**16: Analysis and Design**

The object-oriented paradigm is a new and different way of thinking about programming.

Many people have trouble at first knowing how to approach an OOP project. Now that you understand the concept of an object, and as you learn to think more in an object-oriented style, you can begin to create good designs that take advantage of all the benefits that OOP has to offer. This chapter introduces the ideas of analysis, design, and some ways to approach the problems of developing good object-oriented programs in a reasonable amount of time. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_251)

**Methodology**

A *methodology* (sometimes simply called a *method*)is a set of processes and heuristics used to break down the complexity of a programming problem. Many OOP methodologies have been formulated since the dawn of object-oriented programming. This section will give you a feel for what youre trying to accomplish when using a methodology. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_252)

Especially in OOP, methodology is a field of many experiments, so it is important to understand what problem the methodology is trying to solve before you consider adopting one. This is particularly true with Java, in which the programming language is intended to reduce the complexity (compared to C) involved in expressing a program. This may in fact alleviate the need for ever-more-complex methodologies. Instead, simple methodologies may suffice in Java for a much larger class of problems than you could handle using simple methodologies with procedural languages. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_253)

Its also important to realize that the term methodology is often too grand and promises too much. Whatever you do now when you design and write a program is a methodology. It may be your own methodology, and you may not be conscious of doing it, but it is a process you go through as you create. If it is an effective process, it may need only a small tune-up to work with Java. If you are not satisfied with your productivity and the way your programs turn out, you may want to consider adopting a formal methodology, or choosing pieces from among the many formal methodologies.[.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_254)

While youre going through the development process, the most important issue is this: Dont get lost. Its easy to do. Most of the analysis and design methodologies are intended to solve the largest of problems. Remember that most projects dont fit into that category, so you can usually have successful analysis and design with a relatively small subset of what a methodology recommends.[[102]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn102) But some sort of process, no matter how small or limited, will generally get you on your way in a much better fashion than simply beginning to code. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_255)

Its also easy to get stuck, to fall into analysis paralysis, where you feel like you cant move forward because you havent nailed down every little detail at the current stage. Remember, no matter how much analysis you do, there are some things about a system that wont reveal themselves until design time, and more things that wont reveal themselves until youre coding, or not even until a program is up and running. Because of this, its crucial to move fairly quickly through analysis and design, and to implement a test of the proposed system. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_256)

This point is worth emphasizing. Because of the history weve had with procedural languages, it is commendable that a team will want to proceed carefully and understand every minute detail before moving to design and implementation. Certainly, when creating a Database Management System (*DBMS*), it pays to understand a customers needs thoroughly. But a DBMS is in a class of problems that is very well-posed and well-understood; in many such programs, the database structure *is* the problem to be tackled. The class of programming problem discussed in this chapter is of the wild-card (my term) variety, in which the solution isnt simply re-forming a well-known solution, but instead involves one or more wild-card factorselements for which there is no well-understood previous solution, and for which research is necessary.[[103]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn103) Attempting to thoroughly analyze a wild-card problem before moving into design and implementation results in analysis paralysis because you dont have enough information to solve this kind of problem during the analysis phase. Solving such a problem requires iteration through the whole cycle, and that requires risk-taking behavior (which makes sense, because youre trying to do something new and the potential rewards are higher). It may seem like the risk is compounded by rushing into a preliminary implementation, but it can instead reduce the risk in a wild-card project because youre finding out early whether a particular approach to the problem is viable. Product development is risk management. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_257)

Its often proposed that you build one to throw away. With OOP, you may still throw *part* of it away, but because code is encapsulated into classes, during the first pass you will inevitably produce some useful class designs and develop some worthwhile ideas about the system design that do not need to be thrown away. Thus, the first rapid pass at a problem not only produces critical information for the next analysis, design, and implementation pass, it also creates a code foundation. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_258)

That said, if youre looking at a methodology that contains tremendous detail and suggests many steps and documents, its still difficult to know when to stop. Keep in mind what youre trying to discover: [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_259)

1. What are the objects? (How do you partition your project into its component parts?)
2. What are their interfaces? (What messages do you need to send to each object?)

If you come up with nothing more than the objects and their interfaces, then you can write a program. For various reasons you might need more descriptions and documents than this, but you cant get away with any less. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_260)

The process can be undertaken in five phases, and a Phase 0 that is just the initial commitment to using some kind of structure. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_261)

**Phase 0: Make a plan**

You must first decide what steps youre going to have in your process. It sounds simple (in fact, *all* of this sounds simple), and yet people often dont make this decision before they start coding. If your plan is lets jump in and start coding, fine. (Sometimes thats appropriate when you have a well-understood problem.) At least agree that this is the plan. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_262)

You might also decide at this phase that some additional process structure is necessary, but not the whole nine yards. Understandably, some programmers like to work in vacation mode, in which no structure is imposed on the process of developing their work; It will be done when its done. This can be appealing for a while, but Ive found that having a few milestones along the way helps to focus and galvanize your efforts around those milestones instead of being stuck with the single goal of finish the project. In addition, it divides the project into more bite-sized pieces and makes it seem less threatening (plus the milestones offer more opportunities for celebration). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_263)

When I began to study story structure (so that I will someday write a novel) I was initially resistant to the idea of structure, feeling that I wrote best when I simply let it flow onto the page. But I later realized that when I write about computers the structure is clear enough to me that I dont have to think about it very much. I still structure my work, albeit only semi-consciously in my head. Even if you think that your plan is to just start coding, you still somehow go through the subsequent phases while asking and answering certain questions. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_264)

**The mission statement**

Any system you build, no matter how complicated, has a fundamental purposethe business that its in, the basic need that it satisfies. If you can look past the user interface, the hardware- or system-specific details, the coding algorithms and the efficiency problems, you will eventually find the core of its beingsimple and straightforward. Like the so-called *high concept* from a Hollywood movie, you can describe it in one or two sentences. This pure description is the starting point. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_265)

The high concept is quite important because it sets the tone for your project; its a mission statement. You wont necessarily get it right the first time (you may be in a later phase of the project before it becomes completely clear), but keep trying until it feels right. For example, in an air-traffic control system you may start out with a high concept focused on the system that youre building: The tower program keeps track of the aircraft. But consider what happens when you shrink the system to a very small airfield; perhaps theres only a human controller, or none at all. A more useful model wont concern the solution youre creating as much as it describes the problem: Aircraft arrive, unload, service and reload, then depart. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_266)

**Phase 1: What are we making?**

In the previous generation of program design (called *procedural design*), this is called creating the *requirements analysis* and *system specification*. These, of course, were places to get lost; documents with intimidating names that could become big projects in their own right. Their intention was good, however. The requirements analysis says Make a list of the guidelines we will use to know when the job is done and the customer is satisfied.[[104]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn104) The system specification says Heres a description of *what* the program will do (not *how*) to satisfy the requirements. The requirements analysis is really a contract between you and the customer (even if the customer works within your company, or is some other object or system). The system specification is a top-level exploration into the problem and in some sense a discovery of whether it can be done and how long it will take. Since both of these will require consensus among people (and because they will usually change over time), I think its best to keep them as bare as possibleideally, to lists and basic diagramsto save time (this is in line with Extreme Programming, which advocates very minimal documentation, albeit for small- to medium-sized projects). You might have other constraints that require you to expand them into bigger documents, but by keeping the initial document small and concise, it can be created in a few sessions of group brainstorming with a leader who dynamically creates the description. This not only solicits input from everyone, it also fosters initial buy-in and agreement by everyone on the team. Perhaps most importantly, it can kick off a project with a lot of enthusiasm. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_267)

Its necessary to stay focused on the heart of what youre trying to accomplish in this phase: Determine what the system is supposed to do. The most valuable tool for this is a collection of what are called use cases, or in Extreme Programming, user stories. Use cases identify key features in the system that will reveal some of the fundamental classes youll be using. These are essentially descriptive answers to questions like:[[105]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn105) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_268)

* Who will use this system?
* What can those actors do with the system?
* How does *this* actor do *that* with this system?
* How else might this work if someone else were doing this, or if the same actor had a different objective? (to reveal variations)
* What problems might happen while doing this with the system? (to reveal exceptions)

If you are designing a bank auto-teller, for example, the use case for a particular aspect of the functionality of the system is able to describe what the auto-teller does in every possible situation. Each of these situations is referred to as a *scenario*, and a use case can be considered a collection of scenarios. You can think of a scenario as a question that starts with: What does the system do if...? For example, What does the auto-teller do if a customer has just deposited a check within the last 24 hours, and theres not enough in the account without the check having cleared to provide a desired withdrawal? [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_269)

Use case diagrams are intentionally simple to prevent you from getting bogged down in system implementation details prematurely:

Each stick person represents an actor, which is typically a human or some other kind of free agent. (These can even be other computer systems, as is the case with ATM.) The box represents the boundary of your system. The ellipses represent the use cases, which are descriptions of valuable work that can be performed with the system. The lines between the actors and the use cases represent the interactions. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_270)

It doesnt matter how the system is actually implemented, as long as it looks like this to the user. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_271)

A use case does not need to be terribly complex, even if the underlying system is complex. It is only intended to show the system as it appears to the user. For example: [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_272)

The use cases produce the requirements specifications by determining all the interactions that the user may have with the system. You try to discover a full set of use cases for your system, and once youve done that you have the core of what the system is supposed to do. The nice thing about focusing on use cases is that they always bring you back to the essentials and keep you from drifting off into issues that arent critical for getting the job done. That is, if you have a full set of use cases, you can describe your system and move on to the next phase. You probably wont get it all figured out perfectly on the first try, but thats OK. Everything will reveal itself in time, and if you demand a perfect system specification at this point youll get stuck. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_273)

If you do get stuck, you can kick-start this phase by using a rough approximation tool: Describe the system in a few paragraphs and then look for nouns and verbs. The nouns can suggest actors, context of the use case (e.g., lobby), or artifacts manipulated in the use case. Verbs can suggest interactions between actors and use cases, and specify steps within the use case. Youll also discover that nouns and verbs produce objects and messages during the design phase (and note that use cases describe interactions between subsystems, so the noun and verb technique can be used only as a brainstorming tool because it does not generate use cases).[[106]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn106) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_274)

The boundary between a use case and an actor can point out the existence of a user interface, but it does not define such a user interface. For a process of defining and creating user interfaces, see *Software for Use* by Larry Constantine and Lucy Lockwood, (Addison-Wesley Longman, 1999) or go to *www.ForUse.com*. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_275)

Although its a black art, at this point some kind of basic scheduling is important. You now have an overview of what youre building, so youll probably be able to get some idea of how long it will take. A lot of factors come into play here. If you estimate a long schedule then the company might decide not to build it (and thus use their resources on something more reasonablethats a *good* thing). Or a manager might have already decided how long the project should take and will try to influence your estimate. But its best to have an honest schedule from the beginning and deal with the tough decisions early. There have been a lot of attempts to come up with accurate scheduling techniques (much like techniques to predict the stock market), but probably the best approach is to rely on your experience and intuition. Get a gut feeling for how long it will really take, then double that and add 10 percent. Your gut feeling is probably correct; you *can* get something working in that time. The doubling will turn that into something decent, and the 10 percent will deal with the final polishing and details.[[107]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn107) However you want to explain it, and regardless of the moans and manipulations that happen when you reveal such a schedule, it just seems to work out that way.[[108]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn108) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_276)

**Phase 2: How will we build it?**

In this phase you must come up with a design that describes what the classes look like and how they will interact. An excellent technique in determining classes and interactions is the *Class-Responsibility-Collaboration* (CRC) card. Part of the value of this tool is that its so low-tech: You start out with a set of blank 3 x 5 cards, and you write on them. Each card represents a single class, and on the card you write: [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_277)

1. The name of the class. Its important that this name capture the essence of what the class does, so that it makes sense at a glance. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_278)
2. The responsibilities of the class: what it should do. This can typically be summarized by just stating the names of the methods (since those names should be descriptive in a good design), but it does not preclude other notes. If you need to seed the process, look at the problem from a lazy programmers standpoint: What objects would you like to magically appear to solve your problem? [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_279)
3. The collaborations of the class: What other classes does it interact with? Interact is an intentionally broad term; it could mean aggregation or simply that some other object exists that will perform services for an object of the class. Collaborations should also consider the audience for this class. For example, if you create a class **Firecracker**, who is going to observe it, a **Chemist** or a **Spectator**? The former will want to know what chemicals go into the construction, and the latter will respond to the colors and shapes released when it explodes. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_280)

You may feel that the cards should be bigger because of all the information youd like to get on them. However, they are intentionally small, not only to keep your classes small but also to keep you from getting into too much detail too early. If you cant fit all you need to know about a class on a small card, then the class is too complex (either youre getting too detailed, or you should create more than one class). The ideal class should be understood at a glance. The idea of CRC cards is to assist you in coming up with a first cut of the design so that you can get the big picture and then refine your design. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_281)

One of the great benefits of CRC cards is in communication. Its best done in real time, in a group, without computers. Each person takes responsibility for several classes (which at first have no names or other information). You run a live simulation by solving one scenario at a time, deciding which messages are sent to the various objects to satisfy each scenario. As you go through this process, you discover the classes that you need along with their responsibilities and collaborations, and you fill out the cards as you do this. When youve moved through all the use cases, you should have a fairly complete first cut of your design. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_282)

Before I began using CRC cards, the most successful consulting experiences I had when coming up with an initial design involved standing in front of a teamwho hadnt built an OOP project beforeand drawing objects on a whiteboard. We talked about how the objects should communicate with each other, and erased some of them and replaced them with other objects. Effectively, I was managing all the CRC cards on the whiteboard. The team (who knew what the project was supposed to do) actually created the design; they owned the design rather than having it given to them. All I was doing was guiding the process by asking the right questions, trying out the assumptions, and taking the . from the team to modify those assumptions. The true beauty of the process was that the team learned how to do object-oriented design not by reviewing abstract examples, but by working on the one design that was most interesting to them at that moment: theirs. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_283)

Once youve come up with a set of CRC cards, you may want to create a more formal description of your design using UML.[[109]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn109) You dont need to use UML, but it can be helpful, especially if you want to put up a diagram on the wall for everyone to ponder, which is a good idea (there is a plethora of UML diagramming tools available). An alternative to UML is a textual description of the objects and their interfaces, or, depending on your programming language, the code itself.[[110]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn110) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_284)

UML also provides an additional diagramming notation for describing the dynamic model of your system. This is helpful in situations in which the state transitions of a system or subsystem are dominant enough that they need their own diagrams (such as in a control system). You may also need to describe the data structures, for systems or subsystems in which data is a dominant factor (such as a database). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_285)

Youll know youre done with Phase 2 when you have described the objects and their interfaces. Well, most of themthere are usually a few that slip through the cracks and dont make themselves known until Phase 3. But thats OK. Whats important is that you eventually discover all of your objects. Its nice to discover them early in the process, but OOP provides enough structure so that its not so bad if you discover them later. In fact, the design of an object tends to happen in five stages, throughout the process of program development. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_286)

**Five stages of object design**

The design life of an object is not limited to the time when youre writing the program. Instead, the design of an object appears over a sequence of stages. Its helpful to have this perspective because you stop expecting perfection right away; instead, you realize that the understanding of what an object does and what it should look like happens over time. This view also applies to the design of various types of programs; the pattern for a particular type of program emerges through struggling again and again with that problem (which is chronicled in the book *Thinking in Patterns (with Java)* at*www.BruceEckel.com*). Objects, too, have their patterns that emerge through understanding, use, and reuse. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_287)

**1. Object discovery.**This stage occurs during the initial analysis of a program. Objects may be discovered by looking for external factors and boundaries, duplication of elements in the system, and the smallest conceptual units. Some objects are obvious if you already have a set of class libraries. Commonality between classes suggesting base classes and inheritance may appear right away, or later in the design process. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_288)

**2. Object assembly.**As youre building an object youll discover the need for new members that didnt appear during discovery. The internal needs of the object may require other classes to support it. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_289)

**3. System construction.**Once again, more requirements for an object may appear at this later stage. As you learn, you evolve your objects. The need for communication and interconnection with other objects in the system may change the needs of your classes or require new classes. For example, you may discover the need for facilitator or helper classes, such as a linked list, that contain little or no state information and simply help other classes function. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_290)

**4. System extension.**As you add new features to a system you may discover that your previous design doesnt support easy system extension. With this new information, you can restructure parts of the system, possibly adding new classes or class hierarchies. This is also a good time to consider taking features *out* of a project. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_291)

**5. Object reuse.**This is the real stress test for a class. If someone tries to reuse the class in an entirely new situation, theyll probably discover some shortcomings. As you change it to adapt to more new programs, the general principles of the class will become clearer, until you have a truly reusable type. However, dont expect most objects from a system design to be reusableit is perfectly acceptable for the bulk of your objects to be system-specific. Reusable types tend to be less common, and they must solve more general problems in order to be reusable. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_292)

**Guidelines for object development**

These stages suggest some guidelines when thinking about developing your classes: [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_293)

1. Let a specific problem generate a class, then let the class grow and mature during the solution of other problems. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_294)
2. Remember, discovering the classes you need (and their interfaces) is the majority of the system design. If you already had those classes, this would be an easy project. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_295)
3. Dont force yourself to know everything at the beginning. Learn as you go. This will happen anyway. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_296)
4. Start programming. Get something working so you can prove or disprove your design. Dont fear that youll end up with procedural-style spaghetti codeclasses partition the problem and help control anarchy and entropy. Bad classes do not break good classes. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_297)
5. Always keep it simple. Little clean objects with obvious utility are better than big complicated interfaces. When decision points come up, use an Ockhams Razor[[111]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn111) approach: Consider the choices and select the one that is simplest, because simple classes are almost always best. Start small and simple, and you can expand the class interface when you understand it better. Its easy to add methods, but as time goes on, its difficult to remove methods from a class. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_298)

**Phase 3: Build the core**

This is the initial conversion from the rough design into a compiling and executing body of code that can be tested, and especially that will prove or disprove your architecture. This is not a one-pass process, but rather the beginning of a series of steps that will iteratively build the system, as youll see in Phase 4. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_299)

Your goal is to find the core of your system architecture that needs to be implemented in order to generate a running system, no matter how incomplete that system is in this initial pass. Youre creating a framework that you can build on with further iterations. Youre also performing the first of many system integrations and tests, and giving the stakeholders . about what their system will look like and how it is progressing. Ideally, you are exposing some of the critical risks. Youll probably discover changes and improvements that can be made to your original architecturethings you would not have learned without implementing the system. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_300)

Part of building the system is the reality check that you get from testing against your requirements analysis and system specification (in whatever form they exist). Make sure that your tests verify the requirements and use cases. When the core of the system is stable, youre ready to move on and add more functionality. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_301)

**Phase 4: Iterate the use cases**

Once the core framework is running, each feature set you add is a small project in itself. You add a feature set during an *iteration*, a reasonably short period of development. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_302)

How big is an iteration? Ideally, each iteration lasts one to three weeks (this can vary based on the implementation language). At the end of that period, you have an integrated, tested system with more functionality than it had before. But whats particularly interesting is the basis for the iteration: a single use case. Each use case is a package of related functionality that you build into the system all at once, during one iteration. Not only does this give you a better idea of what the scope of a use case should be, but it also gives more validation to the idea of a use case, since the concept isnt discarded after analysis and design, but instead it is a fundamental unit of development throughout the software-building process. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_303)

You stop iterating when you achieve target functionality or an external deadline arrives and the customer can be satisfied with the current version. (Remember, software is a subscription business.) Because the process is iterative, you have many opportunities to ship a product rather than having a single endpoint; open-source projects work exclusively in an iterative, high-. environment, which is precisely what makes them successful. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_304)

An iterative development process is valuable for many reasons. You can reveal and resolve critical risks early, the customers have ample opportunity to change their minds, programmer satisfaction is higher, and the project can be steered with more precision. But an additional important benefit is the . to the stakeholders, who can see by the current state of the product exactly where everything lies. This may reduce or eliminate the need for mind-numbing status meetings and increase the confidence and support from the stakeholders. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_305)

**Phase 5: Evolution**

This is the point in the development cycle that has traditionally been called maintenance, a catch-all term that can mean everything from getting it to work the way it was really supposed to in the first place to adding features that the customer forgot to mention to the more traditional fixing the bugs that show up and adding new features as the need arises. So many misconceptions have been applied to the term maintenance that it has taken on a slightly deceiving quality, partly because it suggests that youve actually built a pristine program and all you need to do is change parts, oil it, and keep it from rusting. Perhaps theres a better term to describe whats going on. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_306)

Ill use the term *evolution.*[[112]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn112) That is, You wont get it right the first time, so give yourself the latitude to learn and to go back and make changes. You might need to make a lot of changes as you learn and understand the problem more deeply. The elegance youll produce if you evolve until you get it right will pay off, both in the short and the long term. Evolution is where your program goes from good to great, and where those issues that you didnt really understand in the first pass become clear. Its also where your classes can evolve from single-project usage to reusable resources. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_307)

What it means to get it right isnt just that the program works according to the requirements and the use cases. It also means that the internal structure of the code makes sense to you, and feels like it fits together well, with no awkward syntax, oversized objects, or ungainly exposed bits of code. In addition, you must have some sense that the program structure will survive the changes that it will inevitably go through during its lifetime, and that those changes can be made easily and cleanly. This is no small feat. You must not only understand what youre building, but also how the program will evolve (what I call the *vector of change*). Fortunately, object-oriented programming languages are particularly adept at supporting this kind of continuing modificationthe boundaries created by the objects are what tend to keep the structure from breaking down. They also allow you to make changesones that would seem drastic in a procedural programwithout causing earthquakes throughout your code. In fact, support for evolution might be the most important benefit of OOP. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_308)

With evolution, you create something that at least approximates what you think youre building, and then you kick the tires, compare it to your requirements, and see where it falls short. Then you can go back and fix it by redesigning and reimplementing the portions of the program that didnt work right.[[113]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn113) You might actually need to solve the problem, or an aspect of the problem, several times before you hit on the right solution. (A study of *Design Patterns*is usually helpful here. You can find information in *Thinking in Patterns (with Java)* at *www.BruceEckel.com*.) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_309)

Evolution also occurs when you build a system, see that it matches your requirements, and then discover it wasnt actually what you wanted. When you see the system in operation, you may find that you really wanted to solve a different problem. If you think this kind of evolution is going to happen, then you owe it to yourself to build your first version as quickly as possible so you can find out if it is indeed what you want. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_310)

Perhaps the most important thing to remember is that by defaultby definition, reallyif you modify a class, its super- and subclasses will still function. You need not fear modification (especially if you have a built-in set of unit tests to verify the correctness of your modifications). Modification wont necessarily break the program, and any change in the outcome will be limited to subclasses and/or specific collaborators of the class you change. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_311)

**Plans pay off**

Of course you wouldnt build a house without a lot of carefully drawn plans. If you build a deck or a dog house your plans wont be so elaborate, but youll probably still start with some kind of sketches to guide you on your way. Software development has gone to extremes. For a long time, people didnt have much structure in their development, but then big projects began failing. In reaction, we ended up with methodologies that had an intimidating amount of structure and detail, primarily intended for those big projects. These methodologies were too scary to useit looked like youd spend all your time writing documents and no time programming. (This was often the case.) I hope that what Ive shown you here suggests a middle patha sliding scale. Use an approach that fits your needs (and your personality). No matter how minimal you choose to make it, *some* kind of plan will make a big improvement in your project, as opposed to no plan at all. Remember that, by most estimates, over 50 percent of projects fail (some estimates go up to 70 percent!). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_312)

By following a planpreferably one that is simple and briefand coming up with design structure before coding, youll discover that things fall together far more easily than if you dive in and start hacking. Youll also realize a great deal of satisfaction. Its my experience that coming up with an elegant solution is deeply satisfying at an entirely different level; it feels closer to art than technology. And elegance always pays off; its not a frivolous pursuit. Not only does it give you a program thats easier to build and debug, but its also easier to understand and maintain, and thats where the financial value lies. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_313)

**Extreme Programming**

I have studied analysis and design techniques, on and off, since I was in graduate school. The concept of *Extreme Programming* (XP) is the most radical, and delightful, that Ive seen. You can find it chronicled in *Extreme Programming Explained* by Kent Beck (Addison-Wesley, 2000) and on the Web at *www.xprogramming.com*. Addison-Wesley also seems to come out with a new book in the XP series every month or two; the goal seems to be to convince everyone to convert using sheer weight of books (generally, however, these books are small and pleasant to read). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_314)

XP is both a philosophy about programming work and a set of guidelines to do it. Some of these guidelines are reflected in other recent methodologies, but the two most important and distinct contributions, in my opinion, are write tests first and pair programming. Although he argues strongly for the whole process, Beck points out that if you adopt only these two practices youll greatly improve your productivity and reliability. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_315)

**Write tests first**

Testing has traditionally been relegated to the last part of a project, after youve gotten everything working, but just to be sure. It has implicitly had a low priority, and people who specialize in it have not been given a lot of status and have often even been cordoned off in a basement, away from the real programmers. Test teams have responded in kind, going so far as to wear black clothing and cackling with glee whenever they break something (to be honest, Ive had this feeling myself when breaking compilers). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_316)

XP completely revolutionizes the concept of testing by giving it equal (or even greater) priority than the code. In fact, you write the tests *before* you write the code that will be tested, and the tests stay with the code forever. The tests must be executed successfully every time you do a build of the project (which is often, sometimes more than once a day). [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_317)

Writing tests first has two extremely important effects. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_318)

First, it forces a clear definition of the interface of a class. Ive often suggested that people imagine the perfect class to solve a particular problem as a tool when trying to design the system. The XP testing strategy goes further than thatit specifies exactly what the class must look like, to the consumer of that class, and exactly how the class must behave. In no uncertain terms. You can write all the prose, or create all the diagrams you want, describing how a class should behave and what it looks like, but nothing is as real as a set of tests. The former is a wish list, but the tests are a contract that is enforced by the compiler and the test framework. Its hard to imagine a more concrete description of a class than the tests. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_319)

While creating the tests, you are forced to completely think out the class and will often discover needed functionality that might be missed during the thought experiments of UML diagrams, CRC cards, use cases, etc. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_320)

The second important effect of writing the tests first comes from running the tests every time you do a build of your software. This activity gives you the other half of the testing thats performed by the compiler. If you look at the evolution of programming languages from this perspective, youll see that the real improvements in the technology have actually revolved around testing. Assembly language checked only for syntax, but C imposed some semantic restrictions, and these prevented you from making certain types of mistakes. OOP languages impose even more semantic restrictions, which if you think about it are actually forms of testing. Is this data type being used properly? and Is this method being called properly? are the kinds of tests that are being performed by the compiler or run-time system. Weve seen the results of having these tests built into the language: People have been able to write more complex systems, and get them to work, with much less time and effort. Ive puzzled over why this is, but now I realize its the tests: You do something wrong, and the safety net of the built-in tests tells you theres a problem and points you to where it is. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_321)

But the built-in testing afforded by the design of the language can only go so far. At some point, *you* must step in and add the rest of the tests that produce a full suite (in cooperation with the compiler and run-time system) that verifies all of your program. And, just like having a compiler watching over your shoulder, wouldnt you want these tests helping you right from the beginning? Thats why you write them first, and run them automatically with every build of your system. Your tests become an extension of the safety net provided by the language. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_322)

One of the things that Ive discovered about the use of more and more powerful programming languages is that I am emboldened to try more brazen experiments, because I know that the language will keep me from wasting my time chasing bugs. The XP test scheme does the same thing for your entire project. Because you know your tests will always catch any problems that you introduce (and you regularly add any new tests as you think of them), you can make big changes when you need to without worrying that youll throw the whole project into complete disarray. This is incredibly powerful. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_323)

In this third edition of this book, I realized that testing was so important that it must also be applied to the examples in the book itself. With the help of the Crested Butte Summer 2002 Interns, we developed the testing system that you will see used throughout this book. The code and description is in Chapter 15. This system has increased the robustness of the code examples in this book immeasurably. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]A0443)

**Pair programming**

Pair programming goes against the rugged individualism that weve been indoctrinated into from the beginning, through school (where we succeed or fail on our own, and working with our neighbors is considered cheating), and media, especially Hollywood movies in which the hero is usually fighting against mindless conformity.[[114]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn114) Programmers, too, are considered paragons of individualitycowboy coders, as Larry Constantine likes to say. And yet XP, which is itself battling against conventional thinking, says that code should be written with two people per workstation. And that this should be done in an area with a group of workstations, without the barriers that the facilities-design people are so fond of. In fact, Beck says that the first task of converting to XP is to arrive with screwdrivers and Allen wrenches and take apart everything that gets in the way.[[115]](http://iimk.ac.in/gsdl/cgi-bin/library?e=d-000-00---0inftec--00-0-0--0prompt-10---4------0-0l--1-en-50---20-about---00031-001-1-0utfZz-8-00&a=d&c=inftec&cl=CL2&d=HASH013a1053348ed13cf7a48e82.19" \l "fn115) (This will require a manager who can deflect the ire of the facilities department.) [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_324)

The value of pair programming is that one person is actually doing the coding while the other is thinking about it. The thinker keeps the big picture in mindnot only the picture of the problem at hand, but the guidelines of XP. If two people are working, its less likely that one of them will get away with saying, I dont want to write the tests first, for example. And if the coder gets stuck, they can swap places. If both of them get stuck, their musings may be overheard by someone else in the work area who can contribute. Working in pairs keeps things flowing and on track. Probably more important, it makes programming a lot more social and fun. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_325)

Ive begun using pair programming during the exercise periods in some of my seminars, and it seems to significantly improve everyones experience. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_326)

**Strategies for transition**

If you buy into OOP, your next question is probably, How can I get my manager/colleagues/department/peers to start using objects? Think about how youone independent programmerwould go about learning to use a new language and a new programming paradigm. Youve done it before. First comes education and examples; then comes a trial project to give you a feel for the basics without doing anything too confusing. Then comes a real world project that actually does something useful. Throughout your first projects you continue your education by reading, asking questions of experts, and trading hints with friends. This is the approach many experienced programmers suggest for the switch to Java. Switching an entire company will of course introduce certain group dynamics, but it will help at each step to remember how one person would do it. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_334)

**Guidelines**

Here are some guidelines to consider when making the transition to OOP and Java: [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_335)

**1. Training**

The first step is some form of education. Remember the companys investment in code, and try not to throw everything into disarray for six to nine months while everyone puzzles over unfamiliar features. Pick a small group for indoctrination, preferably one composed of people who are curious, work well together, and can function as their own support network while theyre learning Java. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]A0106)

An alternative approach is the education of all company levels at once, including overview courses for strategic managers as well as design and programming courses for project builders. This is especially good for smaller companies making fundamental shifts in the way they do things, or at the division level of larger companies. Because the cost is higher, however, some may choose to start with project-level training, do a pilot project (possibly with an outside mentor), and let the project team become the teachers for the rest of the company. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_336)

**2. Low-risk project**

Try a low-risk project first and allow for mistakes. Once youve gained some experience, you can either seed other projects from members of this first team or use the team members as an OOP technical support staff. This first project may not work right the first time, so it should not be mission-critical for the company. It should be simple, self-contained, and instructive; this means that it should involve creating classes that will be meaningful to the other programmers in the company when they get their turn to learn Java. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_337)

**3. Model from success**

Seek out examples of good object-oriented design before starting from scratch. Theres a good probability that someone has solved your problem already, and if they havent solved it exactly you can probably apply what youve learned about abstraction to modify an existing design to fit your needs. This is the general concept of *design patterns,*covered in *Thinking in Patterns (with Java)* at *www.BruceEckel.com*. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_338)

**4. Use existing class libraries**

An important economic motivation for switching to OOP is the easy use of existing code in the form of class libraries (in particular, the Standard Java libraries, which are covered throughout this book). The shortest application development cycle will result when you can create and use objects from off-the-shelf libraries. However, some new programmers dont understand this, are unaware of existing class libraries, or, through fascination with the language, desire to write classes that may already exist. Your success with OOP and Java will be optimized if you make an effort to seek out and reuse other peoples code early in the transition process. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_339)

**5. Dont rewrite existing code in Java**

It is not usually the best use of your time to take existing, functional code and rewrite it in Java. If you must turn it into objects, you can interface to the C or C++ code using the Java Native Interface or*Extensible Markup Language*(XML). There are incremental benefits, especially if the code is slated for reuse. But chances are you arent going to see the dramatic increases in productivity that you hope for in your first few projects unless that project is a new one. Java and OOP shine best when taking a project from concept to reality. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_340)

**Management obstacles**

If youre a manager, your job is to acquire resources for your team, to overcome barriers to your teams success, and in general to try to provide the most productive and enjoyable environment so your team is most likely to perform those miracles that are always being asked of you. Moving to Java falls in all three of these categories, and it would be wonderful if it didnt cost you anything as well. Although moving to Java may be cheaperdepending on your constraintsthan the OOP alternatives for a team of C programmers (and probably for programmers in other procedural languages), it isnt free, and there are obstacles you should be aware of before trying to sell the move to Java within your company and embarking on the move itself. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_341)

**Startup costs**

The cost of moving to Java is more than just the acquisition of Java compilers (the Sun Java compiler is free, so this is hardly an obstacle). Your medium- and long-term costs will be minimized if you invest in training (and possibly mentoring for your first project) and also if you identify and purchase class libraries that solve your problem rather than trying to build those libraries yourself. These are hard-money costs that must be factored into a realistic proposal. In addition, there are the hidden costs in loss of productivity while learning a new language and possibly a new programming environment. Training and mentoring can certainly minimize these, but team members must overcome their own struggles to understand the new technology. During this process they will make more mistakes (this is a feature, because acknowledged mistakes are the fastest path to learning) and be less productive. Even then, with some types of programming problems, the right classes, and the right development environment, its possible to be more productive while youre learning Java (even considering that youre making more mistakes and writing fewer lines of code per day) than if youd stayed with C. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_342)

**Performance issues**

A common question is, Doesnt OOP automatically make my programs a lot bigger and slower? The answer is, It depends. The extra safety features in Java have traditionally extracted a performance penalty over a language like C++. Technologies such as hotspot and compilation technologies have improved the speed significantly in most cases, and efforts continue toward higher performance. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_343)

When your focus is on rapid prototyping, you can throw together components as fast as possible while ignoring efficiency issues. If youre using any third-party libraries, these are usually already optimized by their vendors; in any case its not an issue while youre in rapid-development mode. When you have a system that you like, if its small and fast enough, then youre done. If not, you begin tuning with a profiler, looking first for speedups that can be done by rewriting small portions of code. If that doesnt help, you look for modifications that can be made in the underlying implementation so no code that uses a particular class needs to be changed. Only if nothing else solves the problem do you need to change the design. If performance is so critical in that portion of the design, it must be part of the primary design criteria. You have the benefit of finding this out early using rapid development.

Chapter 15 introduces *profilers*, which can help you discover bottlenecks in your system so you can optimize that portion of your code (with the hotspot technologies, Sun no longer recommends using native methods for performance optimization). Optimization tools are also available. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_344)

**Common design errors**

When starting your team into OOP and Java, programmers will typically go through a series of common design errors. This often happens due to insufficient . from experts during the design and implementation of early projects, because no experts have been developed within the company, and because there may be resistance to retaining consultants. Its easy to feel that you understand OOP too early in the cycle and go off on a bad tangent. Something thats obvious to someone experienced with the language may be a subject of great internal debate for a novice. Much of this trauma can be skipped by using an experienced outside expert for training and mentoring. [.](mailto:TIJ3@MindView.net?Subject=[TIJ3]Chap01_345)