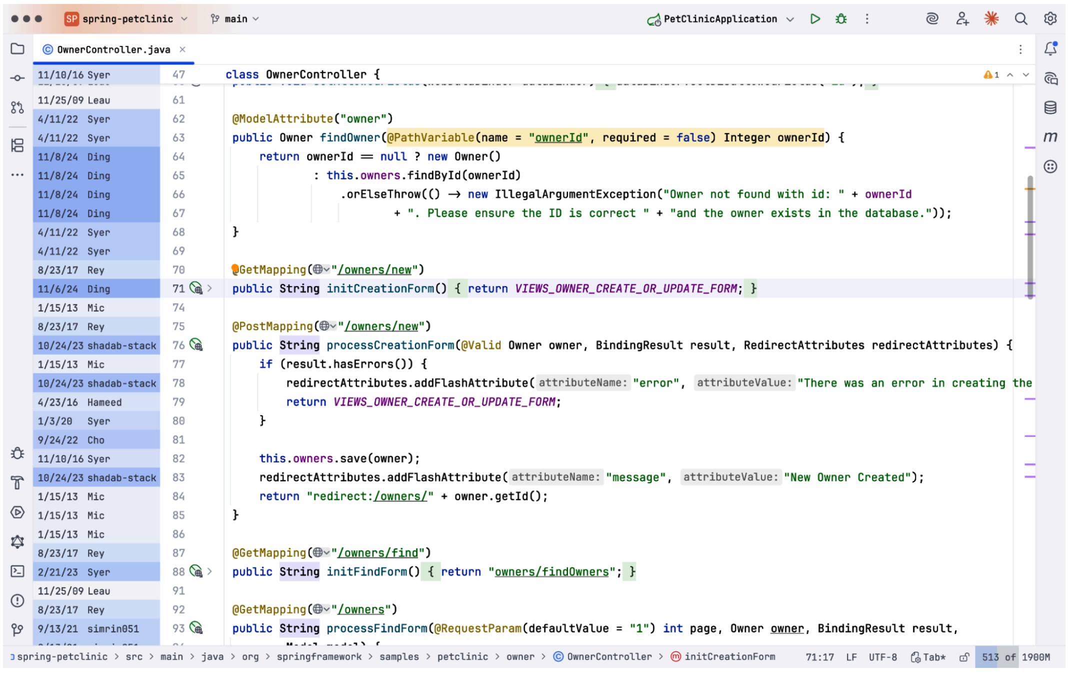
OwnerController	▭									
PetController		▭								
PetTypeFormatter			▭							
PetValidator		1		▭						
VisitController					▭					
OwnerRepository	15	13	5		7	▭				
Owner	25	18			6	5	▭			
Pet		25		7	3		19	▭		
PetType		1	11	1		1		5	▭	
Visit					6		2	3		▭

Figure 2-5. Dependencies in the owner package of the Spring PetClinic application

Use your source code management tool as well (see Figure 2-6). Many modern tools allow you to quickly move about your project. Look at the change history of the files. What changes frequently? What do the commit logs tell you about the updates? Start with the most frequently modified classes, something Git can show you with a command like this:

```
git log -pretty=format: --since="1 year ago" --name-only - "*.java"
| sort | uniq -c | sort -rg | head -10
```

You can also use tools like `git blame` to visualize modifications to the code. If you've just joined a new project, who on your team made the most recent modification or the most frequent changes? Your IDE can also show you the change history if you don't feel like using the command line. However you choose to investigate the code, don't be afraid to reach out to your teammates with questions!

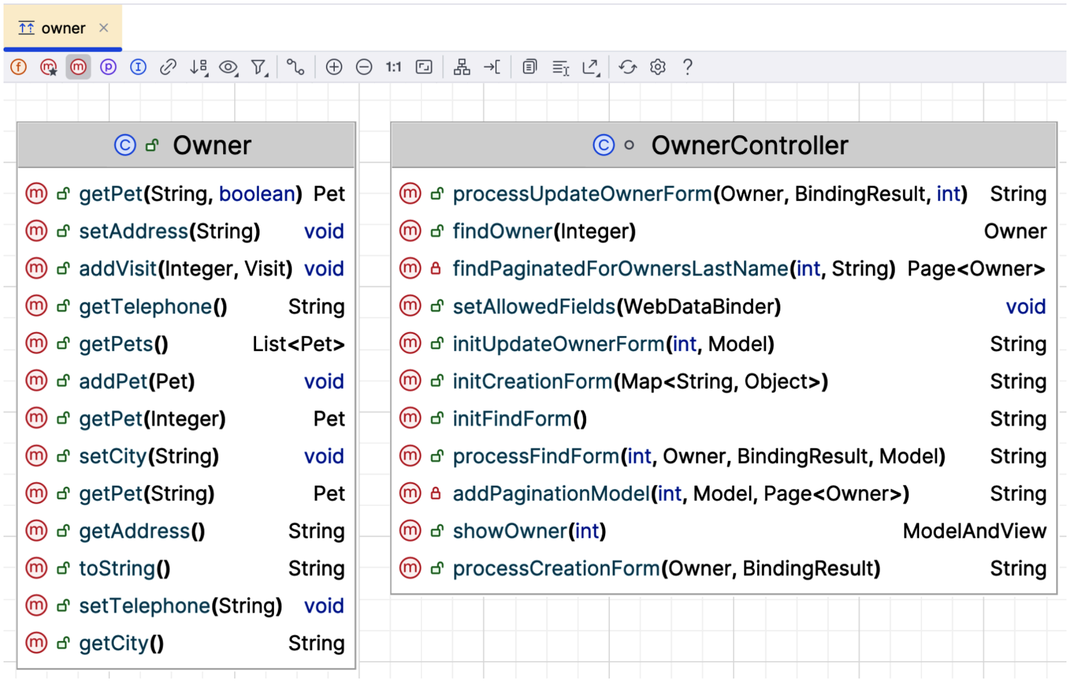


```
11/10/16 Syer           class OwnerController {
11/25/09 Leau
4/11/22  Syer           @ModelAttribute("owner")
4/11/22  Syer           public Owner findOwner(@PathVariable(name = "ownerId", required = false) Integer ownerId) {
11/8/24  Ding               return ownerId == null ? new Owner()
11/8/24  Ding                       : this.owners.findById(ownerId)
11/8/24  Ding                           .orElseThrow(() → new IllegalArgumentException("Owner not found with id: " + ownerId
11/8/24  Ding                                   + ". Please ensure the ID is correct " + "and the owner exists in the database."));
4/11/22  Syer           }
4/11/22  Syer
8/23/17  Rey            @GetMapping("/owners/new")
11/6/24  Ding           public String initCreationForm() { return VIEWS_OWNER_CREATE_OR_UPDATE_FORM; }
1/15/13  Mic
8/23/17  Rey            @PostMapping("/owners/new")
10/24/23 shadab-stack   public String processCreationForm(@Valid Owner owner, BindingResult result, RedirectAttributes redirectAttributes) {
1/15/13  Mic                if (result.hasErrors()) {
10/24/23 shadab-stack           redirectAttributes.addFlashAttribute( attributeName: "error",  attributeValue: "There was an error in creating the
4/23/16  Hameed                 return VIEWS_OWNER_CREATE_OR_UPDATE_FORM;
1/3/20   Syer               }
9/24/22  Cho
11/10/16 Syer               this.owners.save(owner);
10/24/23 shadab-stack       redirectAttributes.addFlashAttribute( attributeName: "message",  attributeValue: "New Owner Created");
1/15/13  Mic                return "redirect:/owners/" + owner.getId();
1/15/13  Mic            }
1/15/13  Mic
8/23/17  Rey            @GetMapping("/owners/find")
2/21/23  Syer           public String initFindForm() { return "owners/findOwners"; }
11/25/09 Leau
8/23/17  Rey            @GetMapping("/owners")
9/13/21  simrin051      public String processFindForm(@RequestParam(defaultValue = "1") int page, Owner owner, BindingResult result,
```

```
spring-petclinic > src > main > java > org > springframework > samples > petclinic > owner > OwnerController > initCreationForm    71:17  LF  UTF-8  Tab*  513 of 1900M
```

Figure 2-6. Modern editors can show you who edited the code and when they did so

While purpose-built project models often fall out of sync with the code as the application evolves, you can always *extract* diagrams from the codebase. Some IDEs will do this with a simple key combination (see Figure 2-7, for example),[7] but you can also use tools like Umbrello, Doxygen, or Structurizr to create a visual representation of the code. Consider adding a step to your build pipeline that automatically generates fresh diagrams whenever code is committed.



Owner	
getPet(String, boolean)	Pet
setAddress(String)	void
addVisit(Integer, Visit)	void
getTelephone()	String
getPets()	List<Pet>
addPet(Pet)	void
getPet(Integer)	Pet
setCity(String)	void
getPet(String)	Pet
getAddress()	String
toString()	String
setTelephone(String)	void
getCity()	String

OwnerController	
processUpdateOwnerForm(Owner, BindingResult, int)	String
findOwner(Integer)	Owner
findPaginatedForOwnersLastName(int, String)	Page<Owner>
setAllowedFields(WebDataBinder)	void
initUpdateOwnerForm(int, Model)	String
initCreationForm(Map<String, Object>)	String
initFindForm()	String
processFindForm(int, Owner, BindingResult, Model)	String
addPaginationModel(int, Model, Page<Owner>)	String
showOwner(int)	ModelAndView
processCreationForm(Owner, BindingResult)	String

Figure 2-7. Class diagrams extracted from the code, giving you a high-level overview of the structure of your classes

Effective Code-Reading Strategies

Now that you have a grasp on the project's purpose and a good lay of the land, it's time to start understanding the code. In this section, you will learn some effective strategies for reading code, such as leveraging IDE features and analyzing tests for insight.

Leveraging IDE Features

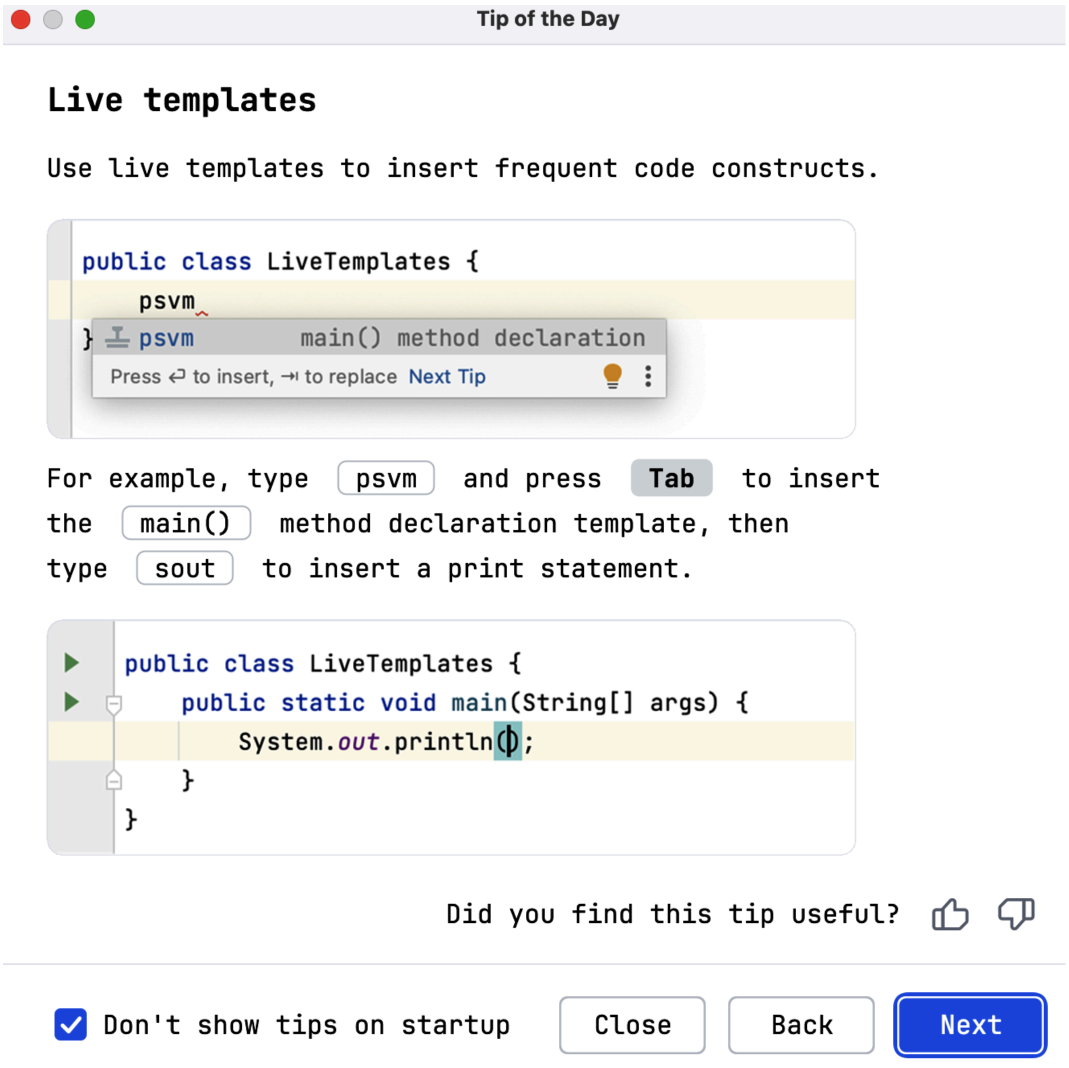
Your IDE is so much more than a text editor for writing code. IDEs have a wide range of features that can help you write, debug, and navigate codebases with ease. When working with existing code, your IDE becomes even more valuable as a tool for exploration and comprehension.

Continuous learning approach

IDEs are so powerful these days that it would be almost impossible to figure out every little trick they have. Instead, you should focus on learning something new every week. For example, JetBrains IntelliJ IDEA is a favorite among Java developers, and it has a really nice feature called the Tip of the Day. When you open up the IDE, there is a tip to help you get acquainted with the features.

These tips cover everything from shortcuts, tools, refactoring, debugging, and plug-ins. Learning the shortcuts of your IDE will make you a much more productive developer, and this is the first place you should start. You will find similar features in a lot of the IDEs on the market today, so check the documentation and start learning.

If you happen to miss the tip of the day in IntelliJ, you can always find it by choosing Help → Tip of the day. In the tip shown in Figure 2-8, IntelliJ is teaching us about a feature called live templates, which can be used to insert frequent code constructs.


Tip of the Day

Live templates

Use live templates to insert frequent code constructs.

```
public class LiveTemplates {
    psvm
}  psvm          main() method declaration
   Press ↩ to insert, ⇥ to replace  Next Tip
```

For example, type psvm and press Tab to insert the main() method declaration template, then type sout to insert a print statement.

```
public class LiveTemplates {
    public static void main(String[] args) {
        System.out.println();
    }
}
```

Figure 2-8. Enabling the tip of the day on your editor of choice is a simple way to practice continuous learning

Tip

Create a personal IDE Tricks document where you record new shortcuts or features you discover. Review it regularly to reinforce what you have learned. Refer back to this document and use the tips you have recorded to build that muscle memory.

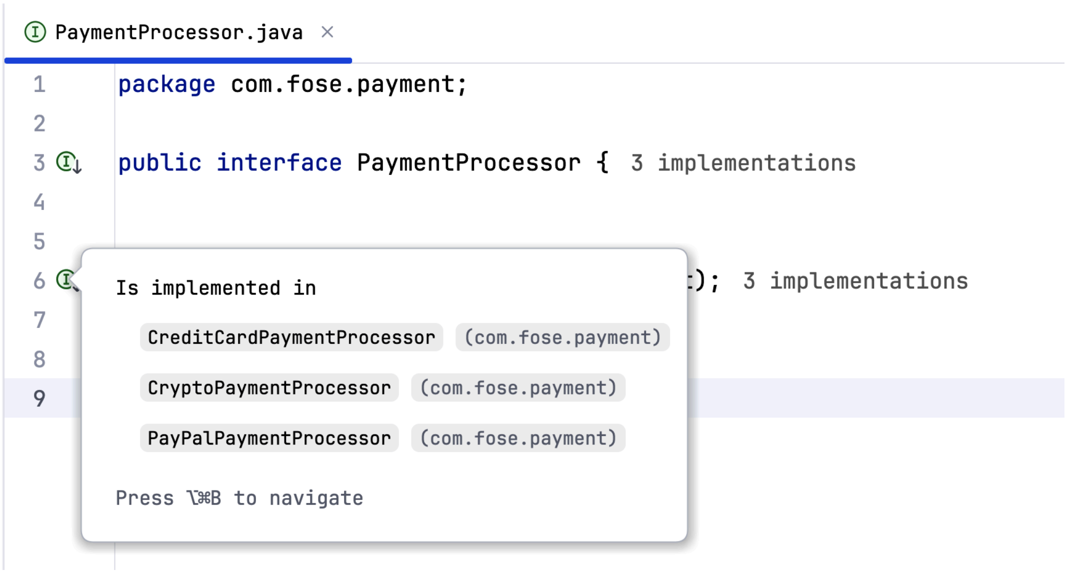
Code navigation tools

One of the most valuable skills when exploring an unfamiliar codebase is the ability to navigate efficiently through the code. Many IDEs will allow you to jump to and even create code in other files without having to manually open up that file through the navigation bar. In the following sections, you will find some practical navigation features that can dramatically speed up your code exploration.

Find usages and references

When you're trying to understand a class or a component, you need to see where in the codebase it is being used. Modern IDEs excel at this task, and it's something you should get familiar with.

If you were to open up a Java interface in IntelliJ, such as the one shown in Figure 2-9, you can click the gutter icon to find all implementations of that interface.[8] This is really handy when you're trying to understand how an abstraction is used in practice. Similarly, you can right-click a method name and select Find Usages to see every place where that method is called.



PaymentProcessor.java

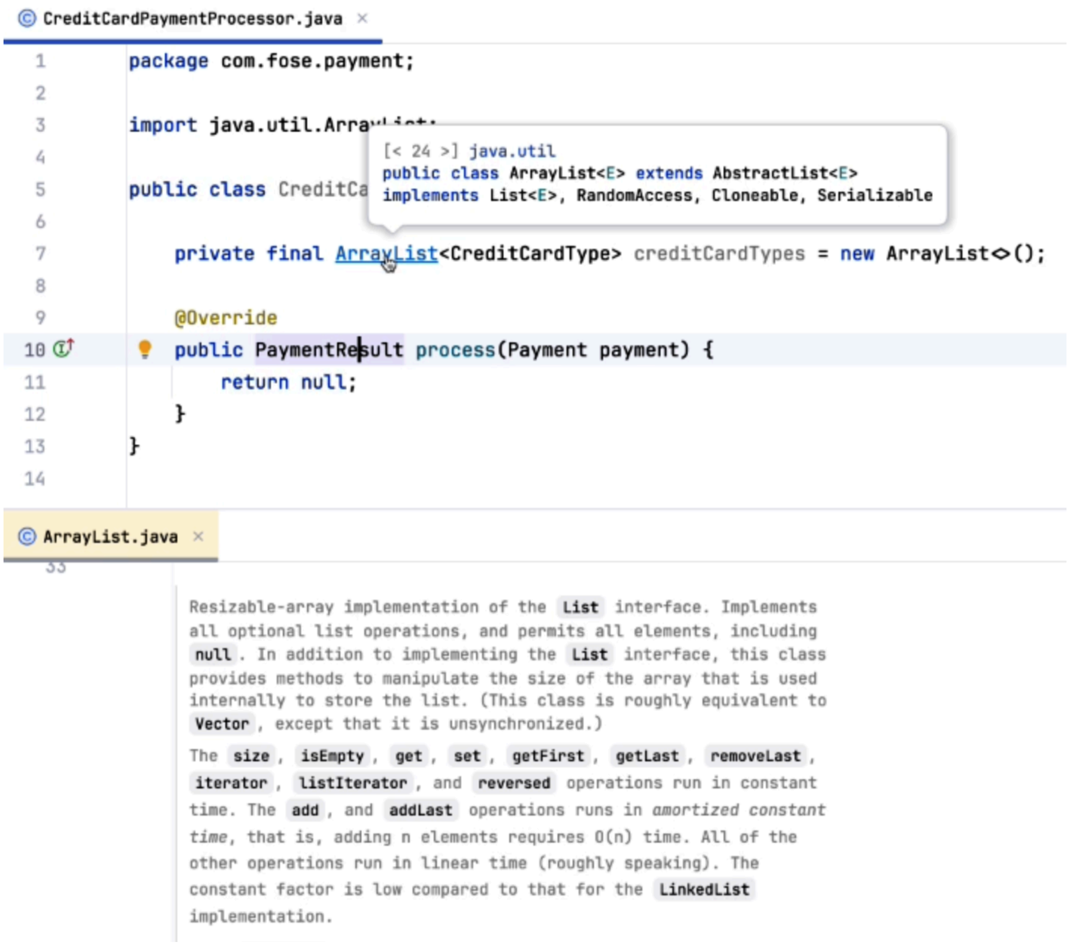
```
package com.fose.payment;

public interface PaymentProcessor {  3 implementations


                                   );  3 implementations
```

```
Is implemented in
  CreditCardPaymentProcessor  (com.fose.payment)
  CryptoPaymentProcessor  (com.fose.payment)
  PayPalPaymentProcessor  (com.fose.payment)

Press ⌥⌘B to navigate
```

Figure 2-9. Modern editors can quickly show you where something is used or referenced and offer the ability to quickly navigate to them

Jump to definition

Another useful IDE navigation feature is the ability to jump to the definition of a class, method, or variable. This lets you quickly move from usage to implementation, following the code's logical flow. For example, if you're writing a Java application and using an `ArrayList` and want to learn how it works, you can Command-click (or Ctrl-click on Windows/Linux) on the `ArrayList` type to be taken to the source, where you can view the code and documentation for that type (see Figure 2-10).



CreditCardPaymentProcessor.java

```java
package com.fose.payment;

import java.util.Arra

public class CreditCa

    private final ArrayList<CreditCardType> creditCardTypes = new ArrayList<>();

    @Override
    public PaymentResult process(Payment payment) {
        return null;
    }
}
```

```
[< 24 >] java.util
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList.java

```
Resizable-array implementation of the List interface. Implements
all optional list operations, and permits all elements, including
null. In addition to implementing the List interface, this class
provides methods to manipulate the size of the array that is used
internally to store the list. (This class is roughly equivalent to
Vector, except that it is unsynchronized.)
The size, isEmpty, get, set, getFirst, getLast, removeLast,
iterator, listIterator, and reversed operations run in constant
time. The add, and addLast operations runs in amortized constant
time, that is, adding n elements requires O(n) time. All of the
other operations run in linear time (roughly speaking). The
constant factor is low compared to that for the LinkedList
implementation.
```

Figure 2-10. Jumping to a definition can help you navigate code quickly

The ability to jump to a definition is valuable when working with the following:

- Third-party libraries
- Framework code
- Base classes and interfaces
- Utility methods used throughout the codebase

Call hierarchies and dependency views

Understanding how components interact is crucial to grasping an unfamiliar codebase. IDEs offer specialized views for visualizing these relationships:

Call hierarchy
: Shows what calls a method and what methods it calls

Type hierarchy
: Displays inheritance relationships

Dependency diagram
: Visualizes how modules or packages depend on each other

For example, in IntelliJ, you can right-click a method and select Show Call Hierarchy to see both incoming calls (where this method is called from) and outgoing calls (methods this method calls). This can quickly show you a method's role in the larger system.

Code analysis features

Modern IDEs have powerful tools to help you navigate code, but they can do so much more. They can actively analyze the code that you're writing and offer insights that can both improve your code and help you learn new features you might not have been aware of.

Automated inspection and suggestions

When you write or modify code, IDEs can offer suggestions on how to refactor code. It's important to remember that these are merely suggestions: you don't always have to accept them, but it is helpful to review them. These suggestions often reveal common patterns and best practices that can not only improve code quality but also teach you something new in the process.

Here's an example in Java where the code is iterating over a collection of `Books`. For each `Book` in the collection, the code simply prints out the name of the book:

```
List<Book> books = library.getBooks();
for (Book book : books) {
    System.out.println(book.getTitle());
}
```

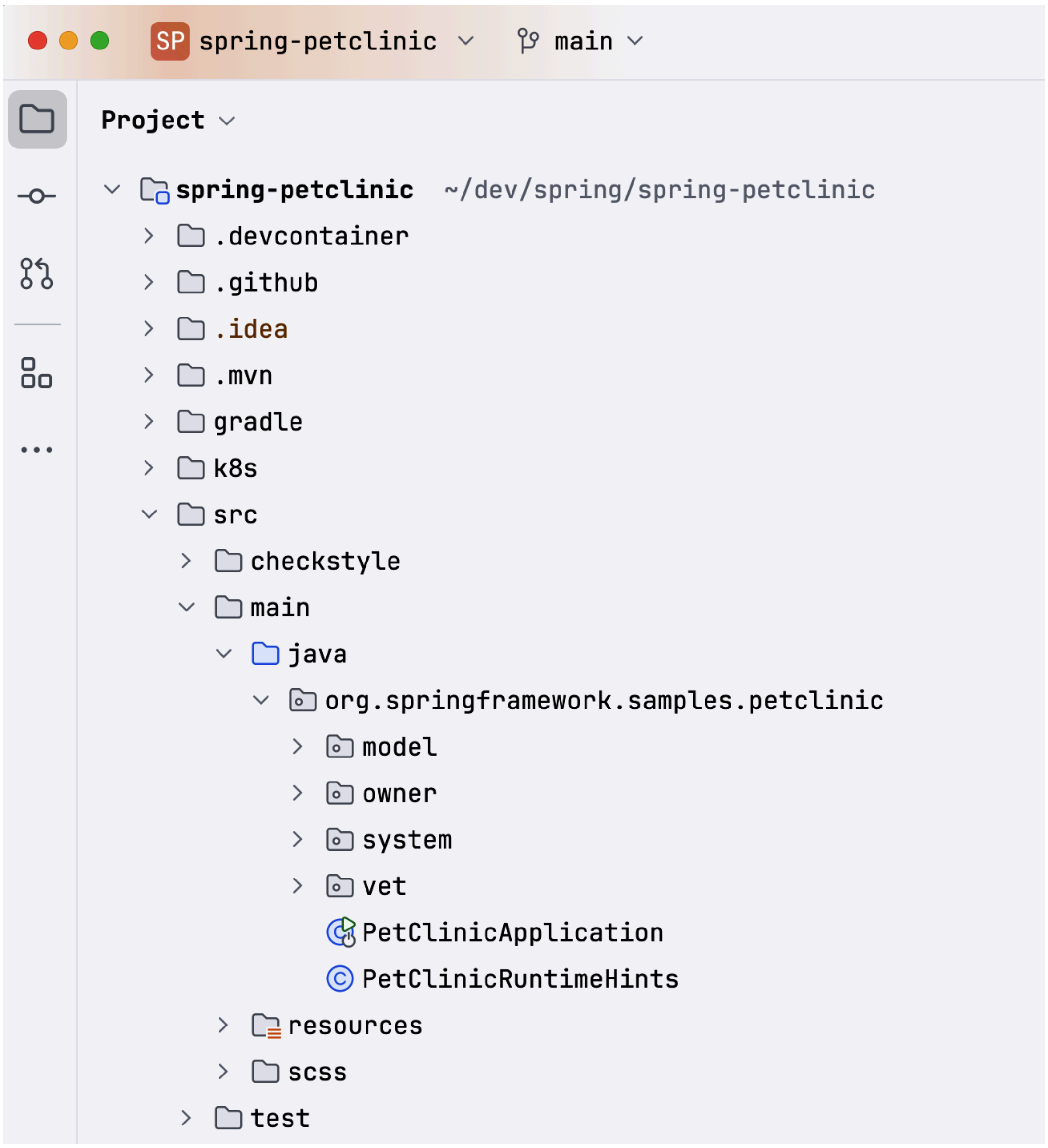
This works, and there is nothing wrong with the code, but IntelliJ might suggest using a more streamlined approach:

```
library.getBooks().forEach(book -> System.out.println(book.getTitle()));
```

These suggestions not only help you write better code but also teach you idiomatic patterns in the language and frameworks you're using.

Code structure visualization

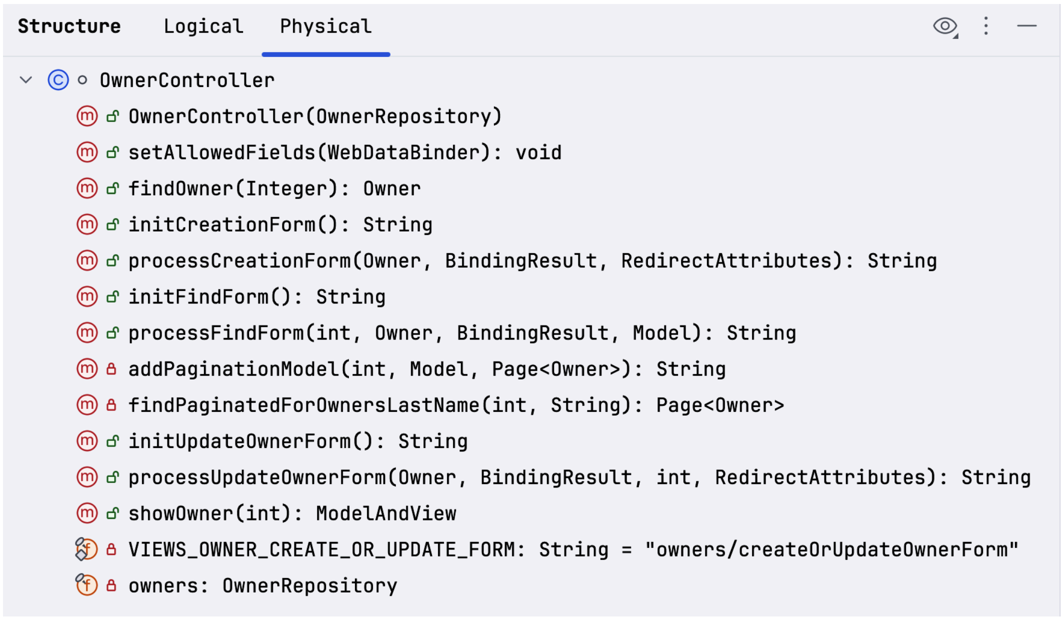
Another important part of being able to effectively read code is understanding how the code is structured within a project or even a single class. IDEs often include features that visualize the structure of your code. If you want to try to understand how the code in Figure 2-11 is architected, for example, you can look in the project structure and quickly see that it is a monolithic application and the code is organized in a package-by-feature arrangement.



```
spring-petclinic  ~/dev/spring/spring-petclinic
  .devcontainer
  .github
  .idea
  .mvn
  gradle
  k8s
  src
    checkstyle
    main
      java
        org.springframework.samples.petclinic
          model
          owner
          system
          vet
          PetClinicApplication
          PetClinicRuntimeHints
      resources
      scss
    test
```

Figure 2-11. The project structure can help you see the bigger picture

Suppose you were trying to understand what methods were available in a certain class. Sure, you could check out the API documentation, but do you even know if that is the most up-to-date version of the docs? You could manually open up a class file and begin scrolling through the source code, but this could take some time. Most IDEs have a way to visualize the structure of a class or component. In the example shown in Figure 2-12, you are looking at the structure of the `OwnerController` in a project called PetClinic. You can see at a glance what methods are available in the class, giving you quick insights into the purpose of this class and its functionality.


Structure Logical Physical

```
OwnerController
    OwnerController(OwnerRepository)
    setAllowedFields(WebDataBinder): void
    findOwner(Integer): Owner
    initCreationForm(): String
    processCreationForm(Owner, BindingResult, RedirectAttributes): String
    initFindForm(): String
    processFindForm(int, Owner, BindingResult, Model): String
    addPaginationModel(int, Model, Page<Owner>): String
    findPaginatedForOwnersLastName(int, String): Page<Owner>
    initUpdateOwnerForm(): String
    processUpdateOwnerForm(Owner, BindingResult, int, RedirectAttributes): String
    showOwner(int): ModelAndView
    VIEWS_OWNER_CREATE_OR_UPDATE_FORM: String = "owners/createOrUpdateOwnerForm"
    owners: OwnerRepository
```

Figure 2-12. Understanding the functionality of a class at a glance with the structure view

Reading Tests for Insight

When it comes to exploring a new codebase, tests serve as another form of documentation for developers. Well-written tests reveal not only what the code does but also why it does it a certain way. While the code will tell you how something works, tests will tell you how it's supposed to work.

Tests as living documentation

Unlike traditional documentation that often becomes outdated, tests must remain functional to pass continuous integration checks. This makes them a reliable, up-to-date source of truth about system behavior.

Tests document the expected inputs and outputs of methods, the interactions between components, and the overall system behavior. By reading tests first, you can understand what a component is supposed to do before diving into how it actually does it. This provides crucial context that makes the implementation easier to comprehend.

The following test verifies that regular customers receive a 10% bulk discount when placing large orders over a certain threshold, ensuring that the pricing service correctly calculates discounts based on order size rather than customer type:

```
@Test
public void shouldApplyBulkDiscountForLargeOrders() {
    // Given
    Customer regularCustomer = new Customer(CustomerType.REGULAR);
    Order bulkOrder = new Order(regularCustomer, 500.00);

    // When
    double finalPrice = pricingService.calculateFinalPrice(bulkOrder);

    // Assert
    assertEquals(450.00, finalPrice, 0.01);
}
```

With that context, you can now dive into the pricing service class and examine the `calculateFinalPrice` method to understand the code that produces that result.

Understanding workflows and processes

Tests can often map to user stories, which often make them a great way of understanding business processes or workflows. If you zoom out and look at integration tests, they can often show you the big picture of how components work together to fulfill business requirements.

Look for tests that verify entire processes from start to finish. These tests often contain setup code that represents real-world scenarios, giving you context about how the system behaves in production.

The following test verifies that when a user submits a product review, the review service correctly updates the product's average rating and sends a notification to the administrator, ensuring both the data persistence and notification aspects of the review process work as expected:

```
@Test
public void submitProductReviewShouldUpdateRatingAndNotifyAdmin() {
    // Given
    Product product = productService.findProductById("XYZ789");
    User user = userService.findByUsername("janedoe");
    Review review = new Review(user, product, 4, "Great quality product!");

    // When
    ReviewResult result = reviewService.submitReview(review);

    // Then
    assertTrue(result.isSuccessful());
    assertEquals(4.2, productService.findProductById("XYZ789")
    .getAverageRating(), 0.01);
    verify(notificationService).sendReviewNotification(anyObject(), eq("XYZ789"));
}
```

With this context, you now understand the whole flow of what happens when a user submits a new review.

Discovering edge cases and boundaries

In software development, an *edge case* is a situation that occurs at the extreme edge of a program's expected input, operating conditions, or usage patterns, and is often the case where assumptions break down. These inputs or conditions are valid but unusual, and they will reveal bugs that don't typically happen during the "typical" or "happy-path" scenarios. In existing codebases, you might find more tests around these because they were discovered over time, and a test was written to ensure they don't surface again.

Pay special attention to tests with names like `shouldHandleEmptyList`, `shouldRejectInvalidInput`, or `shouldTimeoutAfterTenSeconds`. These tests show you the limits of the system's capabilities and highlight potential failure points.

The following test reveals that the system will handle expired credit cards and fail gracefully. This is something that might not be as obvious if you were just reading through the implementation:

```
@Test
public void shouldRejectPaymentWhenCreditCardIsExpired() {
    // Given
    CreditCard expiredCard = new CreditCard("4111111111111111", "05/20");
    Order order = new Order(new Customer(), 50.00);

    // When
    PaymentResult result = paymentService.processPayment(order, expiredCard);
    // Then
    assertFalse(result.isSuccessful());
    assertEquals(PaymentFailureReason.EXPIRED_CARD, result.getFailureReason());
}
```

With this context, you can go through the implementation of the process payment method and look for the functionality that will catch this edge case.

With the ability to understand a project and effectively read code, it's time to modify the code. But how can you go about doing that safely? We'll cover that topic in the next section.

Practice Makes Perfect

At the end of the day, practice some grace with yourself. Modern codebases are often sprawling. One person cannot understand it in its entirety, and that isn't the goal. Your knowledge will grow over time. Rinse and repeat the process as you encounter new parts of your project. It can be intimidating, but every developer has gone through it. You will be fine!

How do you improve your code-reading skills? As much as you may dread it, practice reading code. There are so many well-written, publicly available, open source options in a variety of languages for you to choose from. There aren't any shortcuts; you cannot improve without practice. It does get easier over time, and you will get faster.

What About AI?

Like any discipline, software tooling continues to evolve. From shell-based text editors to IDEs with integrated refactoring tools and IntelliSense code completion, writing code has gotten easier. Of course, the size and complexity of the applications you're working on has also grown, so perhaps it's a wash! With the advent of things like GitHub Copilot and generative AI like ChatGPT, some have even suggested that developers will soon be replaced. The rumors of the end of developers are often grossly exaggerated, from COBOL to fourth-generation programming languages to various drag-and-drop "programming" solutions failing to result in a large-scale reduction in the demand for software engineering.

These tools have a place in your toolbox—for example, you can ask ChatGPT what a chunk of code does,[9] but you still need to understand the context. AI doesn't know the purpose of a given class, so you still need to read the code, wade through the documentation, and talk to your fellow developers. AI doesn't (yet) fully understand an entire enterprise codebase, and the risk of hallucinations is real. While AI can help you, ultimately, *you* must understand the nuance and relationships between code spread across multiple repositories.

As Dan likes to say, when using AI, you are the pilot, not the passenger: trust but verify. AI can understand code in a vacuum, but it may not understand an enterprise-grade codebase without possible hallucinations. Modern applications are often spread across many repositories, leveraging multiple libraries. You, the developer, must understand the relationships and intricacies in order to enable the AI tools to be successful in accelerating development.

Wrapping Up

Arguably, coding is taught backward: you learn to write before you learn to read, and yet you will spend a significant amount of your career reading code written by someone else. While you may not enjoy existing code as much as greenfield development, it comes with the paycheck. Rather than run from the situation, learn to embrace it, as there is much to gain professionally. Be aware of cognitive biases. Don't be afraid to roll up your sleeves and root around in an unfamiliar codebase; you will learn something. As your understanding grows, leave the code better than you found it, easing the path of the next developer—which just might be you!

Putting It into Practice

If you want to get better at reading code, there are no shortcuts; you need to read more code. Luckily, you have a veritable plethora of open source projects at your disposal! Block out a couple of hours to read some of the code in the framework you use (or the one you wish you used) at work or in another project you're interested in. If you're not sure where to start, check out the trending repositories on GitHub. Apply the techniques you learned in this chapter. In a couple of months, pick another part of the project you explored or try a completely different one; was it easier than the first time? Keep at it; over time, your code-reading skills will improve.

Working with existing code is also a skill that needs to be developed, and once again, open source software gives you a massive playground to explore. Contributing to open source is an excellent learning laboratory, and it isn't nearly as hard to get started as you may think.[10] Pick a project and spend a few hours working with it.

Additional Resources

- "Code as Design: Three Essays" by Jack W. Reeves
- "Reading Code Is Harder Than Writing It" by Trisha Gee
- "Reading Other People's Code" by Patricia Aas
- "How to Quickly and Effectively Read Other People's Code" by Alex Coleman
- "How to Read Code Without Ripping Your Hair Out" by Sunny Beatteay

[1] Sometimes known as *vibe coding*.

[2] Or as Neal Ford once said: "Today's best practice is tomorrow's anti-pattern."

[3] Michael I. Norton, Daniel Mochon, and Dan Ariely, "The 'IKEA Effect': When Labor Leads to Love," HBS Working Paper, No. 11-091 (2011), *https://oreil.ly/zuETL*.

[4] Said documentation *may* be out-of-date, trust but verify.

[5] To learn more about ADRs, see Chapter 3 of *Head First Software Architecture* by Raju Gandhi et al. (O’Reilly, 2024)

[6] Of course, not all applications have such cleanly named files; you may have to make ample use of your favorite search tools.

[7] For example, ⌥⇧⌘U (macOS) / Ctrl-Alt-Shift-U (Windows/Linux) in IntelliJ IDEA will generate a UML diagram.

[8] A Java interface is a reference type that defines a contract of methods that implementing classes must provide, allowing for abstraction and polymorphism in Java programs.

[9] Your organization’s lawyers likely have strong opinions about the use of such tools, so double-check your corporate policies before you paste your proprietary pricing algorithm into a model that might be using your code as training data!

[10] Many projects have lists of bugs marked as “for first-time contributors,” but don’t be afraid to reach out to the current project contributors; most will happily help you get up and running.