chapter 3

# Feature-Driven Development—Practices

Answers: What do we have to manage?

*Good habits are a wonderful thing. They allow the team to carry out the basic steps, focusing on content and results, rather than process steps. This is best achieved when process steps are logical and their worth immediately obvious to each team member.*

Coad, LeFebvre, De Luca [Coad 99]

## Integrating Best Practices

Like all good software development processes, Feature-Driven Development (FDD) is built around a core set of "best practices." The chosen practices are not new but this particular blend of the ingredients is new. Each practice complements and reinforces the others. The result is a whole greater than the sum of its parts; there is no single practice that underpins the entire process. A team could choose to implement just one or two of the practices but would not get the full benefit that occurs by using the whole FDD process.

Consider inspections, for example. They are decades old and have a mountain of evidence showing them to be a great tool. However, on their own, they are far from enough. No, it is the *right mix* of the ingredients in the *right amounts* at the *right times* that makes the FDD cake taste so good!

**Mac:** *Steve, are the processes within* FDD *rigid, or can we adapt them to fit the project, the team, and the organization*?

**Steve:** FDD *can certainly be adapted to a particular toolset and to teams of people with various levels of experience. It is this level of flexibility that makes* FDD *relatively easy to adopt within an organization.*

**Mac:** *So, you are telling me that for our project, we can pick and choose what we will implement and that's* FDD?

**Steve:** *Absolutely not!* FDD *is flexible and adaptable, but there are some best practices that need to be included in order for it to be an* FDD *process. I'll list the best practices that make up* FDD. *Take a look at it, and then we'll go over each one in detail.*

## FDD Best Practices

The best practices that make up FDD are:

- Domain Object Modeling
- Developing by Feature
- Individual Class (Code) Ownership
- Feature Teams
- Inspections
- Regular Builds
- Configuration Management
- Reporting/Visibility of Results

***Steve:*** *You could choose to implement a few of the practices and claim you are following a feature-centric process but I reserve the* FDD *name to mean that you are following all of the practices I've listed.*

***Mac:*** *Okay, I can understand that distinction. Let's look at each practice in turn....*

### Domain Object Modeling

Domain object modeling consists of building class diagrams depicting the significant types of objects within a problem domain and the relationships between them. Class diagrams are structural in nature and look a little like the more traditional entity-relationship diagrams of the relational database world. Two big differences are the inclusion of inheritance or generalization/specialization relationships and operations that specify how the objects behave. To support this behavioral view, it is usual to complement the class diagrams with a set of high-level sequence diagrams depicting explicitly how objects interact with each other to fulfill their responsibilities. The emphasis is on what questions objects of a particular class can answer and what calculations or services they can perform; there is less emphasis placed on determining exactly what attributes objects of a particular class might manage.

As analysts and developers learn of requirements from Domain Experts, they start forming mental images of the desired system. Unless they are very careful, they make assumptions about this imaginary design. These hidden assumptions can cause inconsistencies between different people's work, ambiguities in requirements documentation, and the omission of important details. Developing an overall domain object model (Figure 3–1) forces those assumptions out into the open—misunderstandings are resolved, holes in understanding are filled, and a much more complete, common understanding of the problem domain is formed.

In *Extreme Programming Explained*, Kent Beck offers the analogy that software construction is like driving a car [Beck 00]. Driving requires con-

tinual small course adjustments, using the steering wheel; you cannot simply point a car in the right direction and press the accelerator. Software construction, Beck says, is similar. Extending that analogy a bit further, a domain object model is like the road map that guides the journey; with it, you can reach your destination relatively quickly and easily without too many detours or a lot of backtracking; without it, you can very quickly end up lost or driving around in circles, continually reworking and refactoring the same pieces of code.

The domain object model provides an overall framework to which to add function, feature by feature. It helps to maintain the conceptual integrity of the system. Using it to guide them, feature teams produce better initial designs for each group of features. This reduces the amount of times a team has to refactor classes to add a new feature.

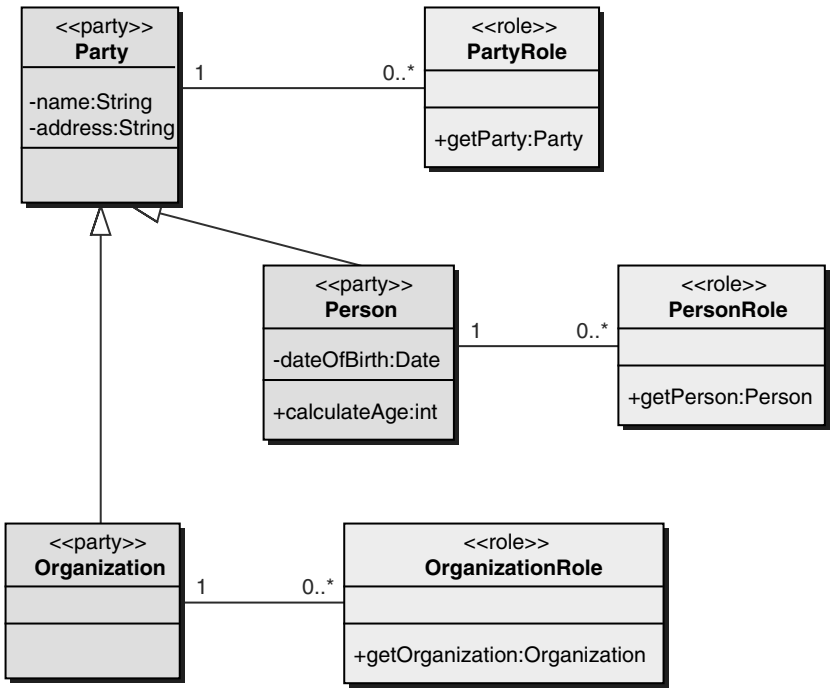


**Figure 3–1**
The early beginnings of a domain object model.

Domain Object Modeling is a form of object decomposition. The problem is broken down into the significant objects involved. The design and implementation of each object or class identified in the model is a smaller problem to solve. When the completed classes are combined, they form the solution to the larger problem.

The best technique the authors know for Domain Object Modeling is "modeling in color." Modeling in color uses four color-coded class arche-

types that interact in defined ways. The use of color adds a layer of "visually detectable" information to the model. Using this technique, a team or individual can very rapidly build a resilient, flexible, and extensible object model for a problem domain that communicates clearly and concisely.

FDD does not mandate the use of modeling in color and modeling in color does not require FDD. However, they do complement each other exceptionally well.

**Mac:** *I understand the analogy you used, but why is that so important to a software project? We have several very experienced programmers who will be working on our project. Some of them are used to listing the initial requirements, then going directly to coding. Sometimes they will prototype the system, but that's about as much of a model as they use.*

**Steve:** *For a very simple problem, that may be all right. However, the more complex the problem, the more imperative it is that the problem be adequately explored and explained. Source code is far too detailed a mechanism with which to do that. The information needs to be accessible to and understandable by all of those involved with specifying the requirements, as well as to those responsible for implementing them. A domain object model is a concise, relatively accessible, reuseable way of storing and communicating that information to everyone involved in the project.*

*The old "building a house" analogy really fits here. I wouldn't mind building a kennel without plans and blueprints, but would I want a builder to build my home that way? Or would you want to live in a 30-story high-rise that was built without blueprints?*

*The domain object model provides a solid framework that can be built within when changes in the business environment require the system to change. It allows designers to add new features and capabilities to the system correctly; it greatly enhances the internal quality and robustness of the system.*

### Developing by Feature

Once we have identified the classes in our domain object model, we can design and implement each one in turn. Then, once we have completed a set of classes, we integrate them and hey, presto! We have part of our system. Easy!...Well, it's a nice dream!

Nontrivial projects that are run in this way have found that they end up delivering a system that does not do what the client requires. Also, classes in these systems are often overly complicated, containing methods and attributes that are never used while missing methods and attribute that are needed. We can produce the most elegant domain object model possible, but if it does not help us to provide the system's clients with the functionality for which they have asked, we have failed. It would be like building a fantastic office skyscraper but either leaving each floor unfurnished, uncarpeted, and without staff, or furnishing it with ornamental but impractical furniture and untrained staff.

In the "Process and People" section at the beginning of Chapter 2, we said that a key element in any project is some statement of purpose, problem statement, or list of goals or very high-level requirements describing what the system needs to do. Without this, there is no reason for the project to exist. This is the functionality that the system must provide for the project to be considered a success.

Every popular method or process contains some form of functional decomposition activity that breaks down this high-level statement into more manageable problems. Functional specification documents, use case models and use case descriptions, and user stories and features all represent functional requirements, and each representation has its own advantages and disadvantages.

Traditionally, we have taken the statement of purpose and broken it down into a number of smaller problems and defined a set of subsystems (or modules) to solve those smaller problems. Then, for each subsystem, we have broken its problem into a hierarchical list of functional requirements. When we have requirements granular enough that we know how to design and implement each of them, we can stop decomposing the problem. We then start designing and implementing each of our functional requirements. The project is driven and tracked by function; sets of functional requirements are given to developers to implement, and their progress is measured.

A major problem is that the functional requirements tend to mix user interface, data storage, and network communication functions with business functions. The result is that developers often spend large amounts of time working on the technical features at the expense of the business features. A project that delivers a system with the greatest persistence mechanism but no business features is a failure.

A good solution to this problem is to restrict our lists of functional requirements to those of value to a user or client and to ensure that requirements are phrased in language that the user or client can understand. We call these *client-valued functions*, or *features*. Once the features for a system have been identified, they are used to drive and track development in FDD. Delivering a piece of infrastructure may be important—even critical—to the project but it is of no significance to the client because it has no intrinsic business value. Showing progress in terms of features completed is something that the client can understand and assign value to. Clients can also prioritize features in terms of significance to the business.

Interestingly, Extreme Programming records functional requirements as user stories on index cards. In *Extreme Programming Explained*, a user story was described as "a name and a short paragraph describing the purpose of the story" [Beck 00]. A year later, in *Planning Extreme Programming Explained*, a user story is "nothing more than an agreement that the customer and developers will talk together about a feature," and a user story is "a chunk of functionality that is of value to the customer" [Beck 01].

**Mac:** What about use cases? Don't they do the same thing? Aren't both FDD and Extreme Programming reinventing the wheel here? Ivar Jacobson introduced the software development world to use cases back in 1992 [Jacobson 92]. He defines a use case as "a description of a set of sequence of actions, including variants, that a system performs that yields an observable result to a particular actor," where an actor is defined as "a coherent set of roles that users of use cases play when interacting with these use cases" [Jacobson, 99]. I know this is a bit of a mouthful but it sounds like a feature to me.

**Steve:** A bit of a mouthful!?!?

**Mac:** Okay, a big mouthful. All it really means is that you:

1. Identify the users of the system (both humans and other computer systems)
2. Identify what each user does (tasks)
3. Categorize users according to their tasks to form a set of user roles (actors)
4. Describe how the system will help each user role perform each of its tasks (use cases)

In other words, a use case approach is user-centric. It groups functional requirements by the type of user of those functions. Driving a project with use cases helps us to ensure that we are developing what users need. This sounds like a step forward, in my opinion.

**Steve:** I agree, the thinking and ideas behind use cases are good, and they sound great in theory. However, despite numerous successes, many projects have struggled to apply use cases successfully in practice.

My main problem with use cases is that their definition does not define at what level of granularity use cases should be written and what format and level of detail their contents should take. The result has been continuous, raging debates, both within teams and on public online discussion forums. For example, in the first edition of UML Distilled: Applying the Standard Object Modeling Language, Martin Fowler writes, "Ivar Jacobson says that for a 10-person-year project, he would expect about 20 use cases....In a recent project of about the same magnitude, I had more than 100 use cases" [Fowler].

This problem becomes worse when the wrong people are asked to write use cases at the wrong time. A team of analysts and Domain Experts used to writing traditional functional specifications is often asked to write all the use cases for a system before any modeling or prototyping is done. The result is often an inconsistent and incomplete set of use cases of mixed granularity, with differing levels of detail, mixing user interface and persistence details with business logic in an overdetailed description of the design of an imaginary system.

**Mac:** Sounds like you are talking from personal experience. One popular answer to this problem is to define long, comprehensive templates to follow when writing the contents of a use case. However, this makes the task of writing use cases expensive in terms of project schedule, and increased manpower is also required to keep the use cases up to date throughout a project.

**Steve:** And Project Managers following that approach need to be very careful not to become bogged down in endlessly writing and rewriting use cases; what one of our colleagues, Bob Youngblood, calls "death by use cases." As with anything new, it's best to

*work with someone who knows what they are doing or at least to buy a book such as* Advanced Use Case Modeling [Miller] *and agree as a team to follow it.*

**Mac:** *So how do we avoid exactly the same problems with features? You've defined them as client-valued functions but there must be more to it than that, surely.*

**Steve:** *Yes...*

The term *feature* in FDD is very specific. A feature is a small, client-valued function expressed in the form:

<action> <result> <object>

with the appropriate prepositions between the action, result, and object.

### Features Are Small

They are small enough to be implemented within two weeks. Two weeks is the upper limit. Most features are small enough to be implemented in a few hours or days. However, features are more than just accessor methods that simply return or set the value of an attribute. Any function that is too complex to be implemented within two weeks is further decomposed into smaller functions until each sub-problem is small enough to be called a feature. Specifying the level of granularity helps to avoid one of the problems frequently associated with use cases. Keeping features small also means clients see *measurable* progress on a *frequent* basis. This improves their confidence in the project and enables them to give valuable feedback early.

### Features Are Client-Valued

In a business system, a feature maps to a step in some activity within a business process. In other systems, a feature equates to some step or option within a task being performed by a user.

Examples of features are:

- Calculate the *total* of a sale.
- Assess the *performance* of a salesman.
- Validate the *password* of a user.
- Retrieve the *balance* of a bank account.
- Authorize a *credit card transaction* of a card holder.
- Perform a *scheduled service* on a car.

As mentioned earlier, features are expressed in the form <action> <result> <object>. The explicit template provides some strong clues to the operations required in the system and the classes to which they should be applied. For example:

- "Calculate the *total* of a sale" suggests a calculateTotal() operation in a Sale class.
- "Assess the *performance* of a salesman" suggests an assessPerformance() operation in a Salesman class.
- "Determine the *validity of the password* of a user" suggests a determinePasswordValidity() operation on a User class that can then be simplified into a validatePassword() operation on the User class.

The use of a natural language, such as English, means that the technique is far from foolproof. However, after a little practice, it becomes a powerful source of clues to use in discovering or verifying operations and classes.

**Mac:** *So if I use the template to name my use cases and keep them to the two-week implementation limit, I would have the benefits of features and use cases? Use cases usually have preconditions, postconditions, and a description of what needs to happen. I could leave these empty to start with and fill them as development proceeds. That would avoid the analysis paralysis you warn of.*

**Steve:** *I suppose you could. I'm not sure what it buys you. One problem you might encounter by calling your features* use cases *is that you are going to confuse others who associate a different level of granularity, format, and application with the name* use case.

*Another problem is that, although nearly every expert I have spoken to recently advocates writing use cases in parallel with building a domain object model, most people still try to write them before doing any modeling or prototyping. In fact, many people advocate using use cases or functional requirements to drive the building of a domain object model.*

**Mac:** *Yes, I know, and I have seen the results many times! Function-heavy classes constantly accessing data-heavy classes, high coupling, low cohesion, and poor encapsulation. Yuck! I definitely prefer building the object model with Domain Experts first or at the same time as writing use cases. The functional decomposition and object-oriented decomposition are orthogonal approaches. Doing both helps to ensure that we deliver the function required within a structure that is robust and extensible.*

### Class (Code) Ownership

Class (code) ownership in a development process denotes who (person or role) is ultimately responsible for the contents of a class (piece of code).

There are two general schools of thought on the subject of code ownership. One view is that of individual ownership, where distinct pieces or groupings of code are assigned to a single owner. Every currently popular object-oriented programming language uses the concept of a class to provide encapsulation; each class defines a single concept or type of entity. Therefore, it makes sense to make classes the smallest

elements of code to which owners are assigned; code ownership becomes class ownership. This is the practice used within FDD; developers are assigned ownership of a set of classes from the domain object model.

## Note

*Throughout the rest of the book, we assume that the readers are using a popular object-oriented programming language such as Java, C++, Smalltalk, Eiffel, C#, etc. Therefore, we make the assumption that* classes *are the programming language mechanism providing encapsulation (also polymorphism and inheritance). Where this is not the case, the reader is requested to translate* class *to whatever fundamental element provides information hiding, abstract typing, or data encapsulation in your programming language.*

The advantages of individual class ownership are many but include the following:

- An individual is assigned the responsibility for the conceptual integrity of that piece of code. As enhancements and new methods are added to the class, the owner will ensure that the purpose of the class is maintained and that the modifications fit properly.
- There is an expert available to explain how a particular piece of code works. This is especially important for complex or business-critical classes.
- The code owner can implement an enhancement faster than another developer of similar ability who is unfamiliar with that piece of code.
- The code owner personally has something that he or she can take pride in doing well.

The first classic problem with class ownership occurs when developer A wants to make some changes to his or her classes, but those changes are dependent on other changes being made in the classes owned by developer B. Developer A could be required to wait a significant amount of time if developer B is busy. Too many of these situations would obviously slow down the pace of the development team.

The second potential problem with individual class ownership that is often raised is that of risk of loss of knowledge about a class. If the owner of a set of classes should happen to leave the project suddenly for some reason, it could take considerable time for the team to understand how that developer's classes work. If the classes are significant, it could put the project schedule under pressure.

At the opposite end of the code ownership spectrum is the view promoted by Extreme Programming proponents, among others. In this world, all the developers in the team are responsible for all of the code. In other words, the team has collective ownership of the source code.

Collective ownership solves the problem of having to wait for someone else to modify code and can ease the risk of someone leaving because, at least in a small system, more than one person has worked on the code.

The main issue with collective ownership, however, is that in practice, it can quickly degenerate into nonownership or an ownership dictated by few dominant individuals on the team. Either nobody ends up being responsible for anything in the system or the dominant few try to do all the work because, in their opinion, they are the only competent members of the team. If nobody takes responsibility for ensuring the quality of a piece of code, it is highly unlikely that the resulting code will be of high quality. If a few dominant developers try to do everything, they may start off well but will soon find themselves overloaded and suffering from burnout. Obviously, teams that encounter these problems struggle to continue to deliver frequent, tangible, working results.

**Mac:** *Hey, Steve. It appears that there is a swelling of opinion in the industry that says any team member should be allowed to change any piece of code. Many of our developers seem to like this idea, but* FDD *promotes individual class ownership. Is collective ownership an option?*

**Steve:** *Supporters of collective ownership claim that it works when combined with other complementary practices* (see [Beck]):

- Pair programming—*two developers working together at one personal computer or terminal to reduce the likelihood of introducing errors into the code and to shorten the time it takes a developer to learn and understand the system.*
- Extensive unit testing *to verify that new code functions as required and that refactored or updated code still functions as required.*
- Coding standards compliance *to improve the readability of code and to minimize errors due to misunderstanding of existing code.*
- Continual integration *of changes into the code base to reduce the likelihood that multiple programmer pairs need to access the same piece of code at the same time.*

*However, if we assume collective ownership, look at what could happen as developers work on features. For any given feature, the developer (or pair of developers) can add or modify operations and attributes of the classes that participate in the feature. A different pair, working on a different feature, can add or modify another operation in some of those same classes* (Figure 3–2). *In a large team, each method of a class could theoretically end up being written by a different developer.*

**Mac:** *My alarm bells are ringing at this point! With small classes and small teams, we might get away with this but on larger, more significant classes with a larger team,*

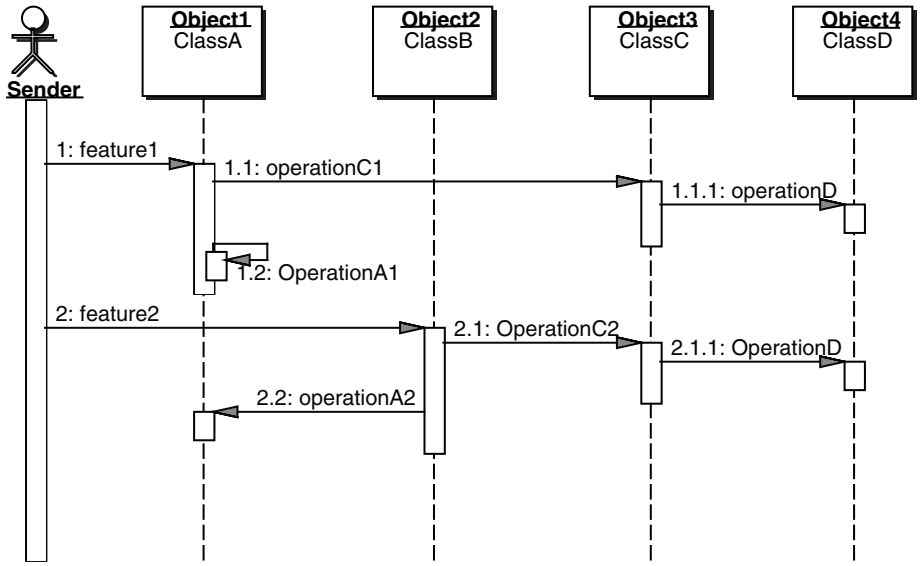


**Figure 3–2**
Features adding and enhancing operations in the same classes.

*consistency and conceptual integrity, not to mention robustness, could become a major problem.*

***Steve:*** *Exactly! The more minds working on a piece of work over time, the harder it is to maintain the conceptual integrity of that work* [Brooks]. *I believe the chances of the class evolving a consistent, elegant, efficient set of methods are greatly reduced if anyone and everyone can write a piece of it. I also think the need for rework and refactoring is going to be greatly increased.*

*Also, to modify a class correctly, the modifiers have to understand how its internals work. This can take time if those developers have not seen the class before or have not worked with it for a while. This is obviously going to take longer than if someone familiar with the code did the modification.*

***Mac:*** *It sounds like individual class ownership is more likely to scale to our size of project and team. However, I can see that, in some cases within a project, collective ownership of parts of the model could be advantageous. Can we use combinations of individual and collective ownership and still call it* FDD? *Does* FDD *allow me to tailor the class ownership practice to the needs and structure of my team and organization?*

***Steve:*** *You're not going to get arrested by the thought police, if that's what you mean. Also, let's not get hung up over a name of a process. We need to do what works for us and our organization. Having said that, I think as we cover the other practices in* FDD, *you'll find less and less of a reason to need collective ownership. The only areas where I personally might consider collective ownership is when building proof of concept prototypes for the technical architecture and user interface. When it comes to production code, I want to know that there is a single responsible person I can go to when there are issues with a particular class.*

***Mac:*** *I suppose there is no way of getting the benefits of both individual class ownership and collective ownership, is there? Or at least get close to that ideal?*

***Steve:*** *Actually, I think the answer to that is yes! We need to combine class ownership with the use of feature teams and inspections.*

### Feature Teams

Building a domain object model identifies the key classes in the problem domain. The class ownership practice assigns those classes to specific developers. We also know that we want to build, feature by feature.

So how do we best organize our class owners to build the features?

We assigned classes to owners to ensure that there was a single person responsible for the development of each class. We need to do the same for features. We need to assign each feature to an owner—somebody who is going to be responsible for ensuring that the feature is developed properly. The implementation of a feature is likely to involve more than one class and, therefore, more than one Class Owner. Thus, the feature owner is going to need to coordinate the efforts of multiple developers—a team lead job. Therefore, we pick some of our better developers, make them team leaders, and assign sets of features to each of them (we can think of a team leader as having an "inbox" of features that he or she is responsible to deliver).

Now that we have Class Owners and team leaders, let's form the development teams around these team leaders. Ah! We have a problem! How can we guarantee that all the Class Owners needed to code a particular feature will be in the same team? This is not an easy problem to solve.

We have four options:

1. We can go through each feature, listing the classes we think are involved, then try to separate the features into mutually exclusive sets. This feels like a good deal of design work just to form teams, and what if we get it slightly wrong? What if there are no convenient, mutually exclusive groupings of features? This does not sound like a repeatable step in the process.
2. We can allow teams to ask members of other teams to make changes to the code they own. However, now we are likely to be waiting for another developer in another team to make a change before we can complete our task. This is exactly the situation that led Extreme Programming to promote collective ownership.
3. We can drop class ownership and go with collective ownership and everything else that it requires to make it work. There is already a book in this series covering this option,

[Astels] and anyway, we know that collective ownership does not scale easily.

4. We can change the team memberships whenever this situation occurs so that a team leader always has the Class Owners he or she needs to build a feature. This is the only realistic option that will allow us both to develop by feature and to have Class Owners.

Actually, there is nothing that requires us to stick to a statically defined team structure. We can change to a more dynamic model. If we allow team leaders to form a new team for each feature they start to develop, they can pick the Class Owners they need for that feature. Once the feature is fully developed, the team is disbanded, and the team leader picks the Class Owners needed to form the team for the next feature. This can be repeated indefinitely until all the features required are developed.

This is a form of dynamic matrix management. Team leaders owning features pick developers based on their expertise (in this case, class ownership) to work in the *feature team* developing those features involving their classes (Figure 3–3).

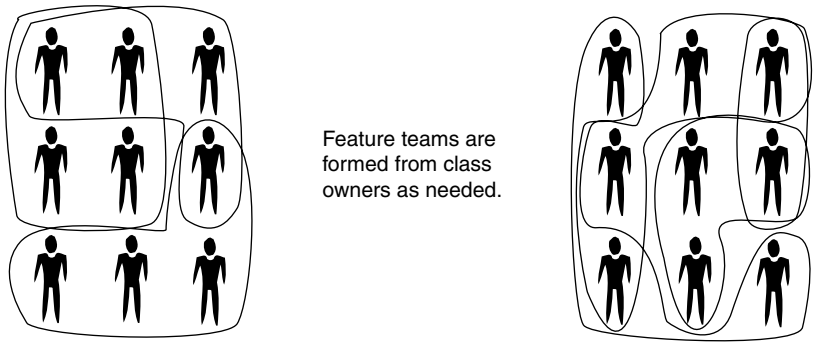


**Figure 3–3**
Feature teams.

**Mac:** *So are feature teams new? I remember that Harlan Mills suggested the idea of Chief Programmer teams back in 1971 [Brooks]. His idea is based on surgical teams, where a surgeon is supported by a number of talented and qualified people, each performing a specific role. In a Chief Programmer team, developers each have their own specific responsibility and support a lead developer.*

**Steve:** *Feature teams are similar but differ in two important aspects:*

- *The team leader acts as more of a coach for the team than some superprogrammer in charge of a bunch of junior or trainee programmers.*
- *The Class Owners' responsibilities are all similar to each other; it is the code that they are responsible for that differs.*

*Every member of a feature team is responsible for playing their part in the success of the team. However, feature team leaders, as all good coaches know, are ultimately re-*

*sponsible for producing results. They own the features, and they are accountable for their successful delivery. Playing this team leader role well normally requires both ability and experience, so we call our feature team leaders* Chief Programmers *in recognition of this and* Mills' [Brooks] *work.*

**Mac:** *Steve, this sounds incredibly flexible and may be the answer to a lot of the problems we have experienced in the past on our software projects, but what happens when a Chief Programmer guesses incorrectly about which classes are involved in the development of a feature? For example: What if a Chief Programmer thought three classes were needed to implement a given feature, and it turns out that two more are involved?*

**Steve:** *Easy! All the Chief Programmer needs to do is contact the appropriate Class Owner if they are not already in the team, verify their availability to work on the feature, and include them on the feature team. The Chief Programmer may have to discuss the availability of the extra developers with other Chief Programmers, if those developers are heavily loaded.*

**Mac:** *What happens if the Class Owners are working in too many teams and cannot take on another feature team for a few days? Does the feature team block?*

**Steve:** *Well, that feature team may block. However, remember that each Chief Programmer has this inbox of features assigned to him or her. If the Class Owners are not available to develop one feature, the Chief Programmer can pick a different feature to develop next, instead.*

**Mac:** *Even more flexible! I like this idea a lot! I guess, though, that there are some restrictions about which features a Chief Programmer can develop next.*

**Steve:** *Yes, there will be some dependencies between features to watch for, and some features will be higher priority than others and will need to be developed sooner rather than later, but that is about it.*

**Mac:** *Is there anything else interesting about feature teams?*

**Steve:** *A couple more points...*

Some things to note about feature teams:

- A feature team, due to the small size of features, remains small, typically three to six people.
- By definition, a feature team is comprised of all the Class Owners who need to modify or enhance one of their classes as part of the development of a particular feature. In other words, the feature team owns all the code it needs to change for that feature. There is no waiting for members of other teams to change code. So we have code ownership and a sense of collective ownership, too.
- Each member of a feature team contributes to the design and implementation of a feature under the guidance of a skilled, experienced developer. Applying multiple minds to evaluate multiple options and select the design that fits best reduces the risk of reliance on key developers or owners of specific classes.

- From time to time, Class Owners may find themselves members of multiple feature teams at the same time. This is not the norm but is not a problem, either. While waiting for others in one feature team, a Class Owner can be working on stuff for another feature team. Most developers can handle belonging to two or even three features teams concurrently for a short period of time. More than that leads to problems switching context from one team to another. Chief Programmers work together to resolve any problematic conflicts and to avoid overloading any particular developer.
- Chief Programmers are also Class Owners and take part in feature teams led by other Chief Programmers. This helps Chief Programmers to work with each other and keeps them close to the code (something most Chief Programmers like).

### Inspections

FDD relies heavily on inspections to ensure high quality of designs and code. Many of us have sat through hours of boring, backbiting, finger-pointing sessions that were called *code reviews*, *design reviews*, or *peer reviews* and shudder at the thought of another process that demands inspections. We have all heard comments such as "Technical inspections, reviews, walkthroughs are a waste of time. They take too long, are of little real benefit, and result in too many arguments" or "I know my job! Why should I let others tell me how to design and write my code?"

However, when done well, inspections are very useful in improving the quality of design and code. Inspections have been recommended since the 1970s, and the evidence weighs heavily in their favor.

> *The Aetna Insurance Company found 82% of the errors in a program by using inspections and was able to decrease its development resources by 25%.*
>
> M.E. Fagan [Fagan]

> *In a group of 11 programs developed by the same group of people, the first 5 were developed without inspections. The remaining 6 were developed with inspections.*
>
> *After all the programs were released to production, the first 5 had an average of 4.5 errors per 100 lines of code. The 6 that had been inspected had an average of only 0.82 errors per 100 lines of code.*
>
> *Inspections cut the errors by over 80%....*
>
> *In a software-maintenance organization, 55% of one-line maintenance changes were in error before code inspections were introduced. After inspections were introduced, only 2% of the changes were in error.*
>
> D.P. Freedman and G.M. Weinberg [Freedman]

> *IBM's 500,000-line Orbit project used 11 levels of inspections. It was delivered early and had only about 1% of the errors that would normally be expected.*
>
> T. Gilb [Gilb 88]

> *The average defect detection rate is only 24% for unit testing, 35% for function testing, and 45% for integration testing. In contrast, the average effectiveness of design and code inspections is 55 and 60% respectively.*
>
> C.L. Jones [Jones]

> *One client found that each downstream software error cost on average 5 hours. Others have found 9 hours (Thorn EMI, Reeve), 20 to 82 hours (IBM, Remus), and 30 hours (Shell) to fix downstream. This is compared to the cost of only one hour to find and fix using inspection.*
>
> T. Gilb and D. Graham [Gilb 93]

Need we say more?

Actually, there is a little more to say. The primary purpose of inspections is the detection of defects. When done well, there are also two very helpful secondary benefits of inspections:

1. Knowledge transfer. Inspections are a means to disseminate development culture and experience. By examining the code of experienced, knowledgeable developers and having them walk through their code, explaining the techniques they use, less experienced developers rapidly learn better coding practices.
2. Standards conformance. Once developers know that their code will not pass code inspection unless it conforms to the agreed design and coding standards, they are much more likely to conform.

> *Even though coding standards can be written (presumably by experienced developers) and distributed, they will not be followed (or maybe not even read) without the sort of encouragement provided by inspections.*
>
> Steve McConnell [McConnell 98]

We can make inspections even more useful by collecting various metrics and using them to improve our processes and techniques. For instance, as metrics on the type and number of defects found are captured and examined, common problem areas will be revealed. Once these problem areas are known, this can be fed back to the developers, and the development process can be tweaked to reduce the problem.

Of course, the catch is that little qualifying phrase that we have used a couple of times in the last few paragraphs—"when done well." Chap-

ter 10, section titled "Verification: Design Inspection," and Chapter 11, section titled "Conduct a Code Inspection," provide hints and tips for achieving exactly this. We make a couple more general points here.

Inspections have to be done in a way that removes the fear of embarrassment or humiliation from the developer whose work is being inspected. Few developers like to be told that something they have sweated over for hours is wrong or could have been done better. Setting the inspection culture is key. Everyone needs to see inspections primarily as a great debugging tool and secondly as a great opportunity to learn from each other. Developers also need to understand that inspections are not a personal performance review[McConnell 93].

Inspections complement the small team and Chief Programmer–oriented structure of FDD beautifully. The mix of feature teams and inspections adds a new dimension. An entire feature team is on the hot seat, not just one individual. This removes much of the intensity and fear from the situation. The Chief Programmer controls the level of formality of each inspection, depending on the complexity and impact of the features being developed. Where design and code have no impact outside the feature team, an inspection will usually involve only the feature team members inspecting each other's work. Where there is significant impact, the Chief Programmer pulls in other Chief Programmers and developers both to verify the design and code and to communicate the impact of that design and code.

### Regular Build Schedule

At regular intervals, we take all of the source code for the features that we have completed and the libraries and components on which it depends, and we build the complete system.

Some teams build weekly, others daily, and still others continuously. It really depends on the size of the project and the time it takes to build the system. If a system takes eight hours to build, a daily build is probably more than frequent enough.

A regular build helps to highlight integration errors early. This is especially true if the tests built by the feature teams to test individual features can be grouped together and run against the completed build to smoke out any inconsistencies that have managed to find their way into the build.

A regular build also ensures that there is always an up-to-date system that can be demonstrated to the client, even if that system does only a few simple tasks from a command line interface. Developing by feature, of course, also means that those simple tasks are of discernible value to the client.

A regular build process can also be enhanced to:

- Generate documentation using tools such as JavaSoft's Javadoc or Together's greatly enhanced documentation-generation capability.
- Run audit and metric scripts against the source code to highlight any potential problem areas and to check for standards compliance.
- Be used as a basis for building and running automated regression tests to verify that existing functionality remains unchanged after adding new features. This can be invaluable for both the client members and the development team.
- Construct new build and release notes, listing new features added, defects fixed, etc.

These results can then be automatically published on the project team or organization's intranet so that up-to-the-minute documentation is available to the whole team.

### Configuration Management

Configuration management (CM) systems vary from the simple to the grotesquely complex.

Theoretically, an FDD project only requires a CM system to identify the source code for all the features that have been completed to date and to maintain a history of changes to classes as feature teams enhance them.

Realistically, a project's demands on a CM system will depend on the nature and complexity of the software being produced; for example, whether multiple versions of the software need to be maintained, whether different modules are required for different platforms or different customer installations, and so on. This is not explicitly related to the use of FDD; it is just business as usual on any sophisticated software development project where work is being done on different versions of a software system simultaneously.

It is a common fundamental mistake, however, to believe that only source code should be kept under version control. It is as important (maybe more important) to keep requirements documents, in whatever form they take, under version control so that a change history is maintained. This is especially true if the requirements form a legal commercial contract between two organizations.

Likewise, analysis and design artifacts should be kept under version control so that it is easy to see why any changes were made to them.

Test cases, test harnesses and scripts, and even test results should also be versioned-controlled so that history can be reviewed.

Any artifact that is used and maintained during the development of the system is a candidate for version control. Even contract documents with clients of the system that document the legal agreement for what is being built are candidates for versioning. The version of the process you are using and any changes and adjustments that may be made during the construction and maintenance of the system may need to be versioned and variances documented and signed by Project Managers or Chief Programmers. This is especially true for systems that fall under regulation of such governmental bodies as the U.S. Food and Drug Administration.

### Reporting/Visibility of Results

*Closely related to project control is the concept of "visibility," which refers to the ability to determine a project's true status....If the project team can't answer such questions, it doesn't have enough visibility to control its project.*

*The working software is a more accurate status report than any paper report could ever be.*

Steve McConnell [McConnell 98]

It is far easier to steer a vehicle in the right direction if we can see precisely where we are and how fast we are moving. Knowing clearly where we are trying to go also helps enormously.

A similar situation exists for the managers and team leaders of a software project. Having an accurate picture of the current status of a project and knowing how quickly the development team is adding new functionality and the overall desired outcome provides team leads or managers with the information they need to steer a project correctly.

FDD is particularly strong in this area. FDD provides a simple, low-overhead method of collecting accurate and reliable status information and suggests a number of straightforward, intuitive report formats for reporting progress to all roles within and outside a project.

Chapter 5, "Progress," is dedicated to the subject of tracking and reporting progress on an FDD project, so we postpone any further discussion on the subject until then.

## Summary

FDD blends a number of industry-recognized best practices into a cohesive whole. The best practices used in FDD are:

- *Domain Object Modeling*—a thorough exploration and explanation of the domain of the problem to be solved, resulting in a framework within which to add features.

- *Developing by Feature*—driving and tracking development through a functionally decomposed list of small, client-valued functions.
- *Individual Class Ownership*—having a single person who is responsible for the consistency, performance, and conceptual integrity of each class.
- *Feature Teams*—doing design activities in small, dynamically formed teams so that multiple minds are always applied to each design decision, and multiple design options are always evaluated before one is chosen.
- *Inspections*—applying the best-known defect-detection technique and leveraging the opportunities it provides to propagate good practice, conventions, and development culture.
- *Regular Builds*—ensuring that there is always a demonstrable system available and flushing out any integration issues that manage to get past the design and code inspections. Regular builds provide a known baseline to which to add more function and against which a quality assurance team can test.
- *Version Control*—identifying the latest versions of completed source code files and providing historical tracking of all information artifacts in the project.
- *Progress Reporting*—frequent, appropriate, and accurate progress reporting at all levels, inside and outside the project, based on completed work.

In the next chapter, we look at exactly how these practices blend together to form the five FDD processes.