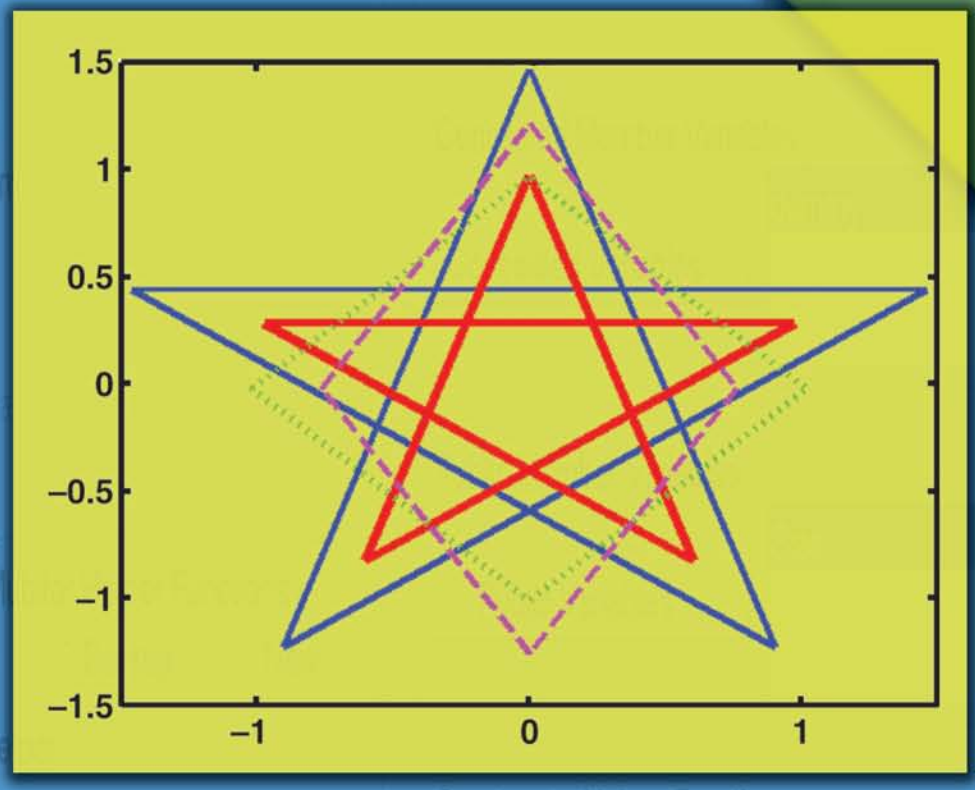
Andy H. Register

A Guide to MATLAB®
Object-Oriented Programming



Chapman & Hall/CRC
Taylor & Francis Group

SCITECH
PUBLISHING, INC.

A Guide to MATLAB®
Object-Oriented Programming

A Guide to MATLAB®
Object-Oriented Programming

Andy H. Register

Georgia Tech Research Institute
Atlanta, Georgia, U.S.A.

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20140114

International Standard Book Number-13: 978-1-58488-912-0 (eBook - PDF)

Dedication

For Mickey

Table of Contents

Figures

Code Listings

Tables

Supplementary Resources Disclaimer

Additional resources were previously made available for this title on CD. However, as CD has become a less accessible format, all resources have been moved to a more convenient online download option.

You can find these resources available here: www.routledge.com/9781584889113

Please note: Where this title mentions the associated disc, please use the downloadable resources instead.

About the Author

Andy Register has been an admitted object-oriented fanatic since his first introduction to the concepts of object-oriented design in the late 1980s. At that time, he was working on his doctoral degree in electrical engineering at the Georgia Institute of Technology, Atlanta. His research involved the real-time control of nonminimum phase systems, human and hardware-in-the-loop simulations, state-of-the-art computer architectures, and low-level programming of multiple-instruction multiple-data (MIMD) parallel computers. Object-oriented programming was still in its infancy with a number of object-oriented contenders: Actor, C++, CLOS, Eiffel, Flavors, and Smalltalk, among others. Dr. Register needed a language that supported a close association between software and hardware, and he found the right combination of performance, utility, and elegance in C++. After using C++ for several years, he published his first two papers on object-oriented programming in 1994.

Fast-forward to the twenty-first century, and we find Dr. Register working at the Georgia Tech Research Institute in Atlanta on complex radar-tracking simulations. These simulations do not require a close association with hardware so that real-world interface requirements dictate much of the software design. In this environment, an object-oriented approach to MATLAB yields big advantages. Dr. Register brought his years of experience developing object-oriented C++ software to bear on MATLAB and developed a set of techniques and tools that allows a standard object-oriented design to peacefully coexist with MATLAB. In his day-to-day work, these techniques allow for interchangeable modules and the capability to add new features to a simulation. In this book, these techniques are described and Dr. Register's Class Wizard tool is explained and demonstrated.

Preface

I am an admitted object-oriented fanatic. I have been designing and implementing object-oriented software for more than twenty years. When I started designing and implementing object-oriented MATLAB®, I encountered many detractors. They would say things like "The object model isn't complete," "You can't have public variables," "The development environment doesn't work well with objects," "Objects and vector operations don't mix," "Object-oriented code is too hard to debug," and "MATLAB objects are too slow." None of these statements matched my experience with MATLAB objects. It quickly became obvious that MATLAB objects don't have a capability problem; rather, they have a public relations problem. Part of the public relations problem stems from the fact that the sheer genius behind the design and implementation of MATLAB's object-oriented extensions is masked by the abbreviated discussion in the user's guide. If you want to use MATLAB to develop object-oriented software, ignore the critics, study the examples in this book, and reap the benefits.

Mark Levedahl exposed me to the possibility of developing object-oriented MATLAB software in 2001. Both of us had written a lot of C++ code, and we spoke the same object-oriented dialect. MATLAB objects are seductive because they seem so easy. Without help, trying to get everything right is anything but easy. My first object-oriented implementation was terrible. Construction was dicey. Interfaces were terrible. Modules were slow. The code was very hard to maintain. Maybe the critics were right. I was still learning. The lessons improved the next implementation, but there still seemed to be a fundamental difference between, for example, object-oriented programming in C++ and object-oriented programming in MATLAB.

MATLAB object-oriented code always bumped up against the same limitation. The elements spelled out in the object-oriented design didn't map easily to an m-file implementation. Part of the reason for the poor match comes from the fact that each design element must be spread into more than one m-file: one module to get a value, another module to set it, and yet another to display it. In an evolving design, files can easily get out of synch. Couple this with the fact that a developer is free to define the mapping, and the result can be chaos. Faced with many competing alternatives, it is fair to ask, "Is one alternative better than the others?" After a lot of consideration and study, I believe the answer is yes. Following the best alternative improved the object-oriented implementations by orders of magnitude. Armed with the best mapping, a software tool to keep the modules and design in synch is a matter of design and implementation. Version 3 of the MATLAB Class Wizard will do this. The Class Wizard tool is included on this book's companion CD.

The first version of Class Wizard was not easy to use. George Brown, Kyle Harrigan, and Mike Baden used it with some success. Their comments helped shape the graphical user interface in the current version. At the same time, I was using my Class Wizard tool to create object-oriented code for a large MATLAB project. That project, the Target Tracking Benchmark, was primarily sponsored by the Missile Defense Agency and involved a cadre of accomplished MATLAB programmers from government, academia, and industry. Good techniques were allowed to blossom, and the bad were very quickly rooted out. Reactions to MATLAB objects were mixed. The ensuing debates improved everyone's understanding of the risks and benefits. Over time, the debate participants included Mark Levedahl, Steve Waugh, Laura Ritter, Dale Blair, Phil West, George Brown, Paul Miceli, Terry Ogle, Paul Burns, Chris Burton, Lisa Ehrman, Dan Leatherwood, Darin Dunham, Steve Kay, Al de Baroncelli, Ron Rothrock, Bob Isbell, Bruce Douglas, Greg Watson, Ben Slocumb,

Mike Klusman, Jim Van Zandt, and Joe Petruzzo. These gifted individuals improved my understanding of MATLAB objects and helped shape the second and current versions of Class Wizard.

The second version of Class Wizard was easier to use, and about three years ago I set out to write a user's guide for it. I quickly discovered that telling someone how to use a tool is a lot different from telling someone why. Many MATLAB programmers seem genuinely interested in learning why. For example, my half-day seminar on object-oriented MATLAB at the 2003 IEEE Southeastern Symposium on System Theory was the best-attended session by a wide margin. After that seminar, I started adding more detail to the Class Wizard user's guide. I also improved Class Wizard by adding a guide-based graphical interface and support for object arrays and multiple inheritance. Shortly after that, Mel Belcher and Dale Blair encouraged me to turn the user's guide into a book. I am very grateful for their insight and moral support. I would never have undertaken this project without their initial prodding and enthusiasm.

1 Introduction

The organization of this book breaks MATLAB object-oriented programming into three sections. The first section covers the required elements and focuses on developing a set of functions that give MATLAB objects first-class status within the environment. In the first section, we will develop a group of eight indispensable functions. These functions provide object initialization, a simple intuitive interface, interaction with the environment's features, and array capability. Even more important, the group of eight is responsible for an object-oriented concept called encapsulation. Encapsulation is fundamental to using object-oriented programming as a better, safer alternative to structures. The default functions in MATLAB seem to be at odds with the information-hiding principle of encapsulation; but the group of eight brings MATLAB back under control. By the end of the first section, you will have an excellent working knowledge of MATLAB's object-oriented capability and be able to use object-oriented programming techniques to improve software development.

The second section builds on the first by developing strategies and implementations that allow the construction of hierarchies without compromises. Such hierarchies are important for achieving true object-oriented programming. The concept of building the next layer of functionality on a firm foundation of mature code is very compelling and often elusive. Encapsulation certainly helps, but another object-oriented concept called inheritance makes it much easier to build and traverse an organizational hierarchy. With inheritance, each successive layer simply builds up additional capability without changing code in the foundation. As the code matures, bug fixes simply make the foundation stronger. At first blush, the desire for both first-class status and an inheritance hierarchy appears incompatible. The section on building a hierarchy delivers a harmonious framework.

The third section discusses advanced strategies and introduces some useful utilities. Advanced strategies include, among others, type-based function selection, also known as polymorphism; passing arguments by reference instead of by value; replacing **`feval's`** function handle with an object; and a utility for rapid object-oriented code development. Do not expect to use all the advanced strategies in every software development. Instead, reserve the advanced techniques for difficult situations. Discussing these concepts is important because it opens the door to what are essentially limitless implementation options. It is also nice to know about advanced strategies when the uncommon need arises.

This book makes two assumptions about you, the reader. The first assumption is that you consider yourself an intermediate or better MATLAB programmer. At every opportunity, example code uses vector syntax. The example code also uses a few important but relatively obscure MATLAB functions. Example code also uses language features that some might consider to be advanced topics, for example, function handles and try-catch error handling. Even though code examples are described line by line, entry-level MATLAB programmers might find the example code somewhat vexing.

The second assumes only a cursory knowledge of object-oriented programming. I dedicate a significant amount of the discussion to the introduction of fundamental object-oriented programming concepts. MATLAB programmers new to object-oriented programming will be able to follow these discussions and thus gain the ability to implement object-oriented designs. Even so, there is also plenty of substance to keep seasoned object-oriented programmers on their toes. Going back to the basics will often reveal important design considerations or expose hidden object-oriented capability. It is my sincere hope that everyone reading this book will mutter the phrase "I didn't know you could do that" at least once.

Most of this book concentrates on MATLAB coding techniques; however, the introductory chapter gives me an opportunity to touch on a few topics critical to general software development that are somewhat peripheral to the mechanics of writing code. It also gives me a place to discuss some of the ideas that support object-oriented programming. I trust you are anxious to dive into the world of MATLAB object-oriented programming, so this introduction will be brief.

Some of you might decide to skip this chapter and dive right into the MATLAB implementation. You will be skipping background information on general object-oriented programming: topics like encapsulation, inheritance, and polymorphism. Nothing in this chapter is critical to the examples; however, if you decide to skip this chapter, you might want to come back and read §1.3 before diving into the second section on inheritance. I will remind you to come back when the time comes.

1.1 EXAMPLES

One of the easiest ways to learn is by example. I have tried to include examples of working source code for every new concept or iterative improvement. Each chapter is complete in that the example source code will run and produce results. Subsequent chapters will often add to or improve modules from earlier chapters, but by the end of the chapter everything should execute. You can work along by either typing in the example code or copying the source from the CD. Every chapter has its own directory. All of the examples are included on the CD that accompanies this book.

Interact with the examples. Type in the example source code or copy it from the CD and experiment. The descriptions that accompany the listings will guide you along by supplying command-line instructions. As an alternative to constantly setting MATLAB's path, it is more convenient to experiment with the examples from each chapter's directory. I will include a listing with the explicit **`cd`** command or the result **`pwd`** (print working directory) when it is important to move to a particular directory. That way, you will know where to navigate before typing the commands. The recommended location for the example files is **c:/oop_guide**.* Of course, the **`cd`** directory or **pwd** display will be different if you copy the example files to a different location.

To save a little space, displayed results use compact spacing. MATLAB displays results using a compact format when the **`'FormatSpacing'`** environment variable is set to **`'compact'`**. The following command can be used to set the environment variable.

```
>> set(0, 'FormatSpacing', 'compact')
```

Set **`'FormatSpacing'`** to **`'loose'`** to get back to the default display spacing.

1.2 OBJECT-ORIENTED SOFTWARE DEVELOPMENT

In lieu of a long discussion, I will instead refer you to authors, books, or websites that I have found to be particularly helpful. The referrals are of course not exhaustive because there are too many effective ways to attack software development. The cited references are simply some of the tools I have found to be effective for me. With time and experience, you will accumulate a set of tools that are effective for you. If you do not already have a favorite resource in some particular area, the citations are a good place to begin. I am confident that this book will find a place in your favored set of tools.

---

* The direction of the directory slash will depend on your operating system. In Windows you can navigate directories using either / or \, but **`pwd`** uses \ in its output. In Unix and Linux, only / may be used. In code, the variable **`filesep`** always returns the directory slash appropriate for the operating system; see **`help filesep`**.

1.2.1 At the Top of Your Game

As you are no doubt aware, software development is not just about implementation. Development involves an extensive set of activities that span a wide range of topics, and for any large project, the human element is vitally important. To attack increasingly difficult problems you will need to sharpen your own development ability. Successful software development also draws upon the collective abilities of individuals, teams, and organizations. As problems grow in size you need to be able to focus the development team and help improve the capability of your entire organization. Such continuous professional development at all levels is personally rewarding and directly leads to bigger, better, and faster software. It also leads to more responsibility and improved salaries.

First, recognize that the development of bulletproof software is an exceptionally difficult undertaking. You need to be at the top of your game, and you need to focus and organize your development team. There are a number of proven techniques that can improve both your personal effectiveness and that of your team. These techniques are not limited to coding but span the entire project scope from design through delivery.

Second, recognize that both MATLAB programming and object-oriented programming represent two areas that by themselves rely on a high level of hard-won expertise. Merging the two represents yet another challenge. The MathWorks software engineers did a very commendable job in adding object-oriented capability to MATLAB. Their object model seamlessly meets all of the basic requirements of object-oriented programming; however, this does come with a price. You must write efficient code or run-time performance will suffer. Gaining efficiency requires advanced MATLAB techniques. There are new functions to learn, and familiar functions will be used in entirely new ways. Even fundamental subjects like the function search path get new rules when objects are involved.

The various quirks of MATLAB's object-oriented model can tax the ability of even the most capable designers. MATLAB contains encapsulation and inheritance capability equal to any modern object-oriented language. Sometimes, however, it is difficult to use all of that capability. To clear that hurdle, simply expand and reuse the coding patterns presented in the various examples. The biggest difference between MATLAB and more typical object-oriented languages stems from one of the fundamental properties of MATLAB, untyped variables. The lack of strong variable typing represents a handicap. The rules that govern search-path searching help in some regard. Even so, minor concessions are usually required when implementing a complex object-oriented design in MATLAB. With very weak typing, MATLAB's use of polymorphism is similarly weak. You as the programmer are responsible for choosing correct functionality based on the data. MATLAB's polymorphism usually leads you to a function in the correct class, but the rest is up to you.

1.2.2 Personal Development

Evolving your personal skills is important, but how do you do this effectively? To paraphrase Watts Humphrey of the Software Engineering Institute,

> If you want to get to where you're going, you need a map;
>
> if you don't know where you are, a map won't help.

Following from this statement the general procedure for continuous improvement is not difficult to describe:

- Gauge your level of expertise; find that big, red "You are here" arrow.
- Identify the skills you want to acquire, that is, identify the destination.
- Plot a path from where you are now to where you want to be.
- Periodically check that you are indeed moving toward the destination.

This sounds simple enough, but as always, the devil is in the details.

One good resource where you can learn to sort out the details is a book by Watts Humphrey titled *Introduction to the Personal Software Process*sm.* The Personal Software Process (PSP) sets up an organized approach that allows you to gauge your existing skill set and control the introduction of new skills. By following the PSP's prescriptions, you can improve all phases of your personal development from planning through delivery. The PSP is particularly effective in helping to eliminate the introduction of errors in the critical design and coding stages. Errors eliminated early in the process cannot then affect later stages.

The PSP is a tailored, one-developer version of a software discipline used to improve team-based software engineering. A multiple-developer software discipline can be found in *The Capability Maturity Model* (*CMM*) by Mark Paulk et al.** The *CMM* is not unique in its objective. A large body of research on the introduction of structure and rigor to team-based software development certainly exists. Among the many resources available, the articles found at http://www.sei.cmu.edu are quite extensive and use the same language as that used in the PSP and *CMM*.

Aligned with personal improvement and software engineering rigor is the software development life cycle. Different software products benefit from using different life cycle models, and indeed, there are many different models. Each model supports a relatively unique development environment. The IEEE/EIA 12207 standard***.**** is a concession by both industry and government that no single development model works for every situation. This gives us the liberty to search for models that work well with both our intended applications and MATLAB.

MATLAB programs are successful across a variety of disciplines. The most successful use is when a small group of technical professionals attempts to solve an entirely new problem. This type of development usually contains an evolving set of interlocking constraints. The software is part of the evolution. Designing and writing one iteration increase problem awareness. The discipline involved in developing the software improves understanding and reveals new issues and constraints. Each new revelation folds back into the requirements and begins a new implementation. In the extreme, the revisions never end and it is difficult to complete one revision before discovering new requirements.

There is no definitive stopping rule. Answers to questions like "When is the software model close enough to reality?" or "When is the algorithm accurate enough?" are often difficult to know in advance because each revision uncovers the need for more detail. The software development process itself has become one method of problem discovery. Consequently, each revision extends the capability of the software. After several iterations, the code often evolves completely away from the initial design. We often refer to the result of this constant change as "spaghetti code" because of all the twisted connections among modules. It does not take too much iteration before continued development becomes painfully slow and protracted. Is this a familiar situation?

It turns out that this "typical" MATLAB project description fits the definition of a so-called wicked problem.***** The extreme-programming life cycle model****** is gaining traction as the preferred method for wicked-problem software development. The extreme-programming model is also increasing in popularity for general software development. Since the topic of this book is object-oriented programming, it should come as no surprise that object-oriented programming and the extreme-programming life cycle model are well suited for each other. In fact, certain protections

---

* Addison-Wesley Professional, 1999.

** Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis, principal eds., *The Capability Maturity Model*, Addison-Wesley Professional, 1995.

*** http://standards.ieee.org/reading/ieee/std_public/description/se/12207.0-1996_desc.html.

**** http://www.stsc.hill.af.mil/crosstalk/about.html.

***** P. DeGrace and L. Stahl, *Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*, Yourdon Click, 1990.

****** http://www.extremeprogramming.org.

afforded by object-oriented programming actually enable the extreme-programming life cycle model. There is more to say about wicked problems and extreme programming.

1.2.3 Wicked Problems

We can classify all problems into one of two categories: tame and wicked. Tame problems can be subdued using traditional linear thinking and thus lend themselves to traditional linear development method (e.g., a waterfall model). Wicked problems by contrast are not so easily domesticated. When dealing with wicked problems you need a different approach, and learning to identify them is a good place to begin.

If developers cannot agree on a shared description of the problem, it is probably wicked. Such consensus is difficult because the definition of the problem changes every time a new solution is considered. Individuals on the development team will be at different stages in problem discovery and thus have different opinions about the problem description. Lack of a shared vision often leads to constantly changing requirements, another bane of software development. When developers finally solve the problem, the solution leads to a shared description.

There are many other clues. Some of the most distinctive characteristics often associated with wicked problems are as follows*:

- You cannot understand the problems until you develop solutions, and unfortunately, every solution is expensive and has lasting unintended consequences.
- You find an evolving set of interlocking issues and constraints.
- Proposed solutions are not necessarily right or wrong but rather better or worse.
- There seems to be no definitive stopping rule aside from exhausting the available resources.
- The problem and the proposed solutions are novel or unique.
- The problem does not seem to provide an ultimate test whether the solution is correct or complete.

Anyone with experience in software development can certainly recall a project or two with some of these characteristics. Many of these projects get into trouble not because the wicked problem exists, but rather due to a failure to identify the problem as wicked and approach the solution with the appropriate tools and techniques. Software development has a dismal record where one third of software projects are canceled, and of those that remain half fail to meet the original budget.** The skill and dedication of developers are not at fault in this record. More likely, the whole methodology of approaching the solution of wicked problems is broken.

For example, the knee-jerk approach in dealing with a failing project is to apply more management scrutiny and impose processes that are more stringent. The hope is that a more detailed definition of the requirements, deeper analysis of the problem, in-depth planning, or more progress tracking will get the project back on track. With a so-called tame problem, this approach might actually work. With a wicked problem, this linear approach will almost certainly fail. Wicked problems are very resistant to up-front detailed analysis. The usual approach is failing so we must consider a new set of tools.

The most important part of the new strategy is to accept that wicked problems do indeed exist. After accepting their existence, we need a method of identification. A development team at odds with each other, at odds with management, or at odds with the customer over exactly what the software is supposed to do is a strong indication. A project that continues to spiral downward after

---

* Horst Rittel and Melvin Webber, "Dilemmas in a General Theory of Planning," Reprint no. 86, the Institute of Urban and Regional Development, University of California, Berkeley.

** Mary Poppendieck and Tom Poppendieck, *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003.

adding more resources or swapping out key personnel is also waving a wicked flag. There are other more subtle indications, and a web search on the keywords "wicked problems" will result in a host of resources for both identifying wicked problems and dealing with them.

Accepting the fact that we must begin the solution before we have all the data is important in dealing with wicked problems. Accepting this allows development to focus on revealing more problem detail rather than trying to solve the complete, poorly defined problem. Additional detail refines the problem statement, which folds back into the next solution. Developers are not upset about modifying or scrapping code because neither the goal nor the schedule called for a solution on the first cycle. After several adaptive cycles, developers understand the problem and the software represents a good solution. The solution process for wicked problems concedes that bouncing among design, implementation, and test is the best way to solve poorly understood problems.

This type of iterative development usually runs counter to the current, generally accepted software development practices; however, the future of software development is iterative. This does not mean that software development will revert to the early days of no process maturity or an ill-defined process framework. Developing software in such a stop-and-go manner can result in an unwieldy design unless the iterative development follows a suitable development model. The current set of development processes go hand in hand with a procedural approach. The extendible power of object-oriented programming enables new development models capable of solving wicked problems.

Decades ago there were dire predictions made about the adoption of object-oriented programming. As we now know, most of these damning predictions turned out to be false. Now, the same voices are shouting warnings about iterative development. History appears to be repeating itself. In spite of such dire predictions, companies are obtaining good results using the combined power of object-oriented programming and iterative development.

1.2.4 Extreme Programming

The extreme-programming development model* is one of several models that embody precisely the kind of iterative development necessary to solve wicked problems. In brief, the extreme-programming model emphasizes the following:

- The use of test suites to define project milestones (fanatical testing)
- Frequent releases with small, stable additions to functionality
- A simple design that is iteratively refined
- Continuous code improvement (to make code faster and easier to maintain)
- Pair programming
- Collective code ownership
- Documented standards

The items in this list and object-oriented programming go hand in hand. Frequent releases and continuous code evolution require the use of a language that supports reliable, extendible, reusable code. Object-oriented languages support these goals, and in §1.3.4 we will see how. Items in the list also encourage more of a team-based approach compared to traditional methods. Collective ownership, pair programming, and documented standards make peer review and code walk through integral parts of code development rather than after-the-fact quality assurance steps. Individual effort is still valuable for innovation. The difference here is in bringing the result of individual innovation into the team-based environment.

Perhaps the only valid criticism of iterative methods like extreme programming involves documentation. With very little predevelopment emphasis on requirements and design, developers

* http://www.extremeprogramming.org.

write documentation concurrently as the code is developed or after the code is complete. Neither is ideal. The evolutionary nature of iterative development makes it extremely difficult to document revisions synchronized with code revisions. The community of developers must take collective ownership of the documentation, but supporting tools are not well established. Pushing the development of documentation to the end of the project yields the same poor results regardless of the life cycle model. The descriptions are often lacking in important detail because developers forget many of the nuances. The truth is that software documentation is a tough problem. Even with traditional methods, documentation is often out of date or incomplete. Iterative methods make some problems of documentation different, but the situation overall is neither better nor worse.

You have to relate the importance of documentation to your development team because good documentation relates to productivity. Effort spent analyzing undocumented code is effort that could have gone toward solving the real problem. Multiply this over several developers and many classes, and the consequences become clear. The iterative development community has adopted a posture that says documentation is not required, a posture that might actually have some merit. Instead of separate documentation, the code itself should be self-documenting. Any description other than code is simply a translation, and all translations are subject to error. Under some strict conditions the idea of self-documenting code might actually work. Generally, these conditions are not exclusive to extreme programming but are conditions of good software development in any environment.

Clearly written code is the first condition. Use variable names that represent the data in them and function names that represent the operation. All developers need to be on the same page with respect to the conventions. Community code ownership demands uniformity. Unfortunately, every problem domain seems to use a different vocabulary, making one universal convention impossible to establish. The convention must be somewhat flexible to change just like the code itself. Clearly written code also limits the number of operations carried out on each line. Sometimes run-time performance issues are at odds with such limiting. The 80–20 rule of thumb says that only 20 percent of the code consumes 80 percent of the run time. Surprisingly accurate, this rule allows you to be judicious in trading run time for code complexity. Where code syntax becomes unusually difficult, add a comment to aid in future maintenance. Code idioms and a modular implementation also improve clarity and quality. Document standard conventions and idioms in a coding standard, but allow the standard to evolve.

Taking advantage of MATLAB's help utility is the second condition. Use a **`Contents.m`** file to display a table-of-contents description of all the functions in a directory. Use a standard, compatible format for header comments. Format all the lines in a header as comments, and MATLAB displays the comments in response to **`help`** *function name*. These header comments should summarize the function's intent and cite important assumptions for input–output arguments. In an extreme-programming environment, the header should also include a list of test functions. The first comment line is particularly important because it plays a significant role. Known as the H1 line, MATLAB displays the first header line in response to a **`lookfor`** command.

Up-to-date requirements and at least a high-level design hierarchy form the minimum level of documentation for the third condition. Documented requirements are necessary because these represent the best view of the problem. Use the requirements to scope the problem and drive development in a particular direction. As the development progresses, requirements can and often do change. A formal update of the requirements keeps everyone's expectations on track. A high-level design hierarchy imposes a shared vision.

Align the design with the requirements and allow it to drive iteration goals. Like the requirements, the design hierarchy evolves with the development. In an ideal situation, the hierarchy simply expands its level of detail. Indeed this should be the goal for the design of the public interface. Sometimes entire branches of the hierarchy need reorganization. Allow this reorganization to set the stage for the next cycle of code refactoring. Documented requirements, an up-to-date high-level design, and a standard for self-documenting code are significant improvements over the typical status quo.

Finally, code specifically designed and developed for reuse needs a higher level of documentation. Presumably, the public interface is mature and the behavior is predictable. In short, the code has ceased to evolve so there is little danger of documentation becoming obsolete. Under this scenario, good documentation can improve productivity because even self-documenting code is harder to understand compared to a carefully written, peer-reviewed, cataloged document. With a documented reuse library, we are plainly trying to discourage a developer from redeveloping the same solution.

1.2.5 MATLAB, Object-Oriented Programming, and You

Effectively dealing with MATLAB object-oriented programming means first effectively dealing with MATLAB. The included code examples and idioms rely on an advanced understanding of the MATLAB path, passing data using variable argument lists, and improving run time with vector syntax. Object-oriented techniques also require an expert's knowledge of both standard and obscure MATLAB functions. Object-oriented programming in MATLAB is an advanced topic, and the examples and idioms assume a certain level of MATLAB-language expertise. My goal is to increase your understanding of MATLAB in general, but this book is not a general language reference. The various manuals that come with MATLAB are one of the best general references. Although cryptic at times, they provide a very concise, complete description of almost every language feature. The help facility makes most of the manual information available from the desktop. Online resources at http://www.matlab.com supplement the manuals with up-to-the-minute documentation and user examples. The discussion groups and contributed utilities on the site are particularly valuable.

Programmers include a continuum of MATLAB expertise, but with respect to object-oriented programming, there are two divisions: *client* and *developer*. Client programmers use objects in their own software but do not develop "low-level" object code. Clients are vital to the development in other ways. Clients are important because they often represent the group of domain experts. Their expertise is not in object-oriented programming but rather is steeped in the real problem. As such, clients are an important resource for defining interfaces and functionality. If it were not for clients, developers would be out of a job. Clients, however, are not the target audience of this book.

Developers, on the other hand, are responsible for developing low-level object code. The remaining chapters develop examples, define idioms, and introduce a software tool specifically designed to ease the burden of object-oriented development in MATLAB. As your experience with object-oriented programming increases, you will be called on to both build the object-oriented foundation and use the foundation elements to build applications. The first role represents developer; and the second, client. Clients and developers use different mind-sets. and part of your job as a developer is being able to apply the client mind-set when playing that role.

Playing the role of developer requires a greater attention to detail because you will design both the outward appearance and the inner workings of each object. The outward appearance is important because this is the only part of the object seen by a client. Here, careful thought and attention to detail make the object easy to use. Indeed, this book describes a set of techniques that can be used to give objects an interface identical to that of a structure. A structure-like interface eases a client's use of objects but the structure-like interface is only half the equation. The other half involves the inner workings or private implementation. While the object interface might appear structure-like, your code is actually taking over and producing a result. You have to be diligent in anticipating every condition or the implementation will fail, usually at the worst possible time. Isn't that how Murphy's Law always works? MATLAB's model for object-oriented programming gives you powerful tools to thwart misuse by clients; but as a developer, you must learn how and when to use each tool. Some of these tools are pervasive across all object-oriented languages, while some are unique to MATLAB.

The remaining chapters and examples put you on the right track of becoming a MATLAB object-oriented developer. Same as with the MATLAB language itself, the examples presume a

certain level of expertise in general programming and in object-oriented design. Unlike the treatment of the MATLAB language, objects in the examples remain relatively simple because the implementation methods for simple and complicated objects are essentially the same. There is no reason to cloud the discussion of implementation issues by trying to attack a difficult problem. Of course, this does put limits on how far we will delve into the problem of object-oriented design. As you try to attack increasingly difficult problems, you will undoubtedly need additional object-oriented design resources. A seminal book focusing on object-oriented design is Grady Booch's *Object-Oriented Analysis and Design with Applications**. Booch is one of the early pioneers and has a very intuitive approach to object-oriented design. Two other object-oriented pioneers are James Rumbaugh and Edward Yourdon.

These three object-oriented giants have put aside their differences to develop a graphical design format called the Unified Modeling Language (UML). UML is the standard development and documentation tool for object-oriented programs. The modeling environment provides a very rich and detailed approach, and the basics are easy to learn. The book by Booch et al. titled *The Unified Modeling Language User Guide*** is one of many UML references.

1.3 ATTRIBUTES, BEHAVIOR, OBJECTS, AND CLASSES

Before we try to answer the fundamental question "Why objects?" let's first discuss the difference between an object and a class. The two terms are closely related but are not interchangeable, even though that is how they are often used. In short, a class is a model that exists as lines of code, and an object is an instance of the model that exists in memory during program execution. A class is a user-defined type and an object is a variable of that type.

For tangible objects, we generally accept that they will have both attributes and behaviors. In addition, we usually know how to link attributes and behaviors depending on the object's type. For example, a hungry baby cries and an alarm clock rings. For tangible objects, an object-modeling approach is easy to rationalize because that is how we naturally organize them. In concept, software objects are not much different from tangible objects. Software objects represent tangible elements of the problem domain. Just like worldly objects, software objects have both attributes (data) and behaviors (functions). In a good design, these attributes and behaviors associate naturally and are inseparable from one another. Perform some thought exercises centered on this idea.

What image enters your mind at the mention of the word "shape"? Is it two-dimensional or three? What is its color? Are the sides straight or curved? If you describe your image, do you think I would agree that it is indeed a shape? It can be square, circular, or star shaped; red, blue, or rainbow colored; stationary, rotating, or zipping about; and it would still be a shape. From experience, we are able to abstract the idea of shape into a general collection of attributes and behaviors. In object-oriented terms, the abstraction is a *class* and any particular shape is an *object* of that class. This particular abstraction is easy because we practice it without even realizing. With practice and experience, abstraction into an object-oriented software design is almost as easy.

1.3.1 From MATLAB Heavyweight to Object-Oriented Thinker

Until fairly recently universities taught most engineers, scientists, mathematicians, and technical professionals to decompose a problem into a series of actions. Converting these actions into a loosely organized set of functions yields a so-called procedural-based design. The procedural-based approach spawned a variety of other software-engineering techniques. Software development life

* Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings, 1991. The 3rd edition was released in 2004.

** Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley Professional, 1998.

cycles are the most notable. In too many cases, the customer's project-planning tools assumed a so-called waterfall life cycle model. Project planning is much easier with a waterfall model.

Unfortunately, the procedural approach and the waterfall life cycle are showing their age. The amount of module-to-module coupling hinders the ability to maintain or extend many large programs. Adding a new feature or fixing an old one takes longer than expected and, far too often, introduces side effects unrelated to the new feature. The use of object-oriented methods can drastically reduce the amount of module-to-module coupling. Many in the software-engineering community believe that shifting to an object-oriented approach is the only way to achieve significant increases in program size and complexity.

The ready availability of commercial MATLAB toolboxes has allowed large increases in complexity even with the use of traditional, procedural methods. Invariably with time, software requirements will grow to the point where even the use of toolboxes will not be enough to offset the limitations of the procedural approach. No one can predict when the typical program size will outstrip the capacity of the current approach; however, some MATLAB projects have already crossed the threshold. Many MATLAB programmers recognize the early-warning signs. If we follow the lead of our software-engineering brethren, embracing object-oriented techniques appears to be the solution. Helping defend this position is the fact that MATLAB includes a very robust object model.

Where would the study of mathematics be without whole, real, and complex numbers? Biology would be equally difficult without taxonomy divisions among plants, animals, fungi, virus, protozoa, and bacteria. In these disciplines, properties rather than behavior drive the decompositions. Object-oriented programming is no different. User-defined types are the central focus of the software architecture. Just like other taxonomies, the types contain both properties and behavior but the decomposition emphasizes the properties. For someone steeped in procedural decomposition, the object-oriented approach appears backward. Instead of focusing on behavior (functions), object-oriented programming focuses on attributes (data). Along with this change in focus come big differences in life cycles, coding development, testing, and integration.

To many, object-oriented development represents a radically different way of thinking. Introducing changes of this scale into an organization can be difficult and protracted. By one estimate, the transition takes an average programmer about one year.* This book should help speed the transition by defining specific coding practices and by exposing potential problem areas. The Class Wizard tool also allows programmers to focus on design rather than implementation (see Chapter 18), further speeding the transition. Other techniques may also hasten the transition. For example, pair programming is a type of co-mentoring activity that should be helpful in shortening the transition time. There are also many more books, seminars, and short courses available today compared to 1994 when the estimate was made.

1.3.2 Object-Oriented Design

Think about shapes again. If asked to design a software representation of a shape, how would you begin? You might have a good idea about shapes but you still need to find out if your ideas match the needs of your clients. You can use client requirements, user stories, and domain experts to help pin down the set of attributes and behaviors required of your software shape. At first these attributes and behaviors might seem disconnected; however, with more analysis, patterns and dependencies usually emerge. First, arrange shapes with similar attributes in a loose taxonomy. Then use behavior differences to infer additional attributes. For example, it might be perfectly reasonable to combine a division between moving and stationary shapes by defining a speed attribute. This gives all shapes the same behavior; however, shapes with zero speed do not appear to move. It might also be perfectly reasonable to keep moving shapes separate from stationary ones. In that case, a moving

* B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

shape is still a shape but it has at least one additional attribute and behavior. The choice affects the software design and code, but the client's experience with the final design is the same. When the taxonomy stops changing, we establish the software architecture. Each leaf in the taxonomy represents a set of attributes that can be implemented as a class. Connections among leaves allow classes higher in the taxonomy to serve as the foundation for lower classes. Lower classes do not redeclare higher-level attributes because they can inherit the higher-level attributes by simply declaring a connection in the taxonomy. The same organization works for behaviors.

The process is similar in many respects to procedural design except that the final organization focuses on data rather than function. In theory, the process sounds reasonable, but in reality, some software problems are maddeningly difficult to organize. Sometimes developers do not have enough experience in the problem area to foster good organization. At other times, the special terms and notation used by the experts simply overwhelm the designer. Object-oriented designers have experienced these difficulties and have developed many techniques useful in difficult design environments. Unfortunately, a full treatment of object-oriented design is outside the scope of this book. If you are new to object-oriented programming, you will gain valuable experience by implementing and evolving someone else's design. When you are ready to design your own object-oriented architecture, a library of books and a wealth of articles and websites are available that fully develop object-oriented design. The authors and references already cited represent good starting points.

1.3.3 Why Use Objects?

Previously, I made the statement that the creation of objects seems to mirror the way we naturally view the world. A brief discussion about shapes was used to demonstrate the idea. If true, the idea that software development can reflect our typical worldview is nice but it certainly would not compel programmers to abandon their current practice. This is particularly true in light of the amount of effort involved in making a change. No, the argument has to be a lot more compelling.

The area of software development most influenced by object-oriented programming is software quality. Demonstrated quality improvements can make converts of even the most grizzled procedural programmers. Quality has many facets, but bug-free software that works correctly the first time it is used is a typical goal. It is hard to disagree that bug-free software somehow equates to high-quality software; however, if bug-free code takes too long to develop or runs too slowly, what then of quality?

In reality software quality is an elusive topic with a lot of "I'll know it when I see it" judgments. Running correctly without crashing is certainly one aspect of quality, but other areas are important too. Assuming the requirements correctly identify what is needed, software engineers generally agree that overall quality is influenced by the following:

- Reliability
- Reusability
- Extendibility

Specific features in object-oriented programming relate to every one of these factors. Another possible factor is productivity. Perhaps it would be better to emphasize productivity rather than quality. After all, we know that bug-free software is impossible to produce. Even if we could get all the bugs out, delivery times would be very long and the production cost would be astronomical. Besides, customers have learned to expect bugs, particularly in the first few versions.

I hope you were *not* nodding in agreement with the last few sentences. These often accepted assumptions are *wrong*, *wrong*, *wrong*. The fact that your competitors believe them gives you an enormous competitive advantage. Proven techniques can both reduce the number of coding errors and hasten the discovery of bugs that do manage to slip in. The introduction of fewer errors along with quicker discovery increases productivity by reducing the amount of unproductive time spent

reworking broken code. With a lower error rate, testing reveals fewer bugs, thus allowing the entire development to run at a faster pace. In the manufacturing sector, Lean-Six-Sigma* techniques dramatically improve both quality *and* productivity. Proven false in the manufacturing sector is the notion that high quality equals low productivity. In fact, attaining both exceptional quality and high productivity can be the rule rather than the exception. There is nothing to prevent the introduction of Lean-Six-Sigma ideas into the software development process.

Customers can also be retrained. Once you start delivering high-quality products, the marketplace will demand the same quality from all producers. The rise of the Japanese auto industry provides a clear example where a customer's appreciation for quality disrupted the marketplace. I predict that the same disruption will eventually occur in the software industry. Currently, India seems to be the likely winner, but China too is coming on strong. I urge you to consider the implications and work to drive your organization toward the delivery of world-class quality. Sooner than you imagine, customers will be demanding it.

1.3.4 A Quality Focus

Proven techniques can enhance software quality. Some techniques focus on one particular quality measure like reuse. Others cut across all measures. Object-oriented techniques belong in the latter group because they create a fundamentally different development environment. It is an environment with a proven ability to improve all areas of quality. Below we summarize the major factors contributing to quality.

1.3.4.1 Reliability

The most visible aspect of software quality is reliability. If the software crashes or produces the wrong result, customers consider the product unreliable. Even when most features work reliably, it follows from Murphy's Law that the one unreliable feature will be the most important to the customer. Contrary to opinion, highly reliable software is not impossible or prohibitively expensive to develop. Consider the selected observations about the state of general software development published in 2001**:

- Half the modules are defect free.
- Disciplined personal practices can reduce the initial defect rates by up to 75 percent.
- Avoidable rework constitutes 40 to 50 percent of the total effort on most software projects.
- It costs 50 percent more per line of code to develop high-dependability software…. However, the initial investment reduces overall cost if the project involves significant operations and maintenance costs.

The fact that on average half the modules are defect free provides strong evidence that it is possible to write defect-free software. Anything that can increase the defect-free percentage will have an enormous impact, and the second observation promises a huge improvement. Reducing initial defect rates by 75 percent means the typical rate of five defective lines out of ten improves to about one in ten. Extending the same improvement to well-implemented modular code means that close to 90 percent of the modules will be error free the first time a developer releases the code for test. At a minimum, this implies fewer trips between test and rework, but the implications on productivity are much deeper.

Examine the effect on resources. Spending 50 percent of your time on rework means that every four hours of programming require, on the average, another four hours to find and fix defects — defects that were *avoidable*. If four hours is the average debug time, how wide is the span around

* Michael L. George, *Lean Six Sigma: Combining Six Sigma Quality with Lean Speed*, McGraw-Hill, 2002.

** Barry Boehm and Victor R. Basili, "Software Defect Reduction Top 10 List," *IEEE Computer*, January 2001, 135–17.