00000010

CHAPTER 2

An Approach To The Art Of Programming

Learning Objectives

- *Describe the difficulties you will encounter in your quest to become a software developer*
- *List and describe the features of an integrated development environment (IDE)*
- *List and describe the stages of the "flow"*
- *List and describe the three roles you will play as a programming student*
- *State the purpose of the project-approach strategy*
- *List and describe the steps of the project approach strategy*
- *List and describe the steps of the software development cycle*
- *List and describe two types of project complexity*
- *State the meaning of the phrases "maximize cohesion" and "minimize coupling"*
- *Describe the differences between functional decomposition and object-oriented design*
- *State the meaning of the term "isomorphic mapping"*
- *Explain the difference between object-oriented programming and functional programming*

Introduction

Don’t be fooled by the term *scripting*. Scripting is programming and programming requires skill and creativity. A good programmer is an artist in every sense of the word. As with any art, you can learn the secrets of the craft. That’s what this chapter is all about.

Perhaps the most prevalent personality trait I’ve noticed in good programmers is a *knack for problem solving*. Problem solving requires creativity and lots of it. When you program a computer you are solving a problem with a machine. You transfer your knowledge of a particular problem into code, transform the code into a form understandable by a machine, and run the result on a machine. Doing this requires lots of creativity especially when you find yourself stumped by a particular problem.

Believe it or not, the hardest part about learning to program a computer, in any programming language, is not learning the language itself, it is learning how to approach the art of problem solving. To this end, the material in this chapter is aimed squarely at the beginner. However, I must issue a word of warning. If you are truly a novice, then some of what you read in this chapter will make less sense to you than to someone already familiar with programming concepts. Don’t worry, if you feel like skipping parts of this chapter now, go right ahead. The material will be here when you need it. In fact, you will grow to appreciate this chapter more as you gain experience as a software developer.

1 The Difficulties You Will Encounter

During your quest to learn how to program you will face many challenges and frustrations. However, the biggest challenge you will face will not be learning the programing language itself, but the many other skills and tools you must learn before writing programs of any significance or gaining any measure of proficiency in solving problems with it. If you are a seasoned student or practicing computer professional returning to the classroom to upgrade your skills, you have the advantage of experience. You can concentrate on learning the syntax and nuances of the language and very quickly apply its powers to problems at hand. If you are an absolute beginner, however, you have much to learn.

1.1 Required Skills

In addition to the syntax and semantics of a programming language you will need to master the following skills and tools:

- The configuration and use of a computer running some version of Microsoft Windows, Linux, or macOS. This includes both operation and incidental system administration.
- The selection and use of a development environment, which could simply be a text editor and command-line tools or an Integrated Development Environment (IDE) that integrates an editor, compiler or interpreter, debugger, and project management applications into one suite of tools
- Problem solving skills
- Project approach techniques

• Project complexity management techniques
• The ability to put yourself in the mood to program
• The ability to stimulate your creativity
• Software analysis, design, and implementation techniques to include object-oriented and functional programming
• Programming language frameworks and libraries

1.2 The Planets Will Come Into Alignment

I use a metaphor to describe what it takes before you can get even the simplest program to execute properly. It's as if the planets must come into alignment. You must learn a little of each skill and tool listed above to write, compile, and run your first program. But, when the planets do come into alignment, and you see your first program execute, and you begin to make sense of all the class notes, documentation, books, blog posts, YouTube videos, and other resources you have studied up to that point, you will leap from your chair and do a victory dance. It's a great feeling!

1.3 Fill Your Buckets

Another way to think about your knowledge acquisition quest is shown in figure 2-1.

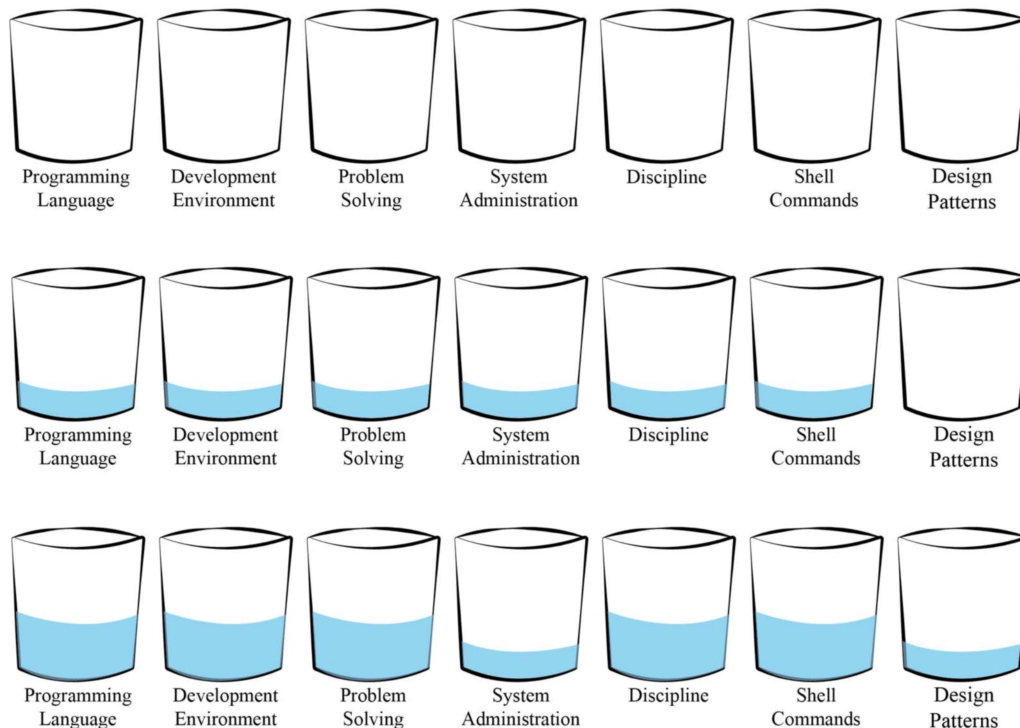

Figure 2-1: The Bucket Analogy

Referring to figure 2-1 — You may start out a complete novice. Fine! Everyone starts out a complete noob. Don't let anyone fool you. I certainly did. Initially, your knowledge buckets are empty. Gradually, as you learn new skills and concepts they begin to fill. Some fill faster than others. The knowledge areas represented above account for but a small sample of the range of topics you'll ultimately encounter along the way.

1.4 How This Chapter Will Help You

In this chapter, I attempt to prepare you for what lies ahead and to help you begin to fill some of your buckets and bring the planets into alignment sooner rather than later. I present an abbreviated software development methodology that formalizes the three primary roles you play as a programming student: *analyst*, *architect*, and *programmer*. I offer tips on how you can tap into the "flow", a transcendental state often experienced by artists when they are completely absorbed in and focused on their work. I also offer several strategies to help manage project complexity, something you will not need to do for very small projects, but will be indispensable for large ones. I recommend you read this chapter at least once in its entirety and refer back to it as necessary as you progress through the text.

Quick Review

The difficulties encountered with learning a programming language lies not with the language itself but with the many other skills that must be mastered along the way. You will find it helpful to know the development roles you must play and to have a project-approach strategy.

2 Personality Traits Of Great Programmers

Software engineers come in all shapes, sizes, and temperaments. I've worked with many over the years. Here I'd like to discuss what I believe are a few of the most important personality traits shared by the best. I'm not trying to describe the perfect person; we all have our strengths and weaknesses. But by observing some really smart people in action, I have formulated a definite opinion regarding the traits they possess that enable them to work well by themselves while at the same time permitting them to shine in a team environment.

2.1 Creative

Like I said at the beginning of the chapter, the most prevalent personality trait great programmers possess is that of *creativity*. Solving problems in such a manner that allows them to be executed by a machine takes truck loads of creativity.

If you say to yourself, "But I'm not creative!" My advice to you is not to sell yourself short. A large part of being creative is simply having an open mind. You must be receptive to alternative solutions and not limit yourself to a "this way or the highway" mode of thinking.

2.2 Tenacious

Great programmers never give up! As computers, operating systems, and programming languages grow increasingly complex, so too grows the complexity of their associated development environments and the range of issues and problems you will encounter when developing solutions in these environments. If you are the type of person who likes to bite into a problem like a pit bull and keep at it until you've kicked its ass, then you'll do well as a programmer.

2.3 Resilient

Great programmers bounce back! When a particular problem has given you a thorough trouncing you must come back strong the next day and fight the battle again. *Programming is one continuous stream of problem solving.* This you must be willing to repeat ad-infinitum. To paraphrase an old Timex® watch advertisement campaign slogan, you must be able to "...take a licking and keep on ticking!"

2.4 Methodical

Great programmers approach everything they do in a methodical way. This holds true regardless of if you work alone or as part of a team, or if a formal methodology exists or not. You must be able to formulate problem attack plans and execute those plans.

2.5 Meticulous

Great programmers are meticulous. Close attention to detail is paramount in the programming profession. One misspelled identifier, one token out of place, can break entire systems.

2.6 Honest

Great programmers can be trusted to do the right thing in the code when no one is looking. They must be honest with themselves but especially with other programmers. Honest programmers put in an honest day's work and give realistic estimates regarding task completion. Honest programmers do not hide their weaknesses.

2.7 Proactive

Great programmers recognize and capitalize upon opportunity. They get up out of their chair and go out and talk to their fellow programmers. When they see problems in the code or areas for improvement they bring it to the attention of the team.

2.8 Humble

Great programmers know when to seek guidance or help. They don't let their ego stand in the way of the greater good. They get up off their duff and talk to their fellow programmers. They share their knowledge and wisdom so that someday they can take a vacation. Most importantly, admitting they don't know something early on can save hundreds of wasted work hours down the line.

2.9 Be a Generalist and a Just-in-Time Specialist

Great programmers are well-versed in many aspects of computing. Rarely have I ever met anyone who referred to themselves as only a this type of programmer or a that type of programmer. I'd rather hire generalists with solid educational backgrounds and the proven ability to teach themselves new tricks, than to bank on a specialist who refuses to grow professionally. In other words, great programmers have a broad range of skills they can apply to the problem. Great programmers can gather requirements, design a solution, write the code, conduct testing, write sup-

porting documentation, deploy the application if necessary, and carry on intelligent conversations with the customer as well. This sounds like a full-stack engineer, doesn't it. Full-stack engineers have been around before full-stack engineering was a thing.

2.10 Focus Like A Laser Beam

Great programmers can focus like a laser beam. They can be surrounded by chaos and still write code. This is due to them being able to get into a transcendental meditative state called the *flow*. I discuss the flow in greater detail later in the chapter.

Quick Review

Great programmers are *creative*, *tenacious*, *resilient*, *methodical*, *meticulous*, *honest*, *proactive*, and *humble*. Great programmers cultivate a broad range of skills and focus on a particular technology when necessary.

3 How To Approach A Project

A large part of being a successful programmer lies in being able to evaluate a problem, design a solution, and then implement the design in a suitable programming language. In this section, I want to discuss the roles you will assume as both a student and as a professional programmer. I will then present a project-approach strategy you can use to help you get started with a new programming project and maintain development momentum.

3.1 Three Software Development Roles

You will find yourself assuming the duties and responsibilities of three software development roles: *analyst*, *architect*, and *programmer*.

3.1.1 Analyst

The first software development role you will play is that of an analyst. When first handed a class programming project you may not understand what, exactly, the instructor is asking you to do. Hey, it happens! Regardless, you, as the student, must read the assignment and design and implement a solution.

Programming project assignments come in several flavors. Some instructors go into painful detail about how they want the student to execute the project. Others prefer to generally describe the type of program they want, thus leaving the details, and the creativity, up to you. There is no one correct method of writing a project assignment; each has its benefits and limitations.

A detailed assignment takes a lot of the guesswork out of what outcome the instructor expects. On the other hand, having every design decision made for you may prevent you from solving the problem in a unique, creative way.

A general project assignment delegates a lot of decision making to the student while also adding the responsibility of determining what project features will satisfy the assignment.

Both types of assignments model the real world to some extent. Sometimes, software requirements are well defined leaving little doubt what shape the final product will take and how it must

perform. More often than not, however, requirements are ill-defined and vaguely worded. As an analyst, you must clarify what is being asked of you. In an academic setting, do this by talking to your instructor and asking them to clarify the assignment. A clear understanding of the assignment will yield valuable insight into possible approaches to a solution.

3.1.2 Architect

The second software development role you will play is that of an architect. Once you understand the assignment you must design a solution. If your project is extremely small, you could perhaps skip this step with no problem. However, if your project contains several objects that interact with each other, then your design, and the foundation it lays, could make the difference between success and failure. A well-designed project reflects a sublime quality that poorly designed projects do not.

Two objectives of good software design are the abilities to accommodate change and tame complexity. Accommodating change, in this context, means the ability to incrementally add features to your project as it grows without breaking the code you have already written.

Several important object-oriented principles have been formulated to help tame conceptual complexity and are discussed at length later in the book. To tame physical complexity, begin by imposing good organization upon your source code files. For simple projects you can group source code files together in one directory. For more complex projects you will want to organize source code files into sub-folders and group related code modules together.

3.1.3 Programmer

The third software development role you will play is that of a programmer. As the programmer, you must implement your design. The important thing to note here is that if you do a poor job in the roles of analyst and architect, your life as a programmer will be miserable. That doesn't mean the design has to be perfect. I will show you how to incrementally develop and make improvements to your design as you code.

Now that you know what roles you will play as a student, let's discuss how you might approach a project.

3.2 A Project-Approach Strategy

Most students have difficulty implementing their first significant programming assignment, not because they lack brains or talent. It's because they lack experience. If you are a novice and feel overwhelmed by your first programming project, rest assured you are not alone. The good news is that with practice and some small victories, you will quickly gain proficiency at formulating approach strategies to your programming assignments.

Even experienced programmers may not immediately know how to solve a problem or write a particular piece of code when tasked to do so. What they do know, however, is how to formulate a strategy to solve the problem.

3.2.1 You Have Been Handed A Project — Now What?

Until you gain experience and confidence in your programming abilities, the biggest problem you will face when given a large programming assignment is where to begin. What you need to

help you in this situation is a *project-approach strategy*. The strategy is presented below and discussed in detail. I have also summarized the strategy in a checklist located in Appendix A. Feel free to reproduce the checklist to use as required.

The project-approach strategy is a collection of areas of concern to take into consideration when you begin a programming project. It's not a hard, fast list of steps you must take. It's intended to put you in control, to point you in the right direction, and to give you food for thought. It's flexible. You may not need to consider every area of concern for every project. After you have used it a few times to get started, you may never use it explicitly again. As your programming experience grows, feel free to tailor the project-approach strategy to suit your needs.

3.2.2 Areas of Concern

As I mentioned above, the project-approach strategy consists of several areas of concern. These areas of concern include *application requirements*, *problem domain*, *language features*, and *application design*. When you use the strategy to help you solve a programming problem, your efforts become focused and organized rather than ad hoc and confused. You will feel like you are making real progress rather than drowning in a sea of confusion.

3.2.2.1 Application Requirements

An application requirement is an assertion about a particular aspect of expected application behavior. A project's application requirements are contained in a project specification or programming assignment. Before you proceed with the project you must ensure you completely understand the project specification. Seek clarification if you do not know or if you are not sure what problem the project specification is asking you to solve. In my academic career, I have seen projects so badly written that I thought I had a comprehension problem. I'd read the thing over and over again until struck by a sudden flash of inspiration. But more often than not, I would verify what I believed the professor required by asking them to clarify any points I did not understand.

3.2.2.2 Problem Domain

The problem domain is the body of knowledge necessary to implement a software solution apart and distinct from the knowledge of programming itself. For example, consider the following application requirement: "Write a program to simulate elevator usage in a skyscraper." You may understand what is being asked of you (requirements understanding) but not know anything about elevators, skyscrapers, or simulations (problem domain). You need to become enough of an expert in the problem domain for what you are solving such that you understand the issues involved. In the real world, subject matter experts (SMEs) augment development teams, when necessary, to help developers understand complex problem domains.

3.2.2.3 Programming Language Features

One source of great frustration for novice programming students at the opening stages of the project is knowing what solution to design but not knowing enough of the programming language features to start the design process. This is when panic sets in and students begin to buy extra books in hopes of discovering the Holy Grail of project wisdom. (*The cheaters try to get experienced programmers on the Internet to do their projects for them and learn nothing in the process.*)

Don't panic! Make a list of the language features you don't understand, then study each one, marking it off your list as you go. This provides focus and a sense of progress. As you read about each feature, make notes on its usage, then refer to your notes when you sit down to formulate your design.

3.2.2.4 High-Level Design & Implementation Strategy

When you are ready to design a solution, you will usually be forced to think along three completely different lines of thought: *procedural* vs. *object-oriented* vs. *functional*.

3.2.2.4.1 Procedural-based Design Approach

A procedural-based design approach identifies and implements program data structures separately from the program code that manipulates those data structures. When taking a procedural-based approach to a solution you generally break the problem into small, easily solvable pieces or units called functions, implement the solution to each function separately, and then combine the functions into a complete solution. This analysis methodology is also known as functional decomposition.

Note that functional decomposition represents an approach to systems analysis. The procedural-based design would follow from the analysis with the resulting application architecture resembling the functionally decomposed system diagram as shown in figure 2-2.

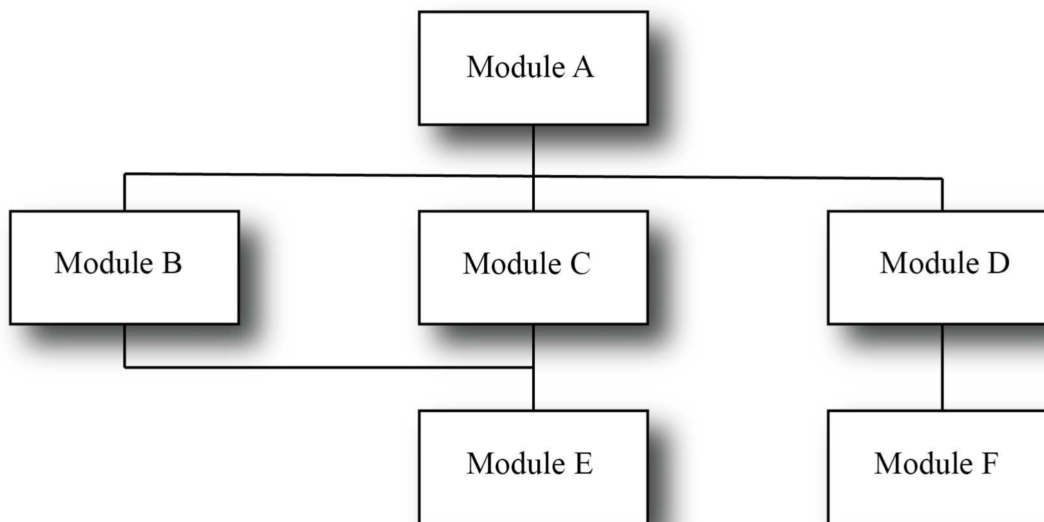

Figure 2-2: Module Hierarchy of Functionally Decomposed System

Referring to figure 2-2 — Applying functional decomposition results in a hierarchy of application module dependencies. The top-most module in the hierarchy represents the sum of all behavior implemented by the system submodules such that a change to one of the submodules would affect the behavior of parent modules. Modules at the top of the hierarchy depend on modules below them. In this diagram, a change to module E would affect modules B and C, which in turn would affect the behavior of module A.

0
0
0
0
0
0
1
0

3.2.2.4.2 Object-Oriented Design Approach

Object-oriented design entails thinking of an application in terms of objects and the interactions between those objects. This approach no longer considers data structures and the methods that manipulate those data structures to be separate. The data an object needs to do its work is contained within the object itself and resides behind a set of public interface methods. (Encapsulation) Data structures and the methods that manipulate them combine to form classes from which objects can then be created.

To solve a programming problem with an object-oriented approach, decompose it into a set of objects and their associated behavior. You can use design tools such as the Unified Modeling Language (UML) to help with this task. Once you've identified system objects, you then define object interface methods. From here you declare classes or structures and implement those interface methods. Finally, you combine these classes or structures together to form the final program. (*This usually takes place in an iterative fashion over a period of time according to a well-defined development process.*) Note that when using the object-oriented approach, you are still breaking a problem into solvable pieces, only now the solvable pieces are objects that represent the interrelated parts of a system.

The primary reason the object-oriented approach is superior to functional decomposition is due to the isomorphic mapping between the problem domain and the design domain as figure 2-2 illustrates.

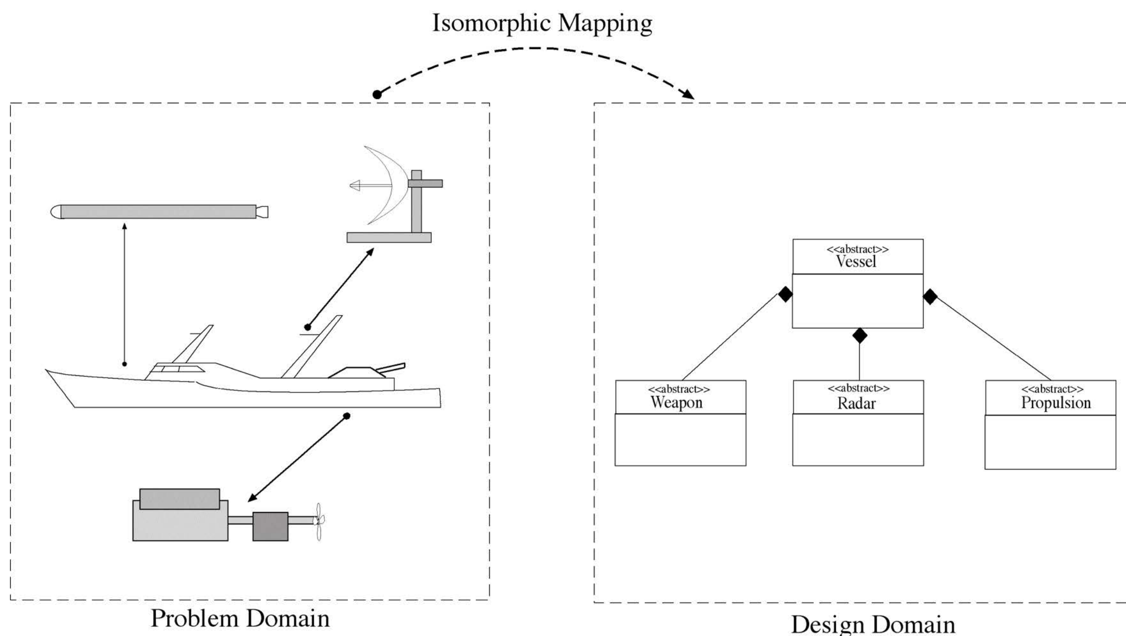

Figure 2-3: Isomorphic Mapping Between Problem Domain and Design Domain

Referring to figure 2-2 — Real world objects like weapon systems, sensors, propulsion systems, and vessels have a corresponding representation in the software system design. The correlation between real world objects and software components fuels the power of the object-oriented approach.

Object-oriented analysis and design does not negate the helpfulness of functional decomposition, but it does turn the dependency hierarchy in its head as shown in figure 2-4.

 © 2024 Pulp Free Press

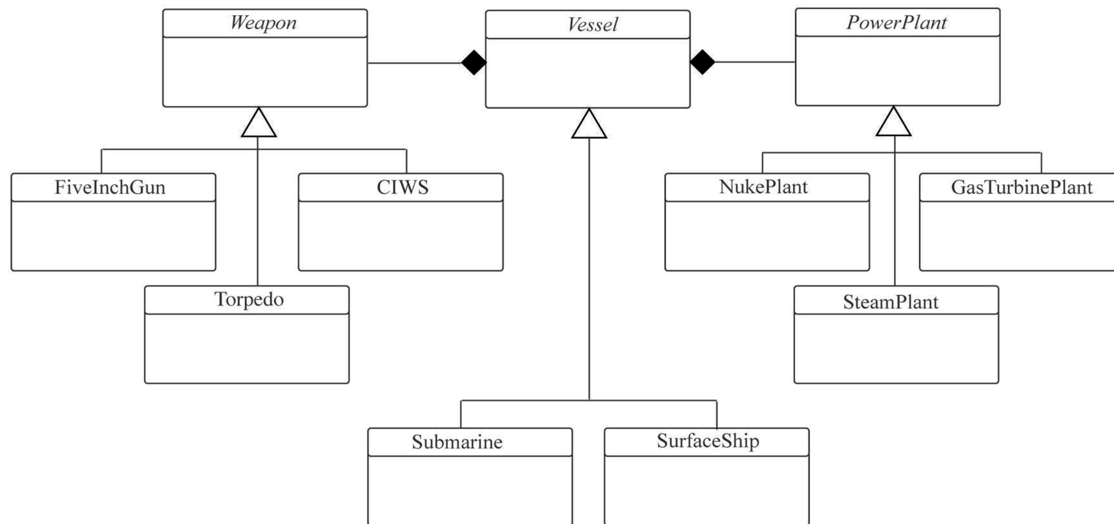

Figure 2-4: Fleet Simulation UML Diagram

Referring to figure 2-4 — Abstract classes sit at the top of the inheritance hierarchy and specify the interface of derived classes. The subclasses depend on the base classes and thus invert the dependencies of traditional application architectures derived via functional decomposition alone.

Another way to think about object-oriented programming is it defines a range of operations on a set of things. Again referring to figure 2-4 — The abstract base classes Weapon, Vessel, and PowerPlant map to real-world objects and define a set of authorized operations on those objects. Subclasses represent new types of things manipulated by the same methods defined in the base class. Compare this to the functional design approach discussed in the following section.

3.2.2.4.3 Functional Design Approach

Functional design and programming differers from object-oriented design and programming primarily by focusing on the data and the functions that manipulate that data. Whereas it's easy to create new types of objects in an object-oriented design (simply create a subclass), those new objects are manipulated by a relatively fixed set of methods specified by base classes and implemented by derived classes.

In a functional programming paradigm, the type and number objects remains fairly stable, but new functions can be created to manipulate those objects in different and unique ways. Objects, in functional programming, refer mostly to collections of data, and data remains immutable.

In object-oriented programming, an object can contain data that represents an objects state. Object state data can be mutated via an object's methods and properties.

Functional programming takes pains to maintain object immutability. An operation on data must not mutate the data. Pure functional programming languages often employ a declarative syntax as compared with object-oriented and procedural-based languages employed in an imperative fashion. One is not better than the other, but functional languages are better at solving some sorts of problems while object-oriented programming languages are better at solving others.

3.2.3 Think Abstractly

One mistake students often make is to think too literally. It is very important to remember that the act of solving a real world problem with a computer requires abstraction. The real world is too complex to model sufficiently with a computer program. One day, perhaps, the human race will produce a genius who will show us how it's done. Until then, analysts must focus on the essence of a problem and distill unnecessary details into a tractable solution that can then be modeled effectively in software.

3.2.4 The Strategy In A Nutshell

The project-approach strategy can be summarized as follows: Identify the problem, understand the problem, make a list of language features you need to study, and check them off as you go. Once you formulate a solution to the problem, break the problem into manageable pieces, solve each piece of the problem, and then combine the solved pieces to form a total solution.

3.2.5 Applicability To The Real World

The project-approach strategy presented previously is not intended to replace a formal course on software engineering, but it will help you when you enter the real world as a professional programmer. In that world, you will soon discover that all companies and projects are not created equal. Different companies have different software development methodologies. Some companies have no software development methodology. If you find yourself working for such a company, you will probably be the software engineering expert. Good luck!

Quick Review

The three development roles you will play as a student are those of *analyst*, *architect*, and *programmer*. As the analyst, strive to understand the project's requirements and what must be done to satisfy those requirements. As the architect, you are responsible for the design of your project. As the programmer, you will implement your project's design in a programming language.

The project-approach strategy helps both novice and experienced developers systematically formulate solutions to programming projects. The strategy deals with the following areas of concern: *application requirements*, *problem domain*, l*anguage features*, and *application design*. By approaching projects in a systematic way, you can put yourself in control and maintain a sense of forward momentum during the execution of your projects. The project-approach strategy can also be tailored to suit individual needs.

4 The Art Of Programming

Programming is an art — it takes a lot of creativity to solve problems with a computer. Creative people have an advantage in that they are not afraid to explore new avenues of design. Their open-mindedness and readiness to accept new ideas gives them the ability to see problems differently from people who tend towards the "cut and dry". This section offers a few suggestions on how you can stimulate your creativity.

4.1 Don't Start At The Computer

Unless you have a good idea about what code to write, sitting down at the computer without first thinking through some design issues is the worst mistake you can make. If you have ever suffered from writer's block when writing a paper for class, then you can begin to understand what you will experience if you begin your project at the computer.

I recommend you forget the computer, go someplace quiet and relaxing with pen and paper, and draft a design document. It doesn't have to be big or too detailed. Entire system designs can be sketched on the back of a napkin. The important thing is that you give some thought regarding your program's design and structure before you start coding.

Your choice of relaxing locations is important. It should be someplace where you feel really comfortable. If you like quiet spaces, then seek quiet spaces; if you like to watch people walk by and observe the world, then an outdoor café may be the place for you. Inside, outside, at the beach, on the ski slope, wherever you prefer.

What you seek is the ability to let your mind grind away on the solution. Let your mind do the work. Writing code at the computer is a mechanical process. Formulating the solution is where real creativity is required, and is the part of the process that requires the most brainpower. Typing code is simply an exercise in attention to detail.

4.2 Inspiration Strikes At The Weirdest Time

If you let your mind work on the problem, it will offer its solution to you at the weirdest times. I solve most of my programming problems in my sleep. As a student, I kept computers in the bedroom and would get up at all hours of the night to work on ideas that popped into my head in a dream.

Try to have something to write on close at hand at all times. A pad of paper and pen next to the bed or next to the toilet can come in handy! You can also use a small tape recorder, digital memo recorder, or your smart phone. Whatever means suit your style. Just be prepared. There's nothing worse than the sinking feeling of having had the solution come to you in the middle of the night, in the shower, or on the drive home from work or school, only to forget it later. You'll be surprised at how many times you'll say to yourself, "Hey, that will work!" only to forget it and have no clue what you were thinking when you finally get hold of a pen.

4.3 Own Your Own Computer

Do not rely on the computer lab! I repeat, do not rely on the computer lab! The computer lab is the worst possible place for inspiration and cranking out code. If at all possible, own your own computer. It should be one sufficiently powerful to use for software development.

4.4 Time and No Money or Money and No Time

The one good reason for not having your own personal computer is severe economic hardship. Full-time students sometimes fall into this category. If you are a full-time student, what you usually have instead of a job or money is gobs of time. You have so much time that you can afford to spend your entire day at school and complain to your friends about not having a social life. But you can stay in the computer lab all day long, even when it is relatively quiet.

On the other hand, you may work full-time and be a part-time student. If this describes you, then you don't have time to screw around driving to school to use the computer lab. You will gladly pay for any book or software package that makes your life easier and saves you time.

4.5 The Family Computer Is Not Going To Cut It!

If you are a family person working full-time and attending school part-time, then your time is a precious commodity. If you have a family computer that everyone shares, adults as well as children, then get another computer, put it off limits to everyone but yourself, and password-protect it. This will ensure that your loving family does not accidentally wipe out your project the night before it is due. Don't kid yourself, it happens. Ensure your peace of mind by having your own computer in your own little space with a sign on it that reads, "Touch This Computer And Die!"

4.6 Set The Mood

When you have a good idea on how to proceed and you're ready to code, you will want to set the proper mood.

4.6.1 Location, Location, Location

Locate your computer work area someplace free from distraction. If you're single this is a whole lot easier than if you're married with children. If you live in a dorm or frat house, good luck! Perhaps the computer lab is an alternative for you after all.

Have your own room, if possible, or at least your own corner of a larger room that is recognized as a quiet zone. Noise-canceling headphones might help if you find yourself in this situation.

Set rules. Let your friends and family know that it's not cool to bother you when you are programming. I know it sounds rude, but when you get into the flow, which I discuss in the following section, you will become agitated when someone interrupts your train of thought to ask you about school lunch tomorrow or the location of the car keys. Establish the ground rules up front that specify when it's a good time to disturb you when you are programming. The rule is never!

4.7 Concept Of The Flow

Artists tend to become absorbed in their work, not eating and ignoring personal hygiene for days, even weeks, at a time. Those who have experienced such periods of intense concentration and work describe it as a transcendental state where they have complete clarity of the idea of the finished product. They tune out the world around them, living inside a cocoon of thought and energy that propels them forward.

Programmers get into the flow. I get into the flow. You will get into the flow. When you do, you will crave the feeling of the flow again. It is a good feeling, one of complete and utter understanding of what you are doing and where you are going with your code. You can do amazing amounts of programming while in the flow.

4.7.1 The Stages of Flow

As with sleep, there are stages to the flow.

4.7.1.1 Getting Situated

The first stage: You sit down at the computer and adjust your keyboard and stuff around you. Take a few deep breaths to help you relax. By now, you should have a good idea of how to proceed with your coding. If not, you shouldn't be sitting at the computer.

4.7.1.2 Restlessness

The second stage: You may find it difficult to clear your mind of the everyday thoughts that block your creativity and energy. Maybe you had a bad day at work, or an especially good day. Perhaps your partner or significant other is being a complete ass! Or perhaps they're being especially nice and you're wondering WTF?

Close your eyes and take deep and regular breaths. Clear your mind and try to think of nothing. It is hard to do at first, but with practice it becomes easy. When you can clear your mind and free yourself from distracting thoughts you will begin to relax and find yourself ready to start coding.

4.7.1.3 Settling In

The third stage: Now your mind is clear. Non-productive thoughts are tucked neatly away. You begin to code. Line by line, your program takes shape. You settle in. The clarity of your purpose takes hold and propels you forward.

4.7.1.4 Calm and Complete Focus

The fourth stage: You don't notice it at first, but at some point between this stage and the previous stage, you have slipped into a deeply relaxed state. You are utterly focused on the task at hand. It is like becoming completely absorbed in a good book. Someone can call your name, but you will not notice. You will not respond until someone either shouts at you or does something to break your concentration.

You will know you were in the flow, if only to a small degree, when an interruption brings you out of this focused state, leaving you feeling agitated and eager to settle in once again. If you avoid getting up from your chair for fear of breaking your concentration or losing your train of thought, then you are in the flow!

4.8 Be Extreme

Kent Beck, in his book Extreme Programming Explained, describes the joy of doing really good programming. The following software development cycle is synthesized from his extreme programming philosophy.

4.8.1 The Software Development Cycle

4.8.1.1 Plan

Plan a little. Your project design should serve as a guide in your programming efforts. Your design should also be flexible and accommodate change. This means that as you program, you may make changes to the design.

Essentially, you will want to design to the point where you have enough of the design to allow you to begin coding. The act of coding will either soon reinforce your design decisions or uncover fatal flaws you must correct if you hope to have a polished, finished project.

4.8.1.2 Code

Code a little. Write code in small, cohesive modules. A class or method at a time usually works well.

4.8.1.3 Test

Test a lot. Test each class, module, or method both separately and in whatever grouping makes sense. You will find yourself writing little programs on the side called unit tests to test the code you have written. This is a good practice to get into. A unit test is nothing more than a little program you write and execute in order to test the functionality of some component or feature before integrating that component or feature into your project. The objective of testing is to break your code and correct its flaws before it has a chance to break your project in ways that are hard to detect.

4.8.1.4 Integrate & Regression Test

Integrate often and perform regression testing. Once you have a tested module of code, be it either a method or complete set of related classes, integrate the tested component(s) into your project regularly. The objective of regular integration and regression testing is to see if the newly integrated component or newly developed functionality breaks any previously tested and integrated component(s) or integrated functionality. If it does, then remove it from the project and fix the problem. If a newly integrated component breaks something, you may have discovered a design flaw or a previously undocumented dependency between components. If this is the case, then the next step in the programming cycle should be performed.

4.8.1.5 Refactor

Refactor the design whenever possible. If you discover design flaws or ways to improve the design of your project, you must revise and improve the design to accommodate further development. An example of design refactoring is the migration of common elements from derived classes into the base class to take better advantage of code reuse.

4.8.1.6 Repeat

Apply the programming cycle in an iterative fashion. You will quickly reach a point in your project where it all starts to come together, and very quickly so.

4.8.2 The Programming Cycle Summarized

Plan a little, code a little, test a lot, integrate often, refactor the design when possible. Don't Wait Until You Think You Are Finished Coding The Entire Project To Compile! Trying to write the entire program before compiling a single line of code is the most frequent mistake new programmers tend to make. The best advice I can offer is this: don't do it! Use the iterative program-

ming cycle previously outlined. Nothing will depress you more than seeing a million compiler errors scroll up the screen after waiting until the bitter end to compile your project.

4.8.3 A Helpful Trick: Stubbing

Use function and method stubbing to speed development and avoid writing a ton of code just to get something useful to work. Stubbing is a programming trick that is best illustrated by example.

Suppose your project requires you to display a text-based menu of program features on the screen. The user would then choose one of the menu items and press ENTER, thereby invoking that menu command. What you would really like to do first is write and test the menu's display and selection methods before worrying about having it actually perform the indicated action. You can do exactly that with stubbing.

A stubbed method, in its simplest form, is a method with an empty body. It's also common to have a stubbed method display a simple message on the screen saying in effect, "Yep, the program works great up to this point. If it were actually implemented, you'd be using this feature right now!"

Stubbing is a great way to incrementally develop your project and maintain momentum. Stubbing will change your life!

4.8.4 Fix The First Compiler Error First

OK. You try to run some code and it results in a slew of errors. What should you do? I recommend you stay calm, take a deep breath, and fix the first compiler error first. Not the easiest error, but the first error. The reason for this is that the first compiler error, if fatal, will generate other compiler errors. Fix the first one first, and you will generally find a lot of the other errors will also be resolved. If you pick an error from the middle of the pack and fix it, you may introduce more errors into your source code. Fix the first compiler error first!

Quick Review

Programming is an art. Formulating solutions to complex projects requires lots of creativity. There are certain steps you can take to stimulate your creative energy. Sketch the project design before sitting at the computer. Reserve quiet space in which to work and, if possible, have a computer dedicated to school and programming projects.

When you're ready to start coding, apply the software development cycle in iterations. Plan a little, code a little, test a lot, integrate and test, and refactor. Use stubbing to maintain momentum by postponing implementation details until later in the development cycle.

If things go south when you run your code, fix the first error first.

5 Managing Project Complexity

Software engineers generally encounter two types of project complexity: *conceptual* and *physical*. All programming projects exhibit both types of complexity to a certain degree, but the approach and technique used to manage small-project complexity will prove woefully inadequate

when applied to medium, large, or extremely large programming projects. This section discusses both types of complexity and suggests an approach for the management of each.

5.1 Conceptual Complexity

Conceptual complexity is that aspect of a software system that is manifested in, dictated by, and controlled by its architectural design. A software architectural design is a specification of how each software module or component will interact with other software components. A project's architectural design is a direct result of the approach conceived by one or more software engineers to implement a solution for a particular problem domain. In formulating this solution, the software engineers are influenced by their education and experience, available technology, and project constraints.

An engineer versed in procedural programming and functional decomposition techniques will approach the solution to a programming problem differently from an engineer versed in object-oriented analysis and design techniques. The former will think in terms of modules and sub-modules, while the latter will draw a direct correlation to real world objects and their derived software components. The functional decomposition approach will almost always yield software modules that are difficult to use out of context. Modules are so tightly integrated with each other that extracting one for reuse in another system may be impossible. Software architectures based on functional decomposition tend to be brittle and change resistant. By brittle, I mean that a change in one module will have negative effects on other, seemingly unrelated modules. Software based on such change resistant architectures is hard to maintain, modify, or extend.

An understanding of software design patterns will give the object-oriented engineer a double advantage. Software design patterns capture the knowledge and experience of many talented software engineers. Their use can significantly increase the flexibility, maintainability, and extensibility of the applications upon which they are based.

However, writing a program using object-oriented programming constructs does not automatically result in a good object-oriented architecture. It takes lots of training and practice to develop good, robust, change-receptive and resilient software architectures.

5.1.1 Managing Conceptual Complexity

Conceptual complexity can either be tamed by a good software architecture, or it can be aggravated by a poor one. Software architectures that seem to work well for small to medium-sized projects will be difficult to implement and maintain when applied to large or extremely large projects.

Tame conceptual complexity by applying sound object-oriented analysis and design principles and techniques to formulate robust software architectures that are well-suited to accommodate change. Well-formulated object-oriented software architectures are much easier to maintain compared to procedural-based architectures of similar or smaller size. That's right — large, well-designed object-oriented software architectures are easier to maintain and extend than small, well-designed procedural-based architectures. It's easier for object-oriented programmers to "wrap their heads around" an object-oriented design than it is for programmers of any school of thought to get their heads around a procedural-based design.

 © 2024 Pulp Free Press

5.1.2 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is the de facto standard modeling language of object-oriented software engineers. UML provides several types of diagrams, which are used during various phases of the software development process such as use-case, component, class, and sequence diagrams. However, UML is more than just pretty pictures. UML is a modeling meta-language implemented by software-design tools like No Magic, Inc's MagicDraw (*Now owned by 3ds.com*). Software engineers can use these design tools to control the complete object-oriented software engineering process. In this book I use UML class and sequence diagrams to illustrate program designs.

5.2 Physical Complexity

Physical complexity is that aspect of a software system determined by the number of design and production documents and other artifacts produced by software engineers over the course of the project's lifetime. A small project will generally have fewer, if any, design documents compared to a large project. A small project will also have fewer source code files than a large project. As with conceptual complexity, the steps taken to manage the physical complexity of small projects will prove inadequate for larger projects. However, there are some techniques you can learn and apply to small programming projects that you can in turn use to help manage the physical complexity of large projects as well.

5.2.1 Managing Physical Complexity

You can manage physical complexity in a variety of ways. Selecting appropriate class names and package structures are two basic techniques that will prove useful not only for small projects, but for large projects as well. However, large projects usually need some sort of configuration-management tool to enable teams of programmers to work together on large source-code repositories. Git and Subversion are two examples of configuration-management tools. In this book you will learn how to create and work with Git repositories.

5.3 The Relationship Between Physical and Conceptual Complexity

Physical complexity is related to conceptual complexity in that the organization of a software system's architecture plays a direct role in the organization of a project's physical source code files. A simple programming project with a handful of classes might be grouped together in one directory. It might be easy to compile every class in the directory at the same time. However, the same one-directory organization will simply not work on a large project with teams of programmers creating and maintaining hundreds or even thousands of source code files or other project artifacts.

5.4 Maximize Cohesion — Minimize Coupling

An important way to manage both conceptual and physical complexity is to maximize software module cohesion and minimize software module coupling.

Cohesion is the degree to which a software module focuses on its intended purpose. A high degree of cohesion is desirable. For example, a method intended to display an image on the screen would have high cohesion if that's all it did, and poor cohesion if it did some things unrelated to image display.

Coupling is the degree to which one software module depends on external software modules. A low degree of coupling is desirable. Coupling can be controlled in object-oriented software by depending upon interfaces or abstract classes rather than upon concrete implementation classes. These concepts are explained in detail later in the book.

Quick Review

There are two types of complexity: *conceptual* and *physical*. Object-oriented programming and design techniques help manage conceptual complexity. Physical complexity is managed with smart project file-management techniques, by splitting projects into multiple files, and by using namespaces to organize source code.

6 The Engineer's Notebook

If you don't already keep one, I strongly recommend you start and maintain an engineer's notebook. The primary purpose of an engineer's notebook is to record design challenges and their solutions for future reference. What you ultimately jot down in your engineer's notebook will be as personal and as varied as you are unique as an individual. In mine I write down system configuration settings when I install software like databases or application servers. I can't recall how many times I've had to refer back to my notebook to recall a particular configuration setting.

I also capture customer requirements or change requests in my notebook, which I later transfer to a formal requirements management system. I make design sketches in UML or list changes that I must make to the code during a particular day's work. In this regard it functions as a daily work journal. If your boss asks, "What did you do for me this year?" you should be able to point to your engineer's notebook and astound him with your productivity and cunning.

Mostly, however, I record particularly vexing development problems and, when I've found a suitable solution, I write that down too. This has saved me countless hours I'd normally spend solving the same problem again, which is likely to happen if enough time has passed between subsequent encounters.

Over the years I've flirted with different notebook formats from loose-leaf paper in a 3-ring binder to single subject spiral notebooks. Lately I've settled on the college-ruled, 100 sheet, cardboard-covered composition book that looks like figure 2-5.

Referring to figure 2-5 — You can find these notebooks practically everywhere stationary supplies are sold. I prefer these because they're slightly more compact than standard 8 1/2 x 11 inch paper while not being too small. The front and back covers are nice and stiff which allows easy writing on both sides of the paper from the first page to the last whereas loose leaf paper in a 3-ring binder is too bulky and clumsy to work with, especially when sitting in front of a server taking notes on deployment configuration.

When I start a new notebook I write the Start Date on the front cover and when I've filled it up I note the End Date as well so I can see at a glance the period it covers.

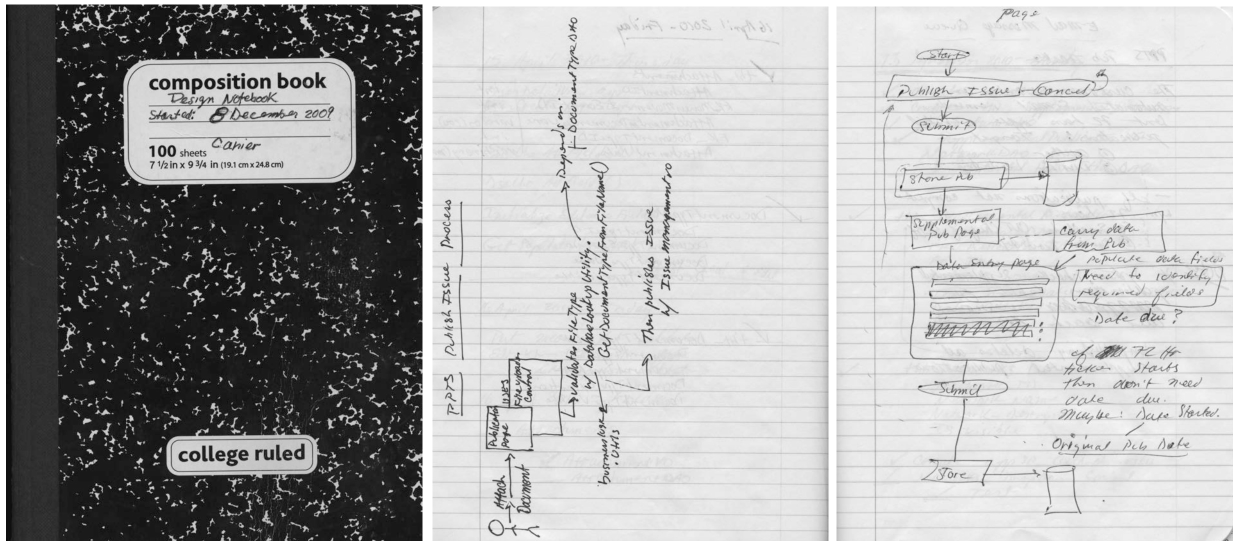

Figure 2-5: Engineer's Notebook: College-Ruled Composition Book with Sample Pages

You'll find your engineer's notebook to be an invaluable resource as you grow professionally as a software engineer.

Quick Review

Start and maintain an engineer's notebook. Use your engineer's notebook to sketch application architectures, frequently-used commands, newly-acquired language features, configuration settings, etc.

Summary

The difficulty with learning a programming language lies not with the language itself, but with the many other skills that must be mastered along the way. You will find it helpful to know the development roles you must play and to have a project-approach strategy.

Great programmers are *creative*, *tenacious*, *resilient*, *methodical*, *meticulous*, *honest*, *proactive*, and *humble*. Great programmers cultivate a broad range of skills and focus on a particular technology when necessary.

The three development roles you will play as a student are those of *analyst*, *architect*, and *programmer*. As the analyst, strive to understand the project's requirements and what must be done to satisfy those requirements. As the architect, you are responsible for the design of your project. As the programmer, you will implement your project's design in a programming language.

The project-approach strategy helps both novice and experienced students systematically formulate solutions to programming projects. The strategy deals with the following areas of concern: *application requirements*, *problem domain*, *language features*, and *application design*. By approaching projects in a systematic way, you can put yourself in control and maintain a sense of forward momentum during the execution of your projects. The project-approach strategy can also be tailored to suit individual needs.

Programming is an art. Formulating solutions to complex projects requires lots of creativity. There are certain steps you can take to stimulate your creative energy. Sketch the project design before sitting at the computer. Reserve quiet space in which to work and, if possible, have a computer dedicated to school and programming projects.

There are five steps to the programming cycle: *plan*, *code*, *test*, *integrate*, and *refactor*. Use function and method stubbing to test parts of a program without having to code the entire function or method.

There are two types of complexity: *conceptual* and *physical*. Object-oriented programming and design techniques help manage conceptual complexity. Physical complexity is managed with smart project file-management techniques, by splitting projects into multiple files, and by using namespaces to organize source code.

Start and maintain an engineer's notebook. Use your engineer's notebook to sketch application architectures, frequently-used commands, newly-acquired language features, configuration settings, etc.

Skill-Building Exercises

1. **Imperative vs. Declarative Programming:** Research the differences between imperative programming languages and declarative programming languages.

2. **Procedural vs. Object-Oriented vs Functional Programming:** Research the differences between these programming paradigms. What programming languages support each paradigm? How is each paradigm supported in Python?

3. **Functions as First-Class Objects:** In Python, functions are considered first-class objects. Research the meaning of that term and explain in your own words what it means.

Suggested Projects

1. **Feng Shui:** If you haven't already done so, stake your claim to your own quiet, private space where you will work on your programming projects. If you are planning on using the school's programming lab, stop by and familiarize yourself with the surroundings.

2. **Procure and Install IDE:** If you are doing your programming on your own computer make sure you have procured and loaded an integrated development environment (IDE) that will meet your programming requirements. If in doubt check with your instructor.

3. **Project-Approach Strategy Checklist:** Familiarize yourself with the project-approach strategy presented in this chapter and summarized in a checklist in Appendix A.

4. **Obtain Reference Books:** Seek your instructor's, friend's, or colleague's recommendation on any reference books that might be helpful to you during this course. There are also many good

computer book-review sites available on the Internet. Also, there are many excellent reference books listed in the reference section of each chapter in this book.

5. **Web Search:** Conduct a search for Python-related websites. Bookmark any site you feel might be helpful to you as you progress through the book. The Python.org documentation site should be first on your list.

Self-Test Questions

1. List at least seven skills you must master in your studies of any programming language.
2. What three development roles will you play as a student?
3. What is the purpose of the project-approach strategy?
4. List and describe the four areas of concern addressed in the project-approach strategy.
5. List and describe the five steps of the programming cycle.
6. What are the two types of complexity?
7. What is meant by the term *isomorphic mapping*?
8. Why do you think it would be helpful to write self-commenting source code?
9. What can you do in your source code to maximize cohesion?
10. What can you do in your source code to minimize coupling?

References

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000. ISBN 201-61641-6

Daniel Goleman. *Emotional Intelligence: Why it can matter more than IQ*. Bantam Books, New York, NY. ISBN: 0-553-37506-7

Notes

0
0
0
0
0
0
1
0

© 2024 Pulp Free Press