6 Feature-Driven Development

▶ **I feel a recipe is only a theme which an intelligent cook can play each time with a variation.**

Madame Benoit

▶ **The ultimate judgment of progress is this: measurable results in reasonable time.**

Robert Anthony

▶ **I measure output, not input.**

Lim Bak Wee

For enterprise-component modeling to be successful, it must live and breathe within a larger context, a software development process.

We've developed such a process in practice, and we detail it in this chapter. We present Feature-Driven Development (FDD) in these sections:

1. The problem: accommodating shorter and shorter business cycles
2. The solution: feature-driven development
3. Defining feature sets and features
4. Establishing a process: why and how
5. The five processes within FDD
6. Chief programmers, class owners, and feature teams
7. Management controls: Tracking progress with precision

6.1 THE PROBLEM: ACCOMMODATING SHORTER AND SHORTER BUSINESS CYCLES

Despite the many advances in software development, it is not uncommon for projects lasting two or more years to use a function-driven process: from functional specs (in traditional paragraph format or in use-case format) to design to code to test to deployment. Along the way, some have made minor modifications to the theme, allowing some influence from iterations. Nevertheless, many software projects exceed budget, blow schedule, and deliver something less than desired (something appropriate two years earlier, yet no longer).

As if that weren't enough pressure, the ever-increasing pace of technological advances makes it less and less likely that a project lasting more than two years will ever succeed.

In fact, more and more, we are mentoring projects with total schedules of 90, 120, or 180 days—or perhaps 9, 12, or 18 months. One market-leader we work with considers any project longer than 180 days as high-risk. Why? Their business changes so rapidly and the supporting technology changes so rapidly that planning nine months out adds risk to the project.

That's quite a change in perspective.

The authors of *BLUR: The Speed of Change in the Connected Economy* put it this way:

> Speed is the foreshortening of product life cycles from years to months or even weeks. . . . Accelerated product life cycles and time-based competition have become part of the business lingo. . . . The faster things move, the less time you have to plan for them. You're much better off iterating and reiterating, adjusting as you go.
>
> STAN DAVIS AND CHRISTOPHER MEYER [DAVIS98]

The norm for fast-cycle-time projects is a feature-driven iterative process, beginning with features and modeling, followed by design-and-build increments.

In this chapter, we formalize the process we call "Feature-Driven Development" (FDD).

We've developed FDD in practice. Project teams apply it with significant success.

Developers like it. With FDD, they get something new to work on every two weeks. (Developers love new things.) With FDD, they get closure every two weeks. Closure is an important must-have element for job satisfaction. Getting to declare "I'm done" every two weeks is such a good thing.

Managers like it too. With FDD, they know what to plan and how to establish meaningful milestones. They get the risk-reduction that comes from managing a project that delivers frequent, tangible, working results. With FDD, they get real percentage numbers on progress, for example, being 57% complete and demonstrating to clients and to senior management exactly where the project is.

Clients like it too. With FDD, they see plans with milestones that they understand. They see frequent results that they understand. And they know exactly how far along the project is at any point in time.

Yes, developers *and* managers *and* clients like FDD. Amazing yet true.

6.2 THE SOLUTION: FEATURE-DRIVEN DEVELOPMENT

What if you and your team adopted a process for delivering frequent, tangible, working results?

Think about it. You could plan for results, measure results, measure your progress in a believable way, and demonstrate working results.

What might this mean for you and your career, the morale of your team, and added business from your clients? Plenty of motivation!

FDD is a model-driven short-iteration process. It begins with establishing an overall model shape. Then it continues with a series of two-week "design by feature, build by feature" iterations.

The features are small "useful in the eyes of the client" results.

Most iterative processes are anything but short and "useful in the eyes of the client." An iteration like "build the accounting subsystem" would take too long to complete. An iteration like "build the persistence layer" is not (directly at least) client-valued.

Moreover, long and IT-centric iterations make life difficult. It's harder to track what's really going on during an iteration. And it's harder to engage the client, not having a steady stream of client-valued results to demonstrate along the way.

In contrast, a small feature like "assign unique order number" is both short and client-valued. In fact, a client knows exactly what it is, can assign a priority to it, can talk about what is needed, and can assess whether or not it truly meets the business need.

A small feature is a tiny building block for planning, reporting, and tracking. It's understandable. It's measurable. It's do-able (with several other features) within a two-week increment.

As in any other development process, FDD prescribes a series of steps and sub-steps. Unlike other processes, FDD uniquely:

- uses very small blocks of client-valued functionality, called features (allowing users to describe what they want in short statements, rather than having to force those thoughts into a "the user does this, the system does that" format),
- organizes those little blocks into business-related groupings (solving the dilemma of what level one should write use-cases for),

- focuses developers on producing working results every two weeks,
- facilitates inspections (making inspections, a best practice, easier to accept and simpler to apply),
- provides detailed planning and measurement guidance,
- promotes concurrent development within each "design by feature, build by feature" increment,
- tracks and reports progress with surprising accuracy, and
- supports both detailed tracking within a project and higher-level summaries for higher-level clients and management, in business terms.

6.3 DEFINING FEATURE SETS AND FEATURES

A *feature* is a client-valued function that can be implemented in two weeks or less.

We name a feature using this template:

<action> the <result> <by|for|of|to> a(n) <object>

where an object is a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)

For example,

- Calculate the total of a sale.
- Assess the fulfillment timeliness of a sale.
- Calculate the total purchases by a customer.

A *feature set* is a grouping of business-related features. We name a feature set this way:

<action><-ing> a(n) <object>

An example is "making a product sale."

And we name a major feature set this way:

<object> management

An example is "product-sales management."

We start an informal features list while developing the overall model. We write down features we hear from domain members and glean content from documents we are working with.

We build a detailed features list after developing an overall model. Some features come by transforming methods in the model to features. Most features come from considering each pink moment-interval (business areas) and writing down the features.

For example, see the model snippet in Figure 6-1.

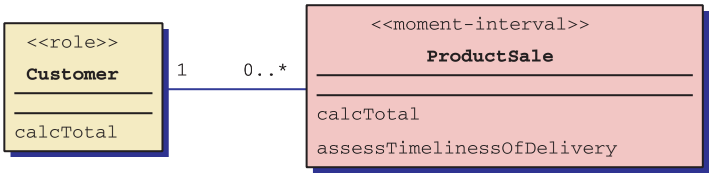


FIGURE 6-1. ▲ A model snippet.

We could transform its methods into:

- Feature set
  Making a product sale to a customer
- Features
  Calculate the total of a sale.
  Assess fulfillment timeliness for a sale.
  Calculate the total purchases by a customer.

Yet we can do even more, considering additional features that will better satisfy client wants and needs. Here's an example:

- Major feature set
  Product-sale management
- Feature set
  Making a product sale to a customer
- Features
  Calculate the total of a sale.
  Assess the fulfillment timeliness for a sale.
  Calculate the total purchases by a customer.
  Calculate the tax for a sale.
  Assess the current preferences of a customer.

For each additional feature, we add corresponding methods to the model. Normally we don't do this right away, but rather during the "design by feature, build by feature" iterations.

In practice, we've seen again and again that building an overall model and an informal features list before developing a detailed features list:

- brings domain members together to talk with each other, listen to each other, and develop a common model of the business—before developing a fully detailed features list,
- increases developer members' understanding about the domain and how things interrelate within it (even if they have built systems in the domain before),

- fosters more creativity and innovation (visual models in color engage spatial thinking, a creativity must-have before moving into linguistic and mathematical-logical thinking),
- encourages exploring "what could be done, what might be done, and what could make a real difference" before locking oneself into a fixed system boundary ("the user does this, the system does that"), and
- leads to the discovery of feature sets and features that bring significant business advantage, rather than passively scribing down the needs for yet another system.

6.4 ESTABLISHING A PROCESS: WHY AND HOW

This section explores these questions:

1. Why use a process?
2. Who selects tools for a process?
3. How might one describe a process?

6.4.1 Why Use a Process?

We think most process initiatives are silly. Well-intentioned managers and teams get so wrapped up in executing process that they forget that they are being paid for results, not process execution.

Process for process' sake alone, as a matter of "process pride," is a shame. Having hundreds of pages of steps to execute demoralizes the team members, to the point that they willingly turn off their minds and simply follow the steps.

Process over-specification does far more harm than good. The process takes on a life of its own and consumes more and more time that could be otherwise spent actually developing software.

A decade ago, one of us wrote up a 110-page process for a large development team. No matter how hard he tried to defend every word of his process as something of great value, the team members looked at the four-page summary in the back and ignored the rest of what he thought was valuable content. He learned from that experience: No matter how much process pride you might have as a leader, short one- to two-page process guides are what developers really want and need.

No amount of process over-specification will make up for bad people. Far better: Staff your project with good people, do whatever it takes to keep them happy, and use simple, well-bounded processes to guide them along the way.

A well-defined and (relatively speaking) lightweight process can help your team members work together to achieve remarkable and noteworthy results. This is significant and worthy of additional consideration.

In this light then, let's take a look at the top reasons for developing and using a process:

1. Move to larger projects and repeatable success.
2. Bring new staff in with a shorter ramp-up time.
3. Focus on high-payoff results.

6.4.1.1 Move to larger projects and repeatable success.

To move to larger projects and repeatable success, you need a good process, a system for building systems.

Simple, well-defined processes work best. Team members apply them several times, make refinements, and commit the process to memory. It becomes second nature to them. It becomes a good habit.

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 the process steps are logical and their worth immediately obvious to each team member.

With complex processes, about all you can hope for is "process pride," since learning and applying the process can keep you away from getting the real work accomplished.

With good habits in using simple, well-defined processes, the process itself moves from foreground to background. Team members focus on results rather than process micro-steps. Progress accelerates. The team reaches a new stride. The team performs!

6.4.1.2 Bring new staff in with a shorter ramp-up time.

Well bounded, simple processes allow the easy introduction of new staff: it dramatically shortens their learning curves and reduces the time it takes to become effective and efficient. When there is a practiced and simple system in place, it takes far less time for someone new to understand how things are done and to become effective. Standardization benefits also come into play here if processes are subject to them (standard language, process templates, naming conventions, where to find things, and the like).

It is far more effective to be able to spend a little time on process training and a lot of time on problem-domain training. The ramp-up to being productive will be shorter and much more efficient.

6.4.1.3 Focus on high-payoff results.

We've seen far too many technologists going beyond what is needed, and in extreme cases striving for (unattainable) perfection on one part of a project, without considering the other parts they compromise by doing so.

It's absolutely essential that your team focuses and stays focused on producing high-payoff results. Here are some suggestions for doing just that.

Help the team come to grips with this proverb:

> Every time you choose to do, you choose to leave something else undone. Choose wisely.
>
> Peter Coad Sr.

That means (in this context) setting and keeping priorities, building the must-have features, getting to "good enough," and not going beyond till other features get their due.

Make weekly progress reports visible to everyone on the team. And make individual progress visible at each desk. Here's how: Use your own form of "features completed on time" scorecards. Some organizations use colorful stickers for this, indicating "feature kills" (features completed on time) and "feature misses" (features that are late). The politically correct prefer "feature wins" rather than "feature kills."

6.4.2 Who Selects Tools for a Process?

Across-the-team-uniformity of tools in dealing with the various process artifacts streamlines what you do. So project tool selection is another important area to have well bounded.

Yet who selects tools? And who builds them?

We find that it's a good idea to designate a Tools Board, one or more people with the charter of defining tools to support the process, selecting most tools from vendors, and building smaller in-house tools as needed.

Use the Tools Board to drive all tooling decisions. And use its existence to thwart side-tracks by your best and brightest (who might occasionally fall in love with a custom tool and spend valuable time designing and building that tool, rather than designing and building client-valued project results).

But beware: Tools for the sake of tools is just as bad as process for the sake of process. Tools support the process. The Tool Board should strive to ensure that the tools work well together in a team environment. If a tool gets in the way, get rid of it. Tools are a means to an end.

6.4.3 How Might One Describe a Process?

The best processes we've applied were expressed in one or two pages. Surprised? It takes extra effort to write a process with simplicity, clarity, and brevity. As Pascal once put it:

> I have made this letter longer than usual, because I lack the time to make it short.[1]
>
> BLAISE PASCAL

[1] "Je n'ai fait cette lettre plus longue que parce que je n'ai pas eu le loisir de la faire plus courte." Blaise Pascal, *Lettres Provinciales* (1656–1657), no. 4.

The best pattern we've found for writing process templates is called ETVX: Entry, Task, Verification, and eXit:

1. Specify clear and well defined entry criteria for the process (can't start without these precursors).
2. Then list the tasks for that process with each task having a title, the project roles that participate in that task, whether that task is optional or required, and a task description (what am I to be doing?).
3. Next, specify the means of verification for the process (when have I accomplished "good enough" functionality?).
4. Finally, specify the exit criteria for the process, that is, how you know when you are complete and what the outputs (work products) are.

Clearly defined process tasks allow you to progress more efficiently. Without them, each developer makes his own way and ends up working harder than necessary to get the desired results.

Exit criteria must define tangible outputs. Define what the produced work products are, what the format is, and where the results go.

6.5 THE FIVE PROCESSES WITHIN FDD

This section presents the five processes within FDD (Figure 6-2):

- Process #1: Develop an overall model (using initial requirements/ features, snap together with components, focusing on shape).
- Process #2: Build a detailed, prioritized features list.
- Process #3: Plan by feature.
- Process #4: Design by feature (using components, focusing on sequences).
- Process #5: Build by feature.

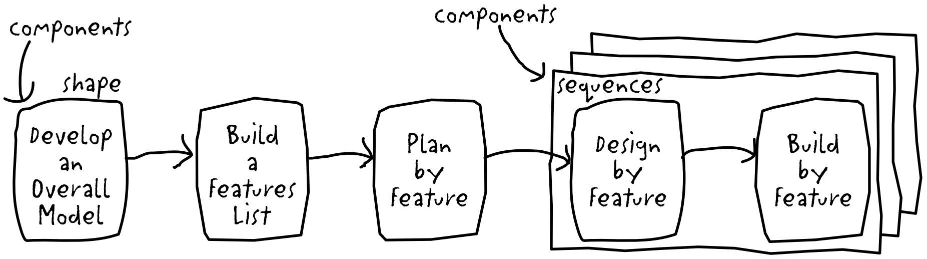


FIGURE 6-2. ▲ The five processes within FDD.

FDD Process #1: Develop an Overall Model

Domain and development members, under the guidance of an experienced component/object modeler (chief architect), work together in this process. Domain members present an initial high-level, highlights-only walk-through of the scope of the system and its context. The domain and development members produce a skeletal model, the very beginnings of that which is to follow. Then the domain members present more detailed walkthroughs. Each time, the domain and development members work in small sub-teams (with guidance from the chief architect); present sub-team results; merge the results into a common model (again with guidance from the chief architect), adjusting model shape along the way.

In subsequent iterations of this process, smaller teams tackle specialized domain topics. Domain members participate in many yet not all of those follow-up sessions.

Entry Criteria

The client is ready to proceed with the building of a system. He might have a list of requirements in some form. Yet he is not likely to have come to grips with what he really needs and what things are truly “must have” vs. “nice to have.” And that’s okay.

Tasks

Form the Modeling Team	Project Management	Required

The modeling team consists of permanent members from both domain and development areas. Rotate other project staff through the modeling sessions so that everyone gets a chance to observe and participate.

Domain Walkthrough	Modeling Team	Required

A domain member gives a short tutorial on the area to be modeled (from 20 minutes to several hours, depending upon the topic). The tutorial includes domain content that is relevant to the topic yet a bit broader than the likely system scope.

Study Documents	Modeling Team	Optional

The team scours available documents, including (if present): component models, functional requirements (traditional or use-case format), data models, and user guides.

Build an Informal Features List	Chief Architect, Chief Programmers	Required

The team builds an informal features list, early work leading up to FDD Process #2. The team notes specific references (document and page number) from available documents, as needed.

Develop Sub-team Models	Modeling Team in Small Groups	Required

The Chief Architect may propose a component or suggest a starting point. Using archetypes (in color) and components, each sub-team builds a class diagram for the domain under consideration, focusing on classes and links, then methods, and finally attributes. The sub-teams add methods from domain understanding, the initial features list, and methods suggested by the archetypes. The sub-teams sketch one or more informal sequence diagrams, too.

Develop a Team Model	Chief Architect, Modeling Team	Required

Each sub-team presents its proposed model for the domain area. The chief architect may also propose an additional alternative. The modeling team selects one of the proposed models as a baseline, merges in content from the other models, and keeps an informal sequence diagram. The team updates its overall model. The team annotates the model with notes, clarifying terminology and explaining key model-shape issues.

Log Alternatives	Chief Architect, Chief Programmers	Required

A team scribe (a role assigned on a rotating basis) logs notes on significant modeling alternatives that the team evaluated, for future reference on the project.

Verification

Internal and External Assessment	Modeling Team	Required

Domain members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, to clarify domain understanding, functionality needs, and scope.

Exit Criteria

To exit this process, the team must deliver the following results, subject to review and approval by the development manager and the chief architect:

- Class diagrams with (in order of descending importance) classes, links, methods, and attributes. Classes and links establish model shape. Methods (along with the initial features list and informal sequence diagrams) express functionality and are the raw materials for building a features list. Plus informal sequence diagrams.
- Informal features list
- Notes on significant modeling alternatives

FDD Process #2: Build a Features List

The team identifies the features, groups them hierarchically, prioritizes them, and weights them.

In subsequent iterations of this process, smaller teams tackle specialized feature areas. Domain members participate in many yet not all of those follow-up sessions.

Entry Criteria

The modeling team has successfully completed FDD Process #1, Develop an Overall Model.

Tasks

Form the Features-List Team	Project Manager, Development Manager	Required

The features-list team consists of permanent members from the domain and development areas.

Identify Features, Form Feature Sets	Features-List Team	Required

The team begins with the informal features list from FDD Process #1. It then:

- transforms methods in the model into features,
- transforms moment-intervals in the model into feature sets (and groupings of moment-intervals into major feature sets),
- (and mainly it) Brainstorms, selects, and adds features that will better satisfy client wants and needs.

It uses these formats:

- For features: <action> the <result> <by|for|of|to> a(n) <object>
- For feature sets: <action><-ing> a(n) <object>
- For major feature sets: <object> management

where an object is a person, place, or thing (including roles, moments in time or intervals of time, or catalog-entry-like descriptions)

Prioritize the Feature Sets and Features	Features-List Team	Required

A subset of the team, the Features Board establishes priorities for feature sets and features. Priorities are A (must have), B (nice to have), C (add it if we can), or D (future). In setting priorities, the team considers each feature in terms of client satisfaction (if we include the feature) and client dissatisfaction (if we don't).

Divide Complex Features	Features-List Team	Required

The development members, led by the chief architect, look for features that are likely to take more than two weeks to complete. The team divides those features into smaller features (steps).

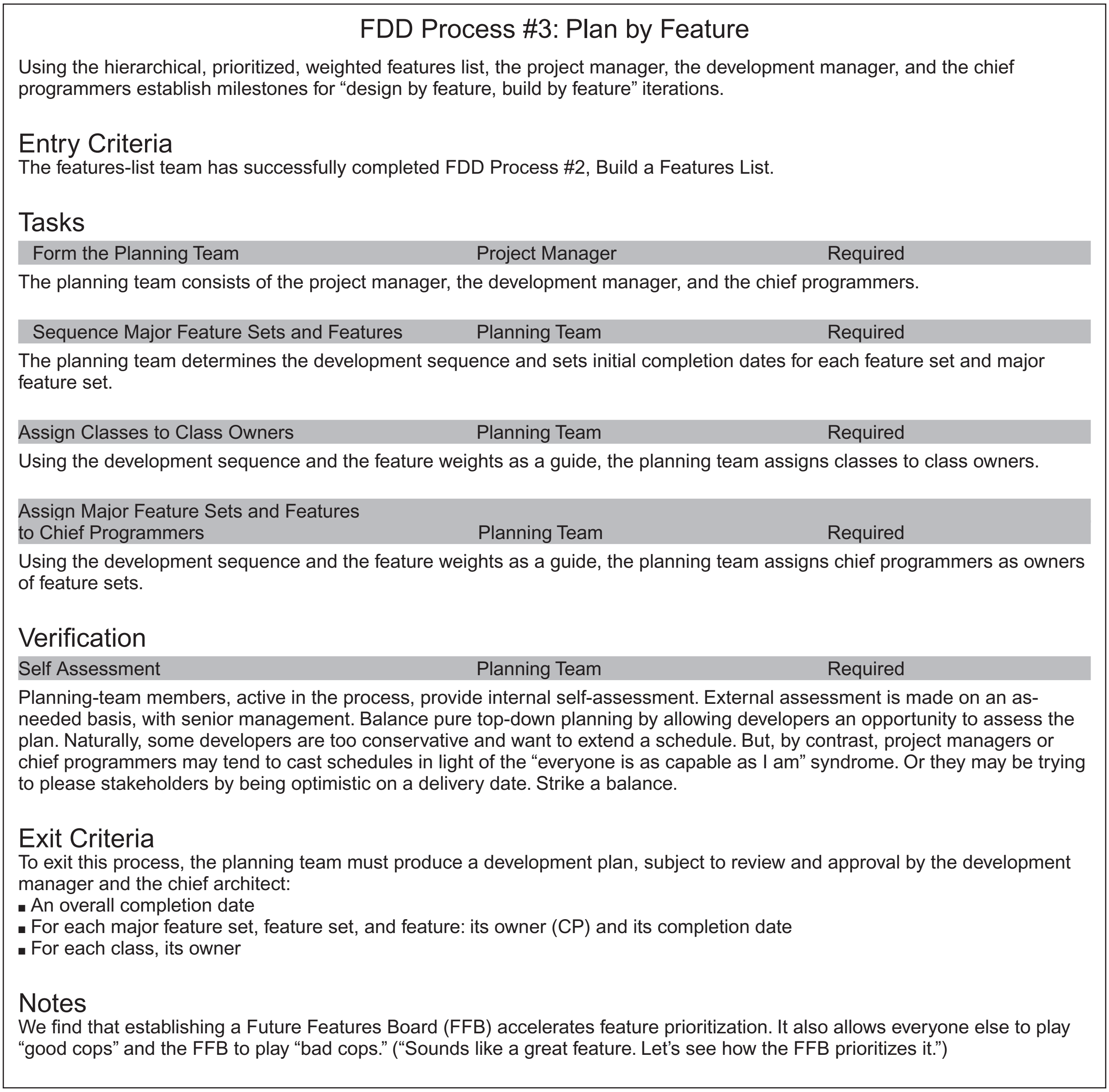
Verification

Internal and External Assessment	Features-List Team	Required

Domain members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, to clarify domain understanding, functionality needs, and scope.

Exit Criteria

To exit this process, the features-list team must deliver a detailed features list, grouped into major feature sets and feature sets, subject to review and approval by the development manager and the chief architect.



FDD Process #3: Plan by Feature

Using the hierarchical, prioritized, weighted features list, the project manager, the development manager, and the chief programmers establish milestones for “design by feature, build by feature” iterations.

Entry Criteria

The features-list team has successfully completed FDD Process #2, Build a Features List.

Tasks

Form the Planning Team	Project Manager	Required

The planning team consists of the project manager, the development manager, and the chief programmers.

Sequence Major Feature Sets and Features	Planning Team	Required

The planning team determines the development sequence and sets initial completion dates for each feature set and major feature set.

Assign Classes to Class Owners	Planning Team	Required

Using the development sequence and the feature weights as a guide, the planning team assigns classes to class owners.

Assign Major Feature Sets and Features to Chief Programmers	Planning Team	Required

Using the development sequence and the feature weights as a guide, the planning team assigns chief programmers as owners of feature sets.

Verification

Self Assessment	Planning Team	Required

Planning-team members, active in the process, provide internal self-assessment. External assessment is made on an as-needed basis, with senior management. Balance pure top-down planning by allowing developers an opportunity to assess the plan. Naturally, some developers are too conservative and want to extend a schedule. But, by contrast, project managers or chief programmers may tend to cast schedules in light of the “everyone is as capable as I am” syndrome. Or they may be trying to please stakeholders by being optimistic on a delivery date. Strike a balance.

Exit Criteria

To exit this process, the planning team must produce a development plan, subject to review and approval by the development manager and the chief architect:

- An overall completion date
- For each major feature set, feature set, and feature: its owner (CP) and its completion date
- For each class, its owner

Notes

We find that establishing a Future Features Board (FFB) accelerates feature prioritization. It also allows everyone else to play “good cops” and the FFB to play “bad cops.” (“Sounds like a great feature. Let’s see how the FFB prioritizes it.”)

FDD Process #4: Design by Feature (DBF)

A chief programmer takes the next feature, identifies the classes likely to be involved, and contacts the corresponding class owners. This feature team works out a detailed sequence diagram. The class owners write class and method prologs. The team conducts a design inspection.

Entry Criteria

The planning team has successfully completed FDD Process #3, Plan by Feature.

Tasks

Form a DBF Team	Chief Programmer	Required

The chief programmer identifies the classes likely to be involved in the design of this feature. From the class ownership list, the chief programmer identifies the developers needed to form the feature team. He contacts those class owners, initiating the design of this feature. He contacts a domain member too, if he needs one to help design this feature.

Domain Walkthrough	Feature Team, Domain	Optional

(This task is optional, depending upon feature complexity.) The domain member gives an overview of the domain area for the feature under consideration. He includes domain information that is related to the feature but not necessarily a part of its implementation to help set context.

Study the Referenced Documents	Feature Team	Optional

(This task is optional, depending upon feature complexity.) Using referenced documents from the features list and any other pertinent documents they can get their hands on, the feature team studies the documents, extracting detailed supporting information about and for the feature.

Build a Sequence Diagram	Feature Team	Required

Applying their understanding of the feature, plus components and informal sequence diagrams, the feature team builds a formal, detailed sequence diagram for the feature. The team logs design alternatives, decisions, assumptions, and notes. The chief programmer adds the sequence diagram (and corresponding class-diagram updates, as is nearly always the case) to the project model.

Write Class and Method Prologs	Feature Team	Required

Each class owner updates his class and method prologs for his methods in the sequence diagram. He includes parameter types, return types, exceptions, and message sends.

Design Inspection	Feature Team	Required

The feature team conducts a design inspection. The chief programmer invites several people from outside the team to participate, when he feels the complexity of the feature warrants it.

Log Design-Inspection Action Items	Scribe	Required

A team scribe logs design-inspection action items for each class owner, for follow-up by that class owner.

Verification

Design Inspection	Feature Team	Required

The feature team walks through its sequence diagram(s) to provide an internal self-assessment. External assessment is made on an as-needed basis, to clarify functionality needs and scope.

Exit Criteria

To exit this process, the feature team must deliver the following results, subject to review and approval by the chief programmer (with oversight from the chief architect):

- The feature and its referenced documents (if any)
- The detailed sequence diagram
- Class-diagram updates
- Class and method prolog updates
- Notes on the team's consideration of significant design alternatives

FDD Process #5: Build By Feature (BBF)

Starting with a DBF package, each class owner builds his methods for the feature. He extends his class-based test cases and performs class-level (unit) testing. The feature team inspects the code, perhaps before unit test, as determined by the chief programmer. Once the code is successfully implemented and inspected, the class owner checks in his class(es) to the configuration management system. When all classes for this feature are checked in, the chief programmer promotes the code to the build process.

Entry Criteria

The feature team has successfully completed FDD Process #4, Design by Feature, for the features to be built during this DBF/BBF iteration.

Tasks

Implement Classes and Methods	Feature Team	Required

Each class owner implements the methods in support of this feature as specified in the detailed sequence diagram developed during DBF. He also adds test methods. The chief programmer adds end-to-end feature test methods.

Code Inspection	Feature Team	Required

The chief programmer schedules a BBF code inspection. (He might choose to do this before unit testing or after unit testing.) The feature team conducts a code inspection (with outside participants when the chief programmer sees the need for such participation).

Log Code-Inspection Action Items	Scribe	Required

A team scribe logs code-inspection action items for each class owner, for follow-up by that class owner.

Unit Test	Feature Team	Required

Each class owner tests his code and its support of the feature. The chief programmer, acting as the integration point for the entire feature, conducts end-to-end feature testing.

Check in and Promote to the Build Process	Feature Team	Required

Once the code is successfully implemented, inspected and tested, each class owner checks in his classes to the configuration management system. When all classes for the feature are checked in and shown to be working end-to-end, the chief programmer promotes the classes to the build process. The chief programmer updates the feature's status in the features list.

Verification

Code Inspection and Unit Test	Feature Team	Required

The features team conducts a code inspection. A team scribe logs action items for each class owner.

Exit Criteria

To exit this process, the feature team must deliver the following results, subject to review and approval by its chief programmer:

- Implemented and inspected methods and test methods
- Unit test results, for each method and for the overall sequence
- Classes checked in by owners, features promoted to the build process and updated by the chief programmer

6.6 CHIEF PROGRAMMERS, CLASS OWNERS, AND FEATURE TEAMS

In FDD, two roles are essential elements: chief programmers and class owners. And one sociological structure is key: feature teams. Let's take a closer look at these three.

6.6.1 Chief Programmer

Feature-driven development requires someone to lead the DBF/BBF processes, feature by feature, leading by example (as a designer and programmer) and by mentoring (especially by way of inspections).

The number of chief programmers limits how fast and how far you can go with your project. If you want to increase project speed, recruit another chief programmer. A chief programmer in this context is someone who is significantly more productive than others on your team. The amplifying factor comes from a combination of raw talent, skills, training, and experience. Occasionally all those talents come together within one human being.

Adding more programmers tends to slow down a project, as Fred Brooks observed decades ago. We find this to be true with one exception: with small, client-valued features and lightweight processes, when you add a chief programmer then you can add people around him and actually accelerate a project by increasing the amount of in-parallel development you can tackle—but again, only to a point.

6.6.2 Class Owner

A class owner is someone responsible for the design and implementation of a class. We find this works very effectively. First, developers gain a sense of ownership of some part of the code, and we find pride of ownership a good and motivating force. Second, it brings local consistency to a class (just one programmer touches the code).

The norm is one class, one class owner. Occasionally, for a class with algorithmically complex methods, you might need one class, one class owner, and one algorithm programmer.

Yet FDD organizes activities by feature, not by class. As it should. After all, FDD is all about producing frequent, tangible, working results—small, client-value features! *Clients use features*. They do not use the organizational framework that developers use to implement little pieces of a feature.

6.6.3 Feature Teams

We assign features to a chief programmer. He takes each feature and identifies the likely class owners who will be involved in delivering that feature. Then he forms a temporary, "lasts just a week or two" team, called a feature team (Figure 6-3).

Class owners work on more than one feature team at a time. Feature-team membership may change with each DBF/BBF iteration.

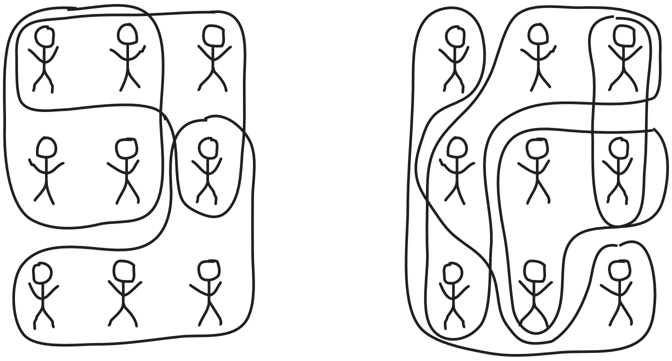


FIGURE 6-3. ▲ Feature-team membership may change with each DBF/BBF iteration.

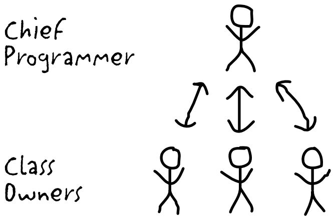


FIGURE 6-4. ▲ Interactions within a feature team.

The chief programmer is just that, the chief! The interactions within the team are primarily between the chief programmer and the other team members (Figure 6-4). Why? We encourage this approach to accelerate progress, ensure on-going mentoring of the team members by the chief programmer, and promote uniformity of design and implementation.

Overall, the chief architect mentors the chief programmers, who in turn mentor the class owners within a feature team.

6.7 TRACKING PROGRESS WITH PRECISION

How much time do teams spend within each of the five processes of FDD? Here are some useful guidelines (Figure 6-5):

Develop an overall model.	10% initial, 4% ongoing
Build a features list.	4% initial, 1% ongoing
Plan by feature.	2% initial, 2% ongoing
Design by feature, build by feature.	77% (cycle time: every 2 weeks)

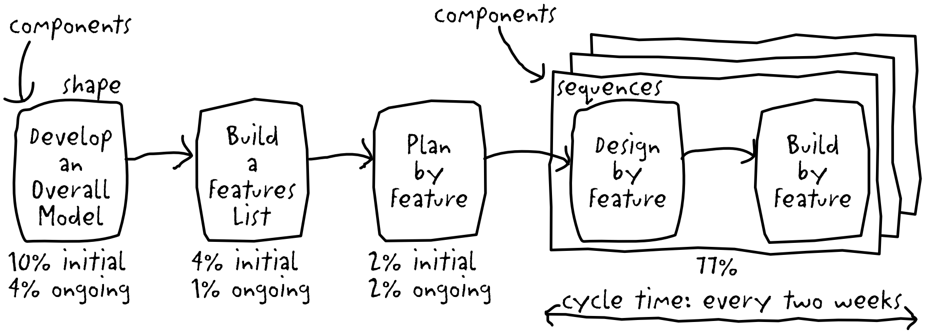


FIGURE 6-5. ▲ FDD processes with schedule percentages.

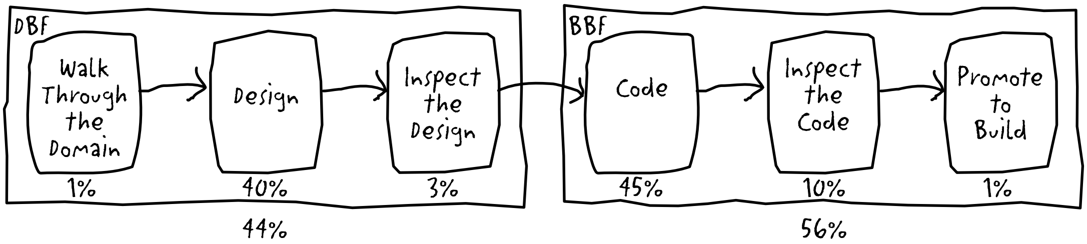


FIGURE 6-6. ▲ DBF/BBF milestone with schedule percentages.

Again, the percentages are useful guidelines (not absolutes).

The initial "develop an overall model, build a features list, and plan by feature" sequence consumes 16% of project schedule. The ongoing iterations of those front-end activities grab another 7%.

It's the other 77% we're concerned about in this section, the time spent in the many "design by feature, build by feature" iterations.

DBF/BBF consists of six little processes and corresponding schedule-percentage guidelines (Figure 6-6):

▪ DBF	
Walk through the domain.	1%
Design.	40%
Inspect the design.	3%
▪ BBF	
Code/test.	45%
Inspect the code.	10%
Promote to build.	1%

Note that the 45% for coding includes building unit-test methods and conducting units tests.

When applying DBF/BBF, do teams really spend less time designing (40% of DBF/BBF) than coding (45% of DBF/BBF)? Yes. Yet if we consider all of FDD and include initial object modeling when doing the comparison, we gain a bit more perspective on what is really happening here: Teams spend more time modeling and designing (45% of FDD) than coding (35% of FDD). The adage is still true: Succeed to plan, plan to succeed.

We plan for and track each DBF/BBF milestone. Remember that the total time from beginning to end is two weeks or less. So these milestones are very tiny—maybe “inch-pebbles.”

The combination of small client-valued features and these six DBF/BBF milestones is the secret behind FDD’s remarkable ability to track progress with precision.

Here’s an example: For a given feature, once you’ve walked through the domain and designed the feature, you count that feature as 41% complete.

6.7.1 Reporting

The release manager meets weekly with the chief programmers. In this 30-minutes-or-less meeting, each chief programmer verbally walks through the status of his features, marking up the project-tracking chart as he goes. Doing this together, verbally, is a good way to make sure the chief programmers take time to listen to each other and are aware of where the others are at in the development process. At the end of the meeting, the release manager takes those results, updates the database, and generates reports.

The release manager issues progress reports weekly, for the team (Figure 6-7) and for clients and senior management (Figure 6-8).

For upper management and client reporting, we report the percentage complete for each major feature set and feature set on a monthly basis. In fact, we like to report progress visually. We draw rectangles for each major feature set, and then inside each rectangle we draw rectangles for each feature set. Then inside the inner rectangles, we show the feature-set name, a progress bar showing percent complete, and the planned completion month. See Figure 6-9.

Note that the symbol is in three sections. Each section has its own color-coding scheme. The upper section indicates overall status: work in progress (yellow), attention (red), completed (green), and not yet started (white). The middle section shows percent complete: percent complete (green). The lower section illustrates completion status: the targeted completion month, or completed (green). When a feature set is fully complete, the entire box turns green.

Figure 6-10 shows what this would look like in a project-wide view.

<Major Feature-Set Name>.<Feature-Set Name> (<# of features>)

Id	Description	Chief Programmer	Class Owners	Walk-through		Design		Design Inspection		Development		Code Inspection		Promote to Build	
				Planned	Actual	Planned	Actual	Planned	Actual	Planned	Actual	Planned	Actual	Planned	Actual
Completion percentage for this feature set: __%															
Expected completion month for this feature set: <month> <year>.															

FIGURE 6-7. ▲ Feature tracking during DBF/BBF.

<Major Feature-Set Name> (<# of features>)

Feature Set (<# of features>)	Total Features	Not Started	In Progress	Behind Schedule	Completed	Inactive	% Completed	Completion Date

FIGURE 6-8. ▲ Major feature set and feature set tracking during DBF/BBF (includes project-wide totals, too)

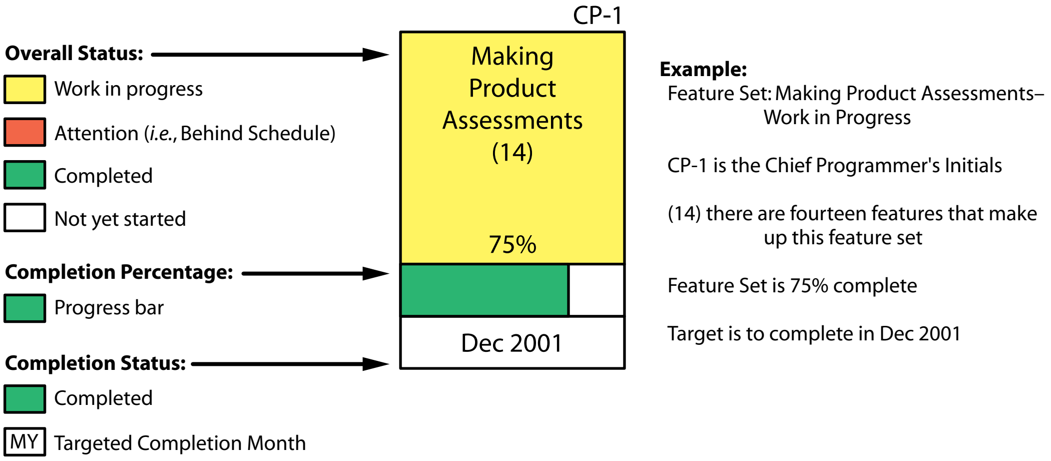

Overall Status:
Work in progress
Attention (*i.e.*, Behind Schedule)
Completed
Not yet started
Completion Percentage:
Progress bar
Completion Status:
Completed
MY Targeted Completion Month
CP-1
Making Product Assessments (14)
75%
Dec 2001
Example:
Feature Set: Making Product Assessments– Work in Progress
CP-1 is the Chief Programmer's Initials
(14) there are fourteen features that make up this feature set
Feature Set is 75% complete
Target is to complete in Dec 2001

FIGURE 6-9. ▲ Reporting progress to upper management and clients.

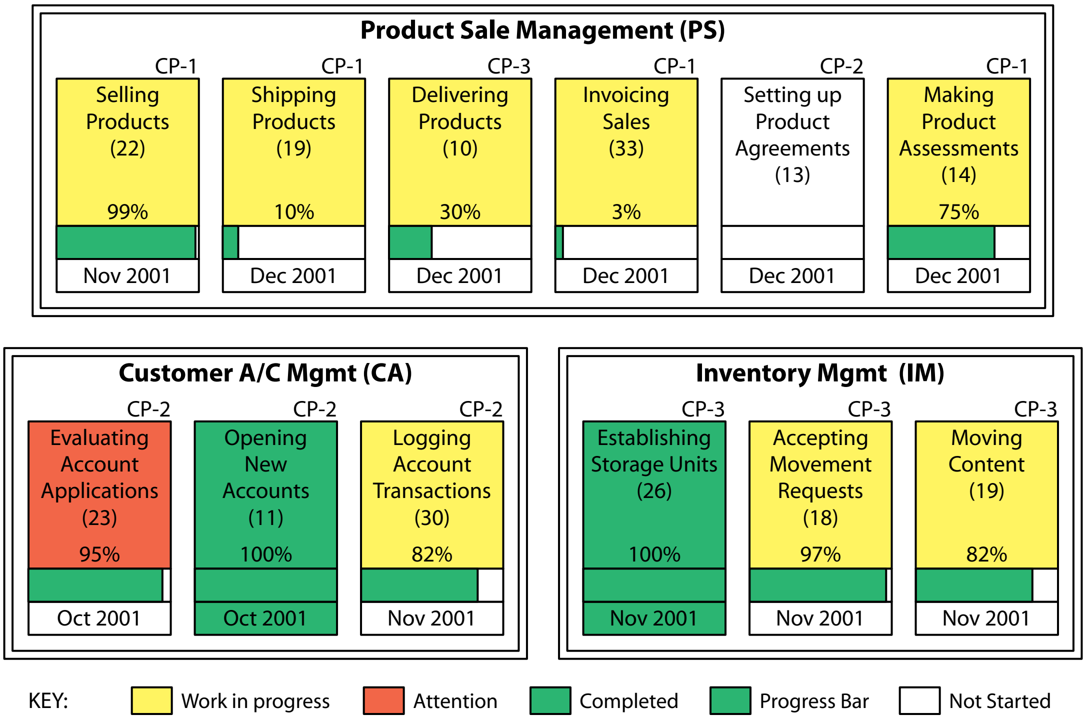


FIGURE 6-10. ▲ Reporting project-wide progress to upper management and clients.

6.7.2 Keeping a Features Database

Capture these items in your features database:

- Type (problem domain, human interaction, or system interaction)
- Identifier (feature-set prefix plus a sequence number)
- Status (on-hold, no longer required, normal)
- Major feature set
- Feature set
- Document references
- Action items
- Chief programmer
- Domain walk-through plan date, actual date
- Design plan date, actual date
- Design-inspection plan date, actual date
- Code plan date, actual date
- Code-inspection plan date, actual date
- Promote-to-build plan date, actual date
- Remarks

Track classes and owners in a separate table.
Automate reporting functions using your features database.

6.8 SUMMARY AND CONCLUSION

Feature-driven development is a process for helping teams produce frequent, tangible working results. It uses very small blocks of client-valued functionality, called features. FDD organizes those little blocks into business-related feature sets. FDD focuses developers on producing working results every two weeks. FDD includes planning strategies. And FDD tracks progress with precision.

We hope that you enjoy putting color archetypes, components, and feature-driven development to work on your projects. We wish you good success!

For ongoing news and updates, subscribe to The Coad Letter (a free series of special reports on better modeling and design, www.oi.com/publications.htm) and visit the Java Modeling home page (for additional components, updates, and more, www.oi.com/jm-book.htm).

Yours for better modeling and processes,

Peter Coad (pc@oi.com)
Eric Lefebvre (lefee@groupe-progestic.com)
Jeff De Luca (jdl@nebulon.com)

REFERENCES

[Brooks95] Brooks, Frederick P., Jr., *The Mythical Man Month: Essays on Software Engineering*. Anniversary Edition. Reading, MA: Addison Wesley, 1995.

[Davis98] Davis, Stan, and Christoper Meyer, *BLUR: The Speed of Change in the Connected Economy*. Reading, MA: Perseus Books, 1998.