Chapter 6. Exploring and Modifying Unfamiliar Systems

> Some of the most valuable experience I gained was from supporting a legacy app. I highly recommend it, but I wouldn't wish it on anyone.
>
> Dalia Abo Sheasha, software developer

While many developers love the blank canvas of greenfield projects, the harsh reality of software development is that most of the work we do will be on established systems. Don't let this reality discourage your view on the profession of software development. Working with existing codebases has its advantages. Many key decisions have already been made, allowing you to focus directly on the problems at hand instead of worrying about infrastructure.

As someone getting onboarded to a new project or simply exploring a new codebase, you will need to develop skills to explore and modify unfamiliar systems. In this chapter, you'll discover how to navigate and understand unfamiliar code and how to make changes safely. Whether you're joining a new team or maintaining legacy systems, these skills will help you confidently contribute to any codebase. This builds on the mechanics of reading code that we covered in Chapter 2.

Understanding Unfamiliar Codebases

Whether you're onboarding at a new company or moving to a new project at your current employer, working with unfamiliar codebases can be stressful and anxiety inducing. This can be caused by the unknown or working with languages, frameworks, or tools that you aren't familiar with. However, if you approach this challenge with a clear plan and techniques to navigate these uncharted waters, you'll boost both your success rate and confidence.

In this section, you will learn techniques for understanding the big picture or goals of the project. You'll learn how to get familiar with a new project by following the flow or execution path of the code. Finally, you'll learn how to incrementally build mental models by breaking complex systems into manageable pieces. As you work through this chapter or explore a new project, remember that it's OK to not immediately understand everything in a new codebase. Even team members who have worked on the project for years don't know everything about it.

Start with the Big Picture

"You can't see the forest through the trees." This means that if you're so focused on tiny details (which version of your framework you're using), you fail to understand the bigger picture or overall situation (the goal of this project). You probably want to dive in and start contributing to the project and proving your worth to the team right away. It's always nice to have a plan, and here are some tips for getting a grasp on the big picture.

Understanding the project

Before you can begin to even understand the code, you need to understand the purpose and intent of the project. Start with a high level: what is the overall reason for this project existing? Who are the stakeholders of this project? What does this project mean to your employer? Is it a large part of the business?

There are many ways to do this, but a good place to start is with someone who has deep knowledge of the product, like a product manager. See if you can set up a meeting with this person and be prepared to learn. Take notes as much as you can so that you can refer back to them later. If this is a virtual meeting, ask if you can record it so you can review it again later. Most importantly, ask questions. The only dumb questions are the ones you don't ask.

After talking with a product owner and getting a high-level overview of the project, it's time to dive deeper into the technical side of things. At this point, reach out to any software engineers who currently work on the project, and if you can find any who originally developed the system, consider this like striking gold during the gold rush. Ideally, you will want to reach out to someone in a tech lead or architecture role who can provide valuable insights about decisions that were made.

Who are the customers of this product? Go out and find any public information about this project. If there is any, try to look at this from the customers' perspective. Some companies develop user personas for their products, which are fictional profiles of the actual users that can help you understand different types of users, their goals, and pain points. This will give you valuable context for why certain technical decisions were made.

Reviewing available documentation

After talking to stakeholders, the next step on the path to understanding a codebase is to comb through any and all available documentation, including ADRs. With a high-level understanding of the project itself, some of this documentation should start to make some sense. At this point, you are trying to learn as much about the codebase as you can through the documentation.

A big part of this is the infamous "onboarding documentation" to a project. This will tell you everything you need to know to get this project up and running on your local machine. This documentation has gotten better over the years with tools like Docker, but a review can still be a tedious process.

While your job is not to write documentation at this point because you're still getting familiar with everything, you should be taking notes and identifying gaps in the documentation. This will be something you can come back to later and fix and is a great way to contribute to the team.

The Documentation Trap

Dan here. I once spent three days trying to understand a complex authentication system, carefully reading through what I thought was up-to-date documentation. When I finally asked a teammate for help, they laughed and said, "Oh, that documentation is from two versions ago. We completely rewrote the auth system last year." The lesson? Documentation can lie, but the code never does. Always verify what you read against what's actually running. And don't be afraid to ask for help. A three-minute conversation would have saved me three days.

Understanding architecture and project structure

After understanding the overall project and reading through all of the available documentation, it's time to examine the architecture and organization of the codebase. This is an important step before diving in and reading any code or configuration or writing code. The project's structure can reveal a lot of information and give you a more holistic view of the project that will help you when navigating the codebase.

Most projects follow some form of architectural pattern, whether by deliberate design or through organic evolution. Start by identifying which pattern (or combination of patterns) the codebase follows.

Package by layer organizes code horizontally based on technical responsibilities. You'll often see top-level directories like controllers, services, repositories, and models. This approach groups similar technical components together, making it easy to find all components of a particular type. The following code is packaged by layer:

```
src/
   ├── controllers/
   │   ├── UserController.java
   │   └── ProductController.java
   ├── services/
   │   ├── UserService.java
   │   └── ProductService.java
   ├── repositories/
   │   ├── UserRepository.java
   │   └── ProductRepository.java
   └── models/
       ├── User.java
       └── Product.java
```

Package by feature organizes code vertically around business capabilities or features. You'll see top-level directories representing business domains like users, products, and orders. This approach encapsulates all aspects of a feature together, making it easier to understand complete business workflows. The following is an example of code packaged by feature:

```
src/
   ├── users/
   │   ├── UserController.java
   │   ├── UserService.java
   │   ├── UserRepository.java
   │   └── User.java
   └── products/
       ├── ProductController.java
       ├── ProductService.java
       ├── ProductRepository.java
       └── Product.java
```

Hexagonal architecture (also known as *ports and adapters*) organizes code to separate business logic from external concerns. Look for a core domain model surrounded by adapters that connect to the outside world. This pattern emphasizes isolation of business rules from technical implementations. The following is an example of code that has a hexagonal architecture:

```
src/
   ├── domain/                  // Business logic
   │   ├── model/
   │   └── service/
   ├── application/             // Use cases, orchestration
   │   └── service/
```

```
├── ports/                  // Interfaces for adapters
│   ├── input/
│   └── output/
└── adapters/               // Technical implementations
    ├── web/
    ├── persistence/
    └── messaging/
```

Microservices architecture splits functionality into multiple independent services. If you're working in a microservices environment, you'll need to understand both the architecture of your specific service and how it fits into the larger ecosystem. In the following example, each top-level service like *user-service* is its own microservice:

```
microservices-system/
├── user-service/
│   ├── controllers/
│   ├── services/
│   ├── repositories/
│   ├── models/
│   └── Dockerfile
├── product-service/
│   ├── controllers/
│   ├── services/
│   ├── repositories/
│   ├── models/
│   └── Dockerfile
├── order-service/
│   ├── controllers/
│   ├── services/
│   ├── repositories/
│   ├── models/
│   └── Dockerfile
├── api-gateway/
│   ├── config/
│   ├── filters/
│   └── routes/
└── shared/
    ├── common-models/
    └── utils/
```

Understand the Execution Flow

Execution flow is the sequential path of instructions that a program follows during runtime, including all decisions, loops, and function calls that determine which code executes and in what order.

Now that you have the big picture of the project and understand its purpose, you can begin looking at code. We aren't actually writing any code or doing any in-depth analysis at this point; we are just trying to understand how the code flows. In this section, you will learn techniques for understanding the execution flow by finding application entry points, tracing requests and responses, and learning how to locate external dependencies.

Finding application entry points

An application has a set of doors that act as entry points into your application. These doors come in the form of an application's main method, public APIs, web UIs, and more. These doorways into the application are a great place to start if you want to learn about the different execution flows in an application.

For example, in a Spring Boot web application,[1] you can find the main class by looking for the `@SpringBootApplication` annotation and locating the main method:

```
@SpringBootApplication
public class PetClinicApplication {
    public static void main(String[] args) {
        SpringApplication.run(PetClinicApplication.class, args);
    }
}
```

When you have identified one of the application's entry points, you can begin to follow the code's execution flow. This is where using your IDE's debugger can be a really valuable tool. Set a breakpoint on the run method and then use the step-through functionality. This will give you insight into what the framework is doing under the hood and provides an excellent starting point for understanding the codebase.

To learn more about the application's entry points, you can locate the public APIs, which in this PetClinic example application are the REST endpoints. The following `@GetMapping` section tells us that we can send a request to */pets/new* and that this is the method that will execute. Now we can use the features of the IDE and our debugging tools to follow the execution path:

```
@GetMapping("/pets/new")
public String initCreationForm(Owner owner, ModelMap model) {
   Pet pet = new Pet();
   owner.addPet(pet);
   return VIEWS_PETS_CREATE_OR_UPDATE_FORM;
}
```

There are many entry points that can help you map out the execution flow of an application. Here are some common ones, but there are many more:

Main/bootstrap methods
: The traditional starting point of execution (like Java's `main()`)

Public APIs/controllers
: Endpoints that expose functionality to external systems

Event handlers/listeners
: Code that executes in response to specific events or triggers

Scheduled tasks/jobs
: Functionality that runs at predetermined intervals

Lifecycle hooks
: Methods called during component creation, startup, or shutdown

Plug-in/extension points
: Interfaces designed for extending application functionality

Message consumers
: Code that responds to messages from queues or message brokers

Command-line argument processors
: Logic that handles startup parameters

Database triggers/stored procedures
: Server-side code that executes in response to data changes

The key is knowing how to find the right doors to open. When you have identified them, you can use them as entryways into understanding the flow of an application.

Following the data: Tracing request journeys

When working with existing codebases, especially web applications and APIs (REST, GraphQL, gRPC), tracing the journey of a request through a system is a

great way to understand how the code works and the systems involved. While documentation or tests might tell you what is supposed to happen, following a request will reveal what actually happens.

Request tracing is valuable because it does the following:

- Reveals the actual path of execution through multiple components and systems
- Helps identify all the involved layers (controllers, services, repositories, etc.)
- Exposes data transformations that happen along the way
- Uncovers hidden business logic and validation rules
- Shows how errors are handled in practice

As a developer exploring a new codebase, some really great tools are at your disposal for tracing requests. In this section, you'll learn about a few of these tools. You might not need to use all of them, but knowing what is available is helpful. Try a few of them out and see what tools work well for you in your workflow.

Browser developer tools

When working with web applications, browser developer tools are a good first place to inspect a request and response. In the following example, we are running the PetClinic application and adding a new pet to an owner (Figure 6-1).

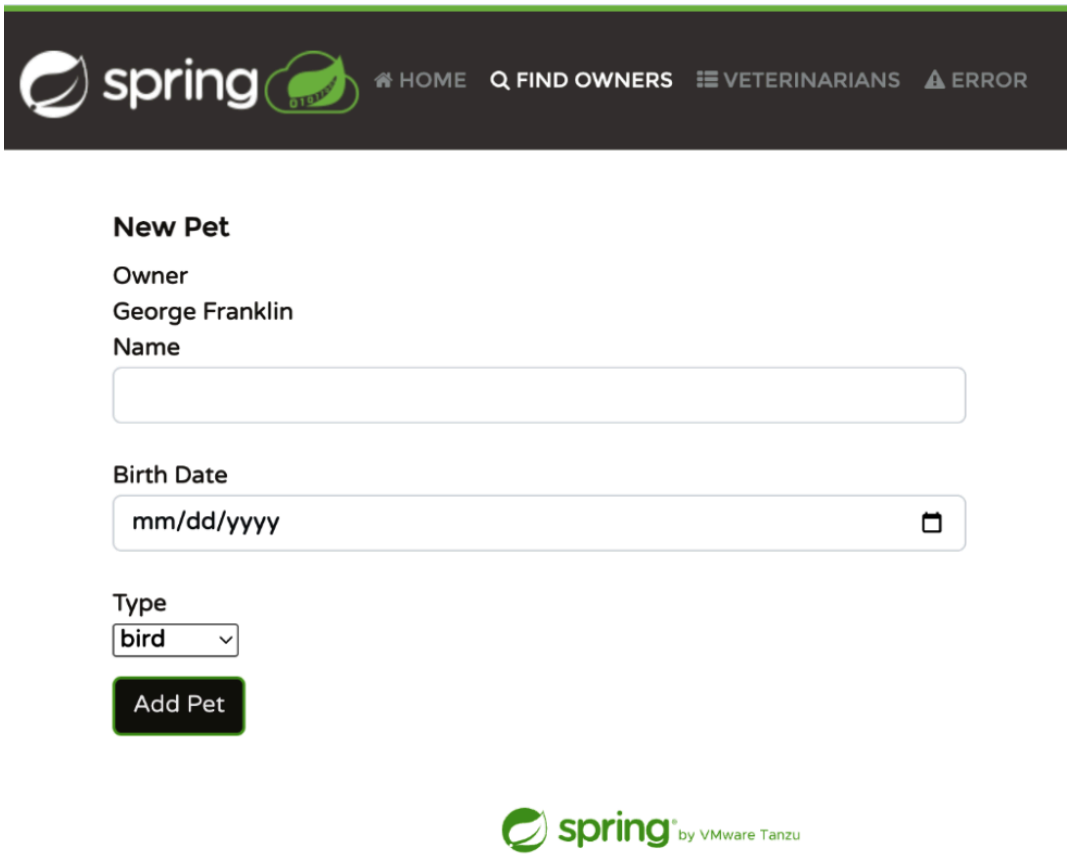

Figure 6-1. The Spring PetClinic application: adding a new pet to an owner form

When you fill out the form and click the Add Pet button, it will send a request to the following endpoint:

```
@PostMapping("/pets/new")
public String processCreationForm(Owner owner, @Valid Pet pet,
BindingResult result,
      RedirectAttributes redirectAttributes) {

   if (StringUtils.hasText(pet.getName()) && pet.isNew() &&
   owner.getPet(pet.getName(), true) != null)
      result.rejectValue("name", "duplicate", "already exists");

   LocalDate currentDate = LocalDate.now();
   if (pet.getBirthDate() != null && pet.getBirthDate().isAfter(currentDate)) {
      result.rejectValue("birthDate", "typeMismatch.birthDate");
   }

   if (result.hasErrors()) {
      return VIEWS_PETS_CREATE_OR_UPDATE_FORM;
   }

   owner.addPet(pet);
   this.owners.save(owner);
   redirectAttributes.addFlashAttribute("message", "New Pet has been Added");
   return "redirect:/owners/{ownerId}";
}
```

If you open up the developer tools and inspect the Network tab, you will see the POST request to the */pets/new* endpoint where you can examine the headers, payload, response, and more, as shown in Figure 6-2.

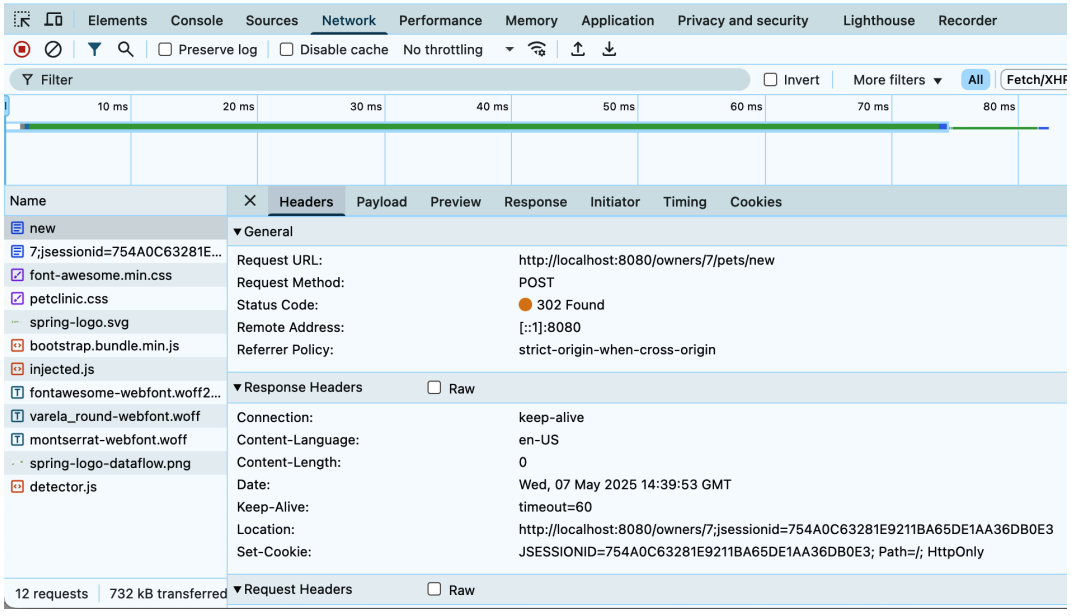

Figure 6-2. Chrome developer tools showing a POST request to /pets/new

API testing tools

If you want to test an API endpoint directly and it's a simple GET request, you can put the URL in your browser and see the result. When it's any other request method like POST/PUT/PATCH/DELETE, you will need to reach for an API testing tool. There are some really great tools on the market like Postman, Insomnia, and Bruno. You will also find plug-ins for a lot of the major IDEs out there that contain similar functionality.

In the previous section, we submitted the form from the browser and used developer tools to inspect the POST request under the hood. What if you didn't

want to go through the UI and test the API endpoint directly? This is where API testing tools shine, giving you the ability to send a POST request along with headers, authorization, a request body, and more. After sending the request, you then have the ability to inspect the response, allowing you to bypass the UI altogether (Figure 6-3).

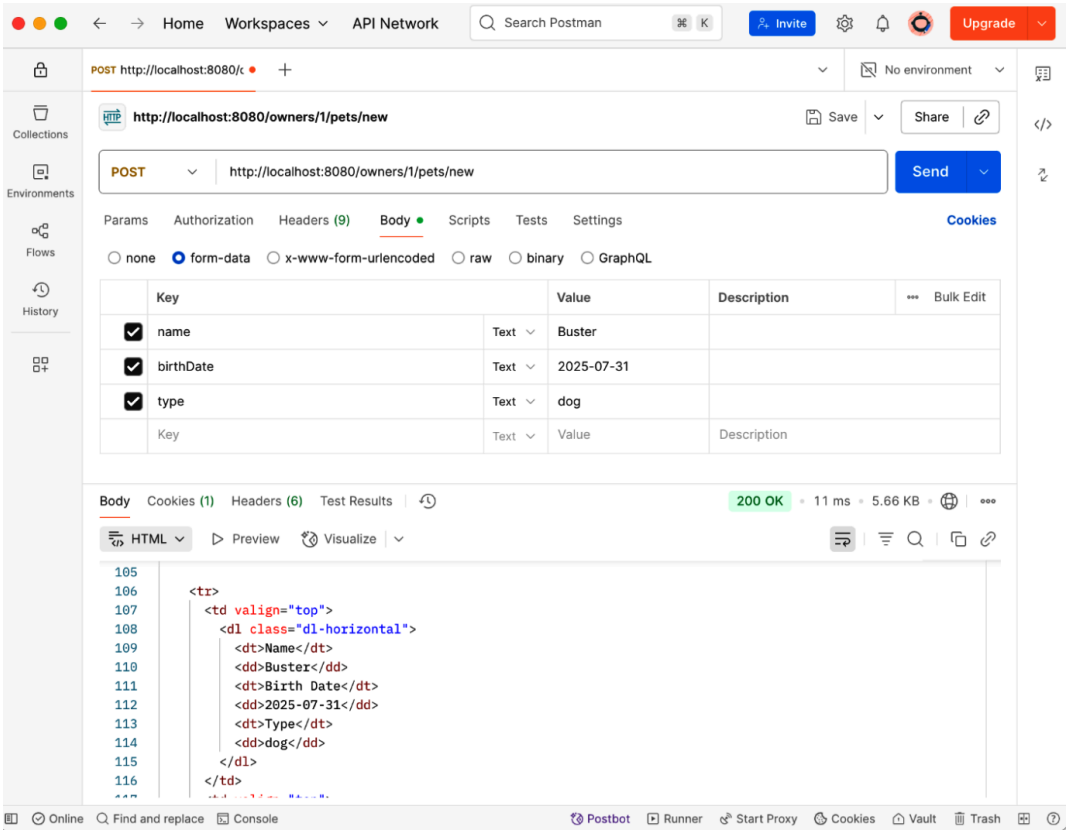

Figure 6-3. Postman API testing tool sending a POST request to the endpoint /posts/new

You can test this endpoint with a tool like Postman and examine the following:

Required fields and data types
: Identify which fields are mandatory in the request body and what format they expect (strings, numbers, arrays, etc.).

Input validation behavior
: Test how the API handles invalid data, missing fields, malformed JSON, and edge cases like empty strings or null values.

Authentication and authorization
: Verify that protected endpoints properly reject unauthorized requests and accept valid credentials.

Response structure and status codes
: Examine what data is returned on successful requests versus different types of failures (400, 401, 404, 500, etc.).

Error message quality
: Assess whether error responses provide clear, actionable feedback for debugging.

Performance characteristics
: Measure response times and identify any endpoints that are unusually slow.

Logging

Logs provide insight into the system's internal behavior. In many applications, logs might be the only tool available for debugging issues or tracing the journey of

a request in production. When examining the code, look for logging statements like this:

```
log.info("Finding owner with id: {}", ownerId);
```

The following endpoint will list out all the pets for a particular owner. If you make a request to this endpoint with a valid owner ID, you will see two new lines in the log (Figure 6-4). You will also notice that the curly braces are replaced with the actual values:

```
@GetMapping("/owners/{id}/pets")
public List<Pet> findAllPets(@PathVariable("id") int ownerId) {
   log.info("Finding owner with id: {}", ownerId);
   Optional<Owner> owner = ownerRepository.findById(ownerId);
   if (owner.isPresent()) {
      var pets = owner.get().getPets();
      log.info("Found {} pets for owner", pets.size());
      return pets;
   }
   return null;
}
```

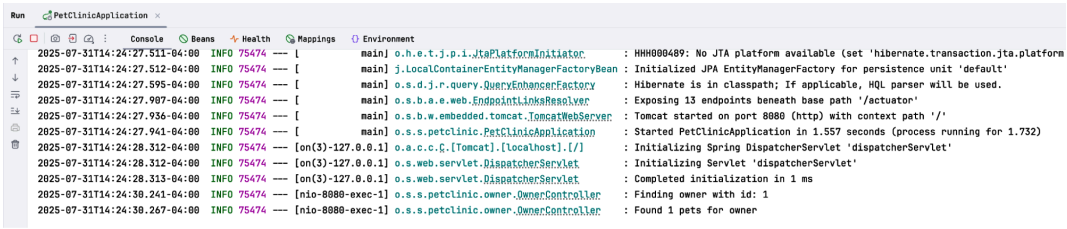

Figure 6-4. IntelliJ IDEA console displaying the log messages from the `findAllPets` method

Debugging

Your IDE's debugging features are invaluable for stepping through code execution, especially when you need to understand complex business logic. Let's revisit the pet creation validation from earlier:

```
@PostMapping("/pets/new")
public String processCreationForm(Owner owner, @Valid Pet pet,
BindingResult result
      RedirectAttributes redirectAttributes) {

   // Set breakpoint here
   if (StringUtils.hasText(pet.getName()) && pet.isNew()
   && owner.getPet(pet.getName(), true) != null)
      result.rejectValue("name", "duplicate", "already exists");

   // Additional validation...
}
```

You know that the application is rejecting a value for a new pet, but why? This conditional has multiple parts that all need to be true for a duplicate name error to occur. By setting a breakpoint on this line, you can step through and examine the following (Figure 6-5):

What each condition evaluates to
: Does `StringUtils.hasText(pet.getName())` return true? Is `pet.isNew()` true?

The state of objects
: What pets does this owner already have? What's the exact name being checked?

Method call results

What does `owner.getPet(pet.getName(), true)` actually return?

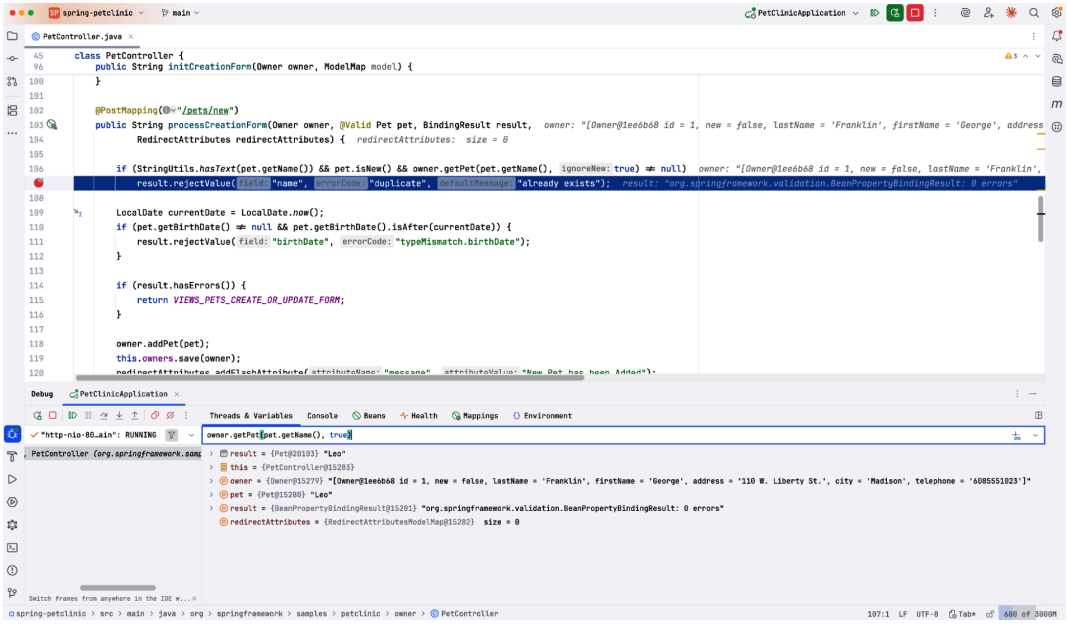

Figure 6-5. Using the debugger in IntelliJ to step through business logic

When you use debugging tools in your IDE to step through complex business logic, the debugger reveals not just *what* path the code takes, but *why* it takes that path. This is especially valuable when behavior doesn't match your expectations. For instance, if logs indicate that a pet name already exists but you're certain that's incorrect, the debugger allows you to verify this directly.

Locating external dependencies

Rarely will you find an application that runs in isolation. Modern applications depend on external services in the form of public APIs and internal services that they communicate with. An important part of understanding the execution flow is locating any and all of the external dependencies your application communicates with. This could be in the form of other APIs, databases, caches, and more.

Without a complete understanding of the entire application, how will you locate these dependencies? A really good place to start is by examining configuration files. In our PetClinic application that is built on Spring Boot, you can look at the *application.properties* or *application.yaml* configuration to see what dependencies are declared. In the following example, you can see that this application depends on a PostgreSQL database, an external service, and an email host:

```
# database
spring.datasource.url=jdbc:postgresql://db-server:5432/petclinic \
  spring.datasource.driver-class-name=org.postgresql.Driver
# external service
payment.service.endpoint=https://payments.example.com/api/v2
# email host
email.service.host=smtp.company.com
```

For containerized applications, check Docker Compose files or Kubernetes manifests to identify linked services:

```
# In docker-compose.yml
services:
  app:
    image: petclinic:latest
    depends_on:
      - mysql
```

```
- redis
- kafka
```

Locating internal frameworks and libraries

One of the biggest challenges in exploring existing codebases is encountering internal or homegrown frameworks and libraries. These are custom solutions built in-house to solve specific business problems or technical challenges. While they might have been the right decision at the time of creation, they often present unique obstacles for new team members trying to explore and modify the codebase.

Why would a team decide to build something internally versus going with an external solution? Internal frameworks typically emerge for several reasons:

- The original team needed functionality that wasn't available in existing libraries.
- They had specific security or performance requirements.
- The total cost of bringing in an external solution was too high.
- They built something to quickly solve a problem that evolved over time.
- They didn't know a solution existed or didn't fully understand it.

> Meta work is more interesting than real work.
>
> Neal Ford, director and software architect at Thoughtworks; author and speaker

Unfortunately, these solutions often suffer from poor documentation, lack of access to the original developers, and inconsistent maintenance.

Identifying internal frameworks

Internal frameworks can be harder to spot than external dependencies because they're embedded directly in the codebase. Here are the key indicators you're dealing with a homegrown solution:

Package naming patterns
: Look for packages with company-specific prefixes or generic names like `com.yourcompany.framework`, `utils.common.core`, or `internal.shared`.

Custom base classes
: Many classes extending from custom base classes with names like `BaseService`, `AbstractController`, or `CommonEntity` indicate framework code.

Custom annotations
: Nonstandard annotations that aren't part of popular frameworks often signal internal solutions:

```
@Entity
@CompanyTable(name = "users", audit = true)
public class User extends BaseCompanyEntity {

    @CompanyField(encrypted = true)
    private String email;

}
```

Utility classes with broad scope
: Classes named `AppUtils`, `BusinessHelper`, or `SystemManager` that handle diverse responsibilities across the application often.

Heavy configuration
: Extensive custom YAML/XML files or configuration classes that don't match standard library patterns.

Working with internal frameworks

The following tips will help you explore internal frameworks and leave the code in a better place than you found it:

Find usage examples first
: Search the codebase for how other developers use the framework. Real examples are often more instructive than documentation.

Look for tests
: Framework code sometimes has better test coverage since it needs to work across multiple scenarios. Tests reveal intended behavior and edge cases.

Document as you learn
: Create notes about what the framework does and how to use it. Even basic documentation helps future developers.

Check version control history
: Use `git blame` and commit messages to understand why the framework was built and how it evolved.

Evaluate the trade-offs
: Consider whether the framework still provides value or is becoming a maintenance burden. Sometimes replacing internal solutions with standard libraries is the right long-term choice.

Make incremental improvements
: Apply the scout rule (more on this later in the chapter) by improving naming, adding documentation, or enhancing test coverage as you work with the framework.

Build Mental Models Incrementally

A *mental model* is your internal representation of how the system works, and developing a robust model takes time and deliberate practice. If you're going to build a mental model of a system, it's much easier to do if you break down the system incrementally. Instead of trying to build a mental model of an entire ecommerce application, take a single path like the checkout process and build from there. It's also helpful to avoid letting tools or perfectionism get in the way. Often, a simple hand-drawn model on a scratch pad will suffice.

In this section, you will learn some practical methods for building mental models such as breaking complex systems into smaller ones, visualizing those models, and gradually expanding your knowledge.

AI Note

Another benefit these days of text-based diagramming is that generative AI can help you with an initial draft that you can refine further manually. You can describe your system or process to an AI tool and ask it to generate Mermaid syntax, then iterate on the output to match your specific needs.

Breaking down complex systems

Breaking down complex systems is an important part of building mental models. When you join a new project with a large codebase, *trying to understand everything at once is overwhelming and inefficient*. Instead, you should focus on breaking complex systems into manageable pieces.

You learned about application entry points in a previous section, and this is a great place to start. For example, in an ecommerce application, you might have defined some of these execution flows:

New customer flow
: A new user signs up, becomes a customer, and receives a promotional coupon.

Checkout flow
: What happens when a user completes a purchase with products in their cart.

Product review flow
: A customer provides feedback on a purchased product.

These are great examples of breaking a complex system into smaller manageable pieces that you can wrap your head around. Once you have seen how a new product review is recorded and you have seen the code associated with those steps, you can begin to build a mental model of this process. Everyone learns differently, and maybe you're someone who wants to build a mental model around the code and not a process. If your application uses a package-by-feature arrangement, you could build models of each feature. In the following example, we can see all of the code associated with the *cart* feature:

```
src/
├── cart/
│   ├── CartController.java
│   ├── CartService.java
│   ├── CartRepository.java
│   ├── Cart.java
│   ├── CartItem.java
│   └── dto/
│       ├── CartDTO.java
│       └── CartItemDTO.java
```

Visualizing your mental models

Visualizing code relationships can help you build these mental models. In Chapter 4, you learned about software modeling and some of the types of diagrams at your disposal. Your visualizations do not need to be professional: they can range from simple sketches to full UML diagrams. Use whatever level of detail helps you understand the system. The first part of visualizing your mental model is to choose your visualization type. Different types of visualizations serve different purposes in software engineering:

Flowcharts
: Help you understand sequential processes and decision points

Entity–relationship diagrams
: Clarify data structures and their relationships

Sequence diagrams
: Illustrate how components interact over time

Component diagrams

Show the high-level architecture and dependencies

Mind maps

Help organize related concepts hierarchically

You can begin by starting with a simple representation and add details as your comprehension of the system grows. For example, when visualizing the checkout flow in an ecommerce system, you might start with just the major components. The following is a simple flowchart written in Mermaid syntax, which is a popular Markdown-based diagramming tool:

```
graph TD
    A[Shopping Cart] --> B[Checkout Form]
    B --> C[Payment Processing]
    C --> D[Order Confirmation]
```

The previous code can generate a flowchart like the one shown in Figure 6-6.

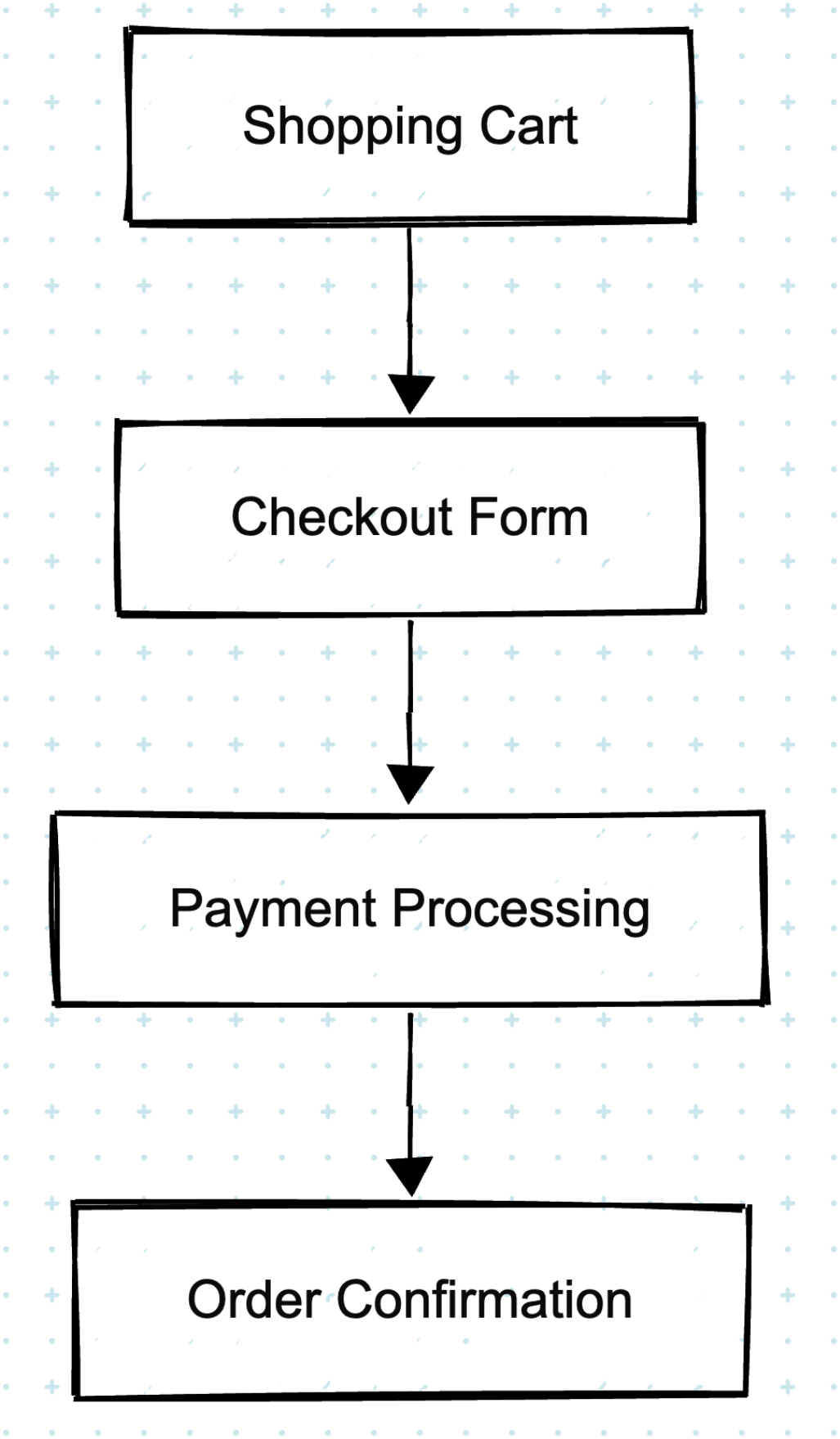

Figure 6-6. A flowchart for a sample checkout process in an ecommerce application

As you learn more, you can expand this flowchart to include specific services, database interactions, and external integrations. For example:

```
sequenceDiagram
    participant U as User
```

```
participant FE as Frontend
participant API as API Layer
participant CS as CartService
participant PS as PaymentService
participant OS as OrderService
participant DB as Database
participant PP as Payment Provider

U->>FE: Click "Checkout"
FE->>API: POST /checkout
API->>CS: validateCart()
CS->>DB: getCartItems()
DB-->>CS: cartItems
CS-->>API: validation result
API->>PS: processPayment()
PS->>PP: authorizeCharge()
PP-->>PS: authorization
PS-->>API: payment result
API->>OS: createOrder()
OS->>DB: saveOrder()
DB-->>OS: orderConfirmation
OS-->>API: order details
API-->>FE: success response
FE-->>U: Display confirmation
```

This code generates the flowchart shown in Figure 6-7.

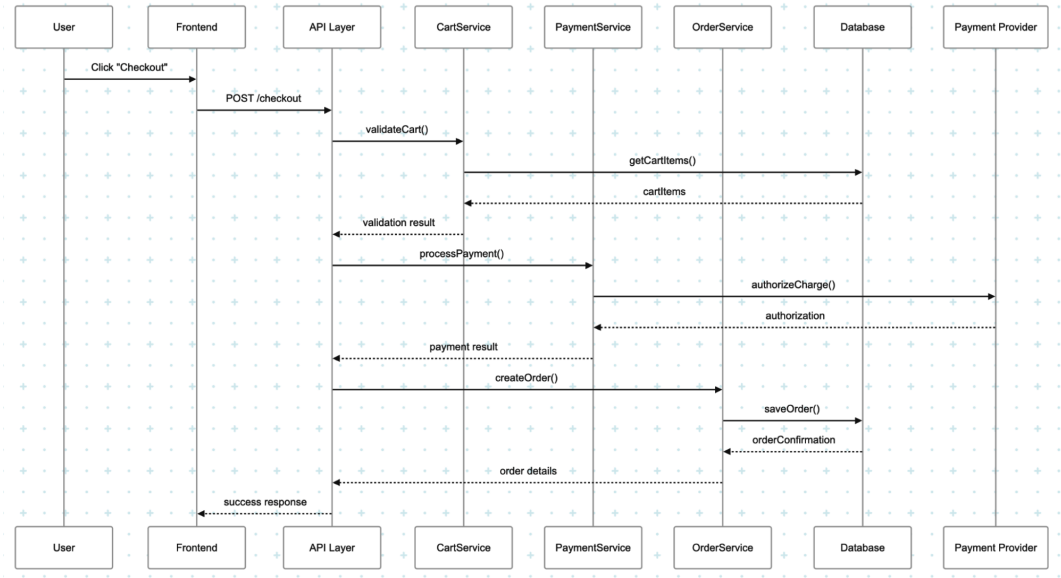

Figure 6-7. A larger flowchart for an ecommerce application

Gradually expanding your scope

Once you understand individual components, incrementally expand your focus to the larger system's behavior:

1. Start with single methods in isolation.
2. Move to classes and their direct dependencies.
3. Continue to feature-level flows that span multiple components.
4. Finally, understand cross-cutting concerns like security or transaction handling.

This incremental approach prevents cognitive overload while building a comprehensive view of the system.

By investing time in building accurate mental models, you'll make better decisions when modifying the codebase. Remember that this is an ongoing process and takes time. Even experienced team members continue refining their understanding of the system as a whole as it evolves. Now that you have learned how to understand unfamiliar codebases, let's look at a sample process for working with existing code.

A Sample Process

Over time, you will develop a feel for working with existing code. However, it can help to have a process. Christopher Judd teaches the following method in his boot camps:

1. Clone the project from the source code management system.
2. Review the README, coding standards, architecture, and other supporting documentation.
3. Take notes (architecture, build commands, common SQL, industry standards, terms, etc.) as you go. Don't be afraid to share these notes with your teammates!
4. Review build scripts.
5. Review the project dependencies.
6. Review the project structure (packages, namespaces, modules, artifacts, etc.).
7. Review the CI/CD pipelines.
8. Install any project dependencies (build tools, runtimes, languages).
9. From your IDE:
    a. Run and debug the application.
    b. Add breakpoints to interesting methods, connecting them back to the running application.
    c. Run and debug unit tests.
    d. Add breakpoints to interesting methods, connecting them back to the running application.
10. From the command line:
    a. Build the project artifacts.
    b. Run the unit tests.
    c. Start any required containers (such as the datastore).
    d. Run the application locally (ports, URLs, etc.).

Remember that understanding an existing codebase is an iterative process. You won't be able to comprehend everything on your first pass, and that is completely

normal. Each time you work on an issue or a new feature, you will gain deeper knowledge.

Making Changes Safely

Now that you have some tips and a process for understanding an existing codebase, it's time to learn how to safely modify one. Existing systems have likely gone through numerous iterations, features, and more. You can't just run into the ring like a wild boxer swinging away and expending all of your energy in the first round. You need to navigate the codebase methodically and be careful that you don't break things. You have likely heard the phrase "move fast and break things," and that might have a place in certain scenarios, but this is not one of them.

Refactoring Safely

When working with legacy systems and existing codebases, there is often potential for refactoring code to improve readability, performance, and more. In this situation, you want to make sure you have a plan and don't try to do too much at once.

Rely on existing tests

As you learned in the previous section, tests play a big role in understanding a codebase, but they also make your job of refactoring code safer and easier. If the project has a comprehensive suite of tests, you'll want to lean on them heavily. Tests serve as your safety net, allowing you to make small changes, run the tests, and ensure that you haven't affected something elsewhere in the project.

In the following example. you are refactoring the `calculateTotal` method to separate out the calculation of the subtotal and application of a discount. When refactoring, you should avoid modifying the tests since the goal is to improve the code structure while preserving its behavior. After each step of this refactor, you should run the tests associated with this method to ensure that behavior remains unchanged:

```
// Before refactoring
public double calculateTotal(Order order) {
    double total = 0;
    for (OrderItem item : order.getItems()) {
        total += item.getPrice() * item.getQuantity();
    }
    if (order.hasDiscount()) {
        total = total * (1 - order.getDiscountRate());
    }
    return total;
}

// After refactoring
public double calculateTotal(Order order) {
    double subtotal = calculateSubtotal(order);
    return applyDiscount(subtotal, order);
}

private double calculateSubtotal(Order order) {
    return order.getItems().stream()
        .mapToDouble(item -> item.getPrice() * item.getQuantity())
        .sum();
}

private double applyDiscount(double amount, Order order) {
```

```
    return order.hasDiscount()
        ? amount * (1 - order.getDiscountRate())
        : amount;
}
```

Add tests before refactoring

In the previous example, we leaned on the tests to ensure that nothing else was affected in the larger scope of the project. If the application doesn't have adequate test coverage on the changes you're proposing, this is your chance to add tests before refactoring any existing code. This approach, sometimes called *test-driven refactoring*, follows these steps:

1. Write tests that document the current behavior.
2. Verify that tests pass with the current implementation.
3. Refactor the code.
4. Verify that tests still pass with the new implementation.

The following unit test verifies that the order service correctly applies a 10% discount when calculating the total for an order that qualifies for discounting:

```
@Test
public void calculateTotalAppliesDiscountWhenOrderHasDiscount() {
    // Given
    OrderItem item1 = new OrderItem("Product1", 10.0, 2);
    OrderItem item2 = new OrderItem("Product2", 15.0, 1);
    Order order = new Order(Arrays.asList(item1, item2), true, 0.1);

    // When
    double total = orderService.calculateTotal(order);

    // Then
    assertEquals(31.5, total, 0.001); // (10*2 + 15*1) * 0.9 = 31.5
}
```

Adding tests like this before making changes helps you understand the existing behavior and gives you confidence that your refactoring preserves that behavior.

Tip

It's important to remember that when adding tests to existing codebases, your first goal is to document the existing behavior, not change it. Even if you know of a better way to refactor the code or fix something that you are sure is a bug, you should first write the tests to confirm the current behavior. You can then come back and address bugs or additional refactoring after your tests are in place.

The Scout Rule

The *scout rule* is a principle in software development that states: "Always leave the code better than you found it." It's inspired by the camping philosophy from the Boy Scouts of America, where scouts are taught to leave a campsite cleaner than they found it.

The scout rule encourages making small, incremental improvements to code quality whenever you touch a file. These improvements might include the following:

- Adding missing documentation
- Improving variable or method names for clarity
- Breaking down overly complex methods
- Removing dead code
- Fixing minor bugs you discover

In the following example, simply by improving the method name, adding type parameters, using an enhanced `for` loop, and adding documentation, you've made the code significantly more maintainable without changing its core functionality:

```
// Before applying the Scout Rule
public void prcs(List l) {
    for (int i = 0; i < l.size(); i++) {
        Object o = l.get(i);
        // Process object
        // ...
    }
}

// After applying the Scout Rule
/**
 * Processes a list of customer records and updates their status in the database.
 *
 * @param customers The list of customers to process
 */
public void processCustomers(List<Customer> customers) {
    for (Customer customer : customers) {
        // Process customer
        // ...
    }
}
```

While the scout rule encourages you to clean up code as you go, you need to find a balance between cleanup and the task at hand. In larger codebases, you could spend all your time simply improving code, only to realize you haven't fixed what you originally set out to fix.

Not all code that could be improved should be improved immediately. Consider these factors when deciding whether to refactor.

You can refactor code in these circumstances:

- The code you're working on is difficult to understand.
- You need to add a feature to the area.
- You're fixing a bug in the code.
- The code is causing performance issues.
- Multiple developers frequently work in this area.

You should consider deferring refactoring in these cases:

- The code works fine and rarely needs changes.
- You're under tight deadline pressure.
- The risk of breaking the code outweighs the benefits.

- The refactoring would require extensive changes beyond your current task.
- You don’t have adequate test coverage to ensure safety.

Small, Reversible Changes

Making incremental, easily reversible changes is the safest way to modify existing code without affecting the entire system. This is a skill you’ll need to continually practice to master. While you might be tempted to tackle complex problems all at once, and product owners might push for bigger, more impactful changes, your goal should be to make smaller modifications that don’t disrupt the system as a whole.

Change management strategies

A *change management strategy* for software teams is a structured approach to handling modifications in your applications. It’s essentially a system for proposing, reviewing, implementing, and tracking changes to maintain code quality and team alignment. When aligning with small, reversible changes, your change management strategy should prioritize safety and visibility.

Here are some practical strategies you can use in your change management approach:

Make changes visible
: Ensure that your changes are well-documented and easy to understand by others. This doesn’t need to be exhaustive and should explain what drove the change and how you implemented it.

Build in verification
: Include ways to verify that your changes work as expected. Verification methods for teams can include automated tests, code reviews, and static analysis.

Plan for rollback
: Murphy’s law states that what can go wrong will go wrong. Always have a plan to revert changes if problems arise. When you make smaller changes as we are proposing here, rolling them back should be easier.

Monitor effects
: Watch for unexpected consequences of your changes. This is where having good observability into your applications really pays off.

Small, testable increments

Once you get into the mindset of working in smaller increments that you can test, your job of modifying existing codebases becomes less stressful and easier to manage. When you adopt this approach, you will see the following benefits:

- Reduces risk by limiting the scope of each change
- Makes testing more focused and effective
- Makes code reviews more manageable
- Allows for easier troubleshooting if issues arise

This may mean pushing back on feature requests if they are too large and breaking them into smaller features to control the size of the change. Remember that it’s

better to deliver several small, successful changes than one large change that introduces bugs.

Let's say that you receive a request to modify the way your system will process an order. In the following example, you can see that the process order already does too much and is a candidate for refactoring. Instead of trying to change the logic for the entire method, consider breaking some of the functionality into smaller methods, writing tests for those methods, and validating that each works correctly before moving on to another one:

```
// Instead of changing this entire method at once...
public void processOrder(Order order) {
    // Validate order
    // Calculate totals
    // Apply discounts
    // Calculate shipping
    // Apply taxes
    // Process payment
    // Update inventory
    // Send confirmation
}

// Break it down into smaller, separate changes:
// 1. First, extract the validation logic
private boolean validateOrder(Order order) {
    // Validation logic here
    return isValid;
}

// 2. Next, extract the totals calculation
private double calculateOrderTotal(Order order) {
    // Total calculation logic here
    return total;
}

// 3. Continue with other extractions, testing after each change
```

Version control best practices

Most of the new projects you work on probably already have version control in place. In some scenarios, you might find yourself working with existing code that lacks proper version control or might need to establish best practices from the start:

Legacy codebases
: You may encounter older projects that have version control but lack best practices. These applications might commit everything to main with no branching strategy in place. They might contain inconsistent commit messages, massive unmerged changes, and repositories with generated files or dependencies that should have never been tracked in the first place.

Personal or small-scale projects
: When starting your own projects or working in very small teams, it's tempting to skip version control. You might think "it's just me" or "we can communicate directly," but establishing good habits early prevents technical debt and makes it easier when the project grows or when you need to onboard others.

Open source contributions
: When forking repositories or contributing to open source projects, you need to understand and follow the project's version control conventions while maintaining your own fork properly.

Contractor or consultant work

When joining existing projects, you may need to establish or improve version control practices that weren't previously prioritized.

No matter which scenario you find yourself in, version control systems like Git are essential for making safe changes to existing codebases. When used effectively, they provide safety nets that allow you to experiment, track history, and collaborate without fear of losing work or breaking functionality.

Commit strategically

When working with existing code, your commits should tell a story of how the code evolved:

Make atomic commits
: Each commit should represent a single logical change that can stand on its own. This makes it easier to understand, review, and, if necessary, revert changes.

Write meaningful commit messages
: A good commit message explains both what changed and *why*. Future developers (including your future self) will thank you for the context. The following code demonstrates good and bad commit message examples:

```
# Poor commit message
Fix bug

# Better commit message
Fix order calculation when a discount code is applied twice

The system was double-applying discount codes when customers
entered them in both mobile and desktop sessions. This fix
ensures the discount is only applied once per order.
```

Commit frequently
: Don't wait until you've made dozens of changes before committing. Small, frequent logical commits make it easier to identify when and where issues were introduced.

Branch wisely

Branching strategies will vary by organization, team, and project, but certain principles will apply when working on existing code:

Create feature branches
: Isolate your changes in a dedicated branch until they're ready for integration. This keeps the main branch stable and allows you to experiment freely.

Keep branches short-lived
: Long-lived branches diverge further from the main codebase over time, making integration more difficult. Aim to merge your changes back within days, not weeks.

Rebase before merging
: When appropriate, rebase your changes on top of the latest main branch to ensure you're working with the most current version of the codebase. *Important*: Rebasing rewrites commit history, so use it carefully on shared branches.

Pull requests and code reviews

A PR is a developer's proposal to merge changes from one branch into another, opening the work for review, discussion, and automated checks before integration. You will want to keep your PRs focused and manageable. A PR that changes 1,000 lines of code across 20 files is difficult to review. Instead, aim for smaller, focused PRs that address a single concern.

In the code review, make sure you explain what problem you're solving and how your changes address it. Include any references you have to relevant issues or requirements. Code reviews are about improving the solution, not criticizing the developer. Be open to suggestions and willing to iterate on your approach.

As you can see, although you shouldn't charge into modifying existing codebases with wild abandon, there's no need to be afraid of making a mark. Refactor safely with tests, follow the scout rule, and focus on small, reversible changes.

Wrapping Up

Working with existing code is one of those fundamental skills that can take you from coder to software engineer. Throughout this chapter, we've explored strategies for navigating, understanding, and safely modifying codebases. You've learned how to understand unfamiliar codebases by starting with the big picture, reviewing documentation, and grasping the project's architecture. We've covered how to follow the execution flows by finding entry points, tracing request journeys, and identifying external dependencies. You now know how to build mental models incrementally by breaking down complex systems and visualizing relationships. We also examined how to make changes safely through careful refactoring, following the scout rule, and using version control best practices.

These skills aren't just nice to have; they are essential for your growth as a software engineer. While creating new applications from scratch can be fun and exciting, the reality is that most of your career will involve working with existing code. Mastering these techniques will help you quickly and confidently contribute to any codebase, regardless of its size or complexity.

Putting It into Practice

Implementing what you've learned requires action, not just knowledge. The following practical steps will help you apply these code navigation and modification skills when working with existing code:

1. Find an open source project that interests you and spend 30 minutes exploring its codebase without writing any code. Focus on understanding its structure and organization.

2. Practice request tracing end-to-end by selecting a web application (like the PetClinic example) and tracing a complete user journey from UI interaction through API calls to database changes by using browser dev tools, logging, and debugging in sequence.

3. Pick a method from an open source project that's difficult to understand and refactor it to improve readability without changing its behavior.

4. Find a legacy project with minimal test coverage and practice adding tests that document existing behavior.

5. Create a mental model diagram by picking one business process from a real codebase and create a visual representation (hand drawing, flowchart, sequence diagram, or component diagram) that shows how the code actually flows, then validate it with a team member.

6. Make a complex change to an existing codebase by using small, reversible commits. Get feedback from a peer on your commit strategy.

7. Apply the scout rule to a project you're working on by identifying three small improvements you can make to leave the code better than you found it.

8. Practice safe dependency changes by identifying an external dependency in a project and researching how to safely upgrade or replace it, including impact analysis and rollback planning.

Additional Resources

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler (Addison-Wesley Professional, 2018)
- *Working Effectively with Legacy Code* by Michael Feathers (Prentice Hall, 2004)
- *Getting to Know IntelliJ IDEA* by Trisha Gee and Helen Scott (JetBrains, 2021)

[1] We'll be using the Spring PetClinic app for examples throughout this chapter. Even if you aren't an expert in Java or Spring, it should be fairly straightforward to understand.