Practical Statecharts in C/C++

Quantum Programming for Embedded Systems

- Model reactive systems with UML statecharts
- Efficiently code statecharts directly in C/C++
- Rapidly build embedded software with statechart-based frameworks

Miro Samek, Ph.D.
Practical Statecharts in C/C++

Quantum Programming for Embedded Systems

Miro Samek, Ph.D.

CMPBooks
San Francisco, CA • New York, NY • Lawrence, KS
# Table of Contents

Preface ........................................................................................................ vii

## PART I  STATECHARTS ................................................................. 1

**Chapter 1  Whirlwind Tour of Quantum Programming .......... 3**
  1.1  The Ultimate Hook — Anatomy of a GUI Application ........ 4
  1.2  A Better Way of Programming — A Calculator That Works .... 6
  1.3  Object-Oriented Analogy .......................................................... 16
  1.4  Quantum Analogy .................................................................. 19
  1.5  Summary .............................................................................. 20

**Chapter 2  A Crash Course in Statecharts ......................... 23**
  2.1  The Essence of Finite State Machines ................................. 24
  2.2  The Essence of UML Statecharts ......................................... 30
  2.3  Examples of State Models ..................................................... 44
  2.4  Summary .............................................................................. 52

**Chapter 3  Standard State Machine Implementations .......... 55**
  3.1  State Machine Interface ....................................................... 56
  3.2  Nested switch Statement ...................................................... 57
  3.3  State Table .......................................................................... 60
  3.4  State Design Pattern ............................................................ 65
  3.5  Optimal FSM Implementation .............................................. 69
# Table of Contents

3.6 State Machines and C++ Exception Handling ................. 73
3.7 Role of Pointer-to-Member Functions .......................... 73
3.8 Implementing Guards, Junctions, and Choice Points .......... 76
3.9 Implementing Entry and Exit Actions .......................... 76
3.10 Dealing with State Hierarchy .................................. 77
3.11 Summary ....................................................... 79

**Chapter 4 Implementing Behavioral Inheritance ............. 81**
4.1 Structure ....................................................... 83
4.2 An Annotated Example ........................................ 94
4.3 Heuristics and Idioms .......................................... 101
4.4 The Event Processor ............................................ 106
4.5 C Implementation .............................................. 120
4.6 Caveats ......................................................... 126
4.7 Summary ......................................................... 128

**Chapter 5 State Patterns .............................. 131**
5.1 Ultimate Hook ................................................... 133
5.2 Reminder ........................................................ 138
5.3 Deferred Event ................................................ 144
5.4 Orthogonal Component ......................................... 149
5.5 Transition to History ........................................... 160
5.6 Summary ......................................................... 164

**Chapter 6 Inheriting State Models .................... 167**
6.1 Statechart Refinement Example in C++ ....................... 168
6.2 Statechart Refinement Example in C ........................ 177
6.3 Caveats ......................................................... 180
6.4 Summary ......................................................... 185
PART II QUANTUM FRAMEWORK .......................... 187

Chapter 7 Introducing the Quantum Framework ........... 189
7.1 Conventional Approach to Multithreading ................. 191
7.2 Computing Model of the QF ............................. 197
7.3 Roles of the QF ........................................ 206
7.4 Summary ............................................. 212

Chapter 8 Design of the Quantum Framework ............ 215
8.1 Embedded Real-Time Systems ............................ 216
8.2 Handling Errors and Exceptional Conditions ............. 218
8.3 Memory Management ................................... 225
8.4 Mutual Exclusion and Blocking ......................... 230
8.5 Passing Events ....................................... 235
8.6 Active Objects ........................................ 248
8.7 Initialization and Cleanup ............................... 253
8.8 Time Management .................................... 255
8.9 QF API Quick Reference ............................... 258
8.10 Summary ............................................ 263

Chapter 9 Implementations of the Quantum Framework .... 265
9.1 The QF as a Parnas Family ............................... 266
9.2 Code Organization .................................... 267
9.3 Common Elements ..................................... 272
9.4 DOS: The QF without a Multitasking Kernel .............. 283
9.5 Win32: The QF on the Desktop ......................... 290
9.6 RTKernel-32: The QF with a Preemptive Priority-Based Kernel .... 295
9.7 Summary ............................................ 302

Chapter 10 Sample Quantum Framework Application ...... 305
10.1 Generating a QF Application ......................... 306
10.2 Rules for Developing QF Applications .................... 315
10.3 Heuristics for Developing QF Applications ............... 317
10.4 Sizing Event Queues and Event Pools .................... 318
10.5 System Integration .................................... 323
10.6 Summary ............................................ 323
# Table of Contents

**Chapter 11 Conclusion** ................................................................. 325
  11.1 Key Elements of QP .......................................................... 326
  11.2 Propositions of QP ............................................................ 329
  11.3 An Invitation ...................................................................... 333

**Appendix A “C+” — Object-Oriented Programming in C** ........ 335
  A.1 Abstraction ................................................................. 336
  A.2 Inheritance ................................................................. 339
  A.3 Polymorphism ............................................................... 341
  A.4 Costs and Overheads .................................................... 349
  A.5 Summary ................................................................. 350

**Appendix B Guide to Notation** .................................................. 353
  B.1 Class Diagrams ............................................................ 353
  B.2 Statechart Diagrams ..................................................... 356
  B.3 Sequence Diagrams ....................................................... 357
  B.4 Timing Diagrams .......................................................... 357

**Appendix C CD-ROM** ................................................................. 359
  C.1 Source Code Structure .................................................. 361
  C.2 Installation ................................................................. 361
  C.3 Answers to the Exercises .............................................. 362
  C.4 Resources ................................................................. 362

**Bibliography** ........................................................................... 365

**Index** ..................................................................................... 371

**What’s on the CD?** ................................................................. 400
Preface

What we do not understand we do not possess.
— Goethe

Almost two decades ago, David Harel invented statecharts as a powerful way to describe complex reactive (event-driven) systems [Harel 87]. Subsequently, statecharts have gained almost universal acceptance as a superior formalism and have been adopted as components of many software methodologies, most notably as part of the Unified Modeling Language (UML). Nevertheless, the use of statecharts in everyday programming has grown slowly. Among the many reasons, the most important is that statecharts have always been taught as the use of a particular tool, rather than the way of design.

This heavy reliance on tools has affected the software community in three ways. First, the aggressive marketing rhetoric of tool vendors has set unrealistically high expectations for purely visual programming and fully automatic code generation. Second, the rhetoric of the argument for automatic code synthesis depreciated the role of manual coding. Finally, the accidental association between statecharts and CASE (computer-aided software engineering) tools gave rise to a misconception that the more advanced UML concepts, such as statecharts, are only viable in conjunction with sophisticated code-synthesizing tools.

The reality is that CASE tools haven’t made manual coding go away. Even the best-in-class code-synthesizing tools can generate only a fraction of the software (the so-called housekeeping code [Douglass 99]). The difficult, application-specific code
still must be written explicitly (although it is typically entered through the dialog boxes of a tool rather than typed into a programming editor). This also means that the models are not purely visual, but a mixture of diagrams and textual information (mostly snippets of code in a concrete programming language).

Moreover, for many projects, a design automation tool is not the best solution. The fundamental problem, as always, is the cost versus the return. Even if you ignore the dollar cost of the tool, you must ask whether the benefits outweigh the compounded complexity of the problem and the tool. The complete cost function must also include training and adaptation of the existing infrastructure to the tool (e.g., the compiler/linker/debugger tool chain, the host and target operating systems, the directory structure and file names, version control, and the software build process). After weighing all the pros and cons and struggling with a tool for a while, many teams notice that they spend more time fighting the tool than solving problems. For many developers, the tool simply can’t pull its own weight and ends up as shelfware or a not-so-simple drawing tool.

**Mission**

My primary mission in this book is to offer a simple, lightweight alternative to a design automation tool by providing concrete, efficient, and proven implementations of statecharts that every practitioner reasonably proficient in C or C++ can start using within days.

To achieve these goals, I describe the major components of every typical code-synthesizing tool.

- The techniques needed to implement and use UML statecharts — the main constructive element in the UML specification (presented in Part I).
- A real-time application framework — a complete software infrastructure for executing statecharts, tailored to embedded real-time systems and based on active objects and asynchronous event passing (presented in Part II).

At first glance, the approach can be classified as a set of common elaborative techniques for implementing UML models. Even as such, it spares many practitioners from reinventing the wheel. In this book, I present ready-to-use, generic, and efficient elements that you can use to implement and execute hierarchical state machines; generate, queue, and dispatch events; integrate state machines with real-time operating systems (RTOSs); and much more. These software elements vary little from system to system but are hard to find in the literature. It’s even harder to make them work well together. The value of this book is similar to that of a multivitamin pill: in one fell swoop (or a few chapters in this case), you get all the necessary ingredients, well balanced and complementing each other. If you use this book only in this manner, my most important goal is already accomplished.
Why Quantum Programming?

By providing concrete implementations of such fundamental concepts as statecharts and statechart-based computing models, the book lays the groundwork for a new programming paradigm, which I propose to call Quantum Programming (QP). I chose this name to emphasize the striking and fundamental analogy between reactive software systems and microscopic objects. As the laws of quantum mechanics describe, at the fundamental level, most microscopic objects (such as elementary particles, nuclei, atoms, and molecules) exhibit state behavior. Quantum objects are, in fact, little state machines, which spend their lives in strictly defined, discrete quantum states and can change state only in certain ways via uninterruptible transitions known as quantum leaps. Correspondingly, QP models state transitions with run-to-completion (RTC) steps. The only way quantum systems interact with one another is through an exchange of field quanta (intermediate vector bosons), which are mediators of fundamental forces.1 Similarly, QP requires reactive systems to interact only by exchanging event instances. I explain more about this quantum analogy in Chapters 1, 2, and 7.

As a programming paradigm, QP has much more to offer than merely the snippets of code published in this book. I see and use QP as a set of techniques that increases the level of abstraction of a conventional programming language (such as C or C++). The additional abstractions in QP allow me to efficiently model reactive systems directly in C++ or C. The role of QP can be compared to that of an object-oriented (OO) programming language. Just as Smalltalk, C++, or Java enable object-oriented programming (OOP) through direct support for the three fundamental OO design meta-patterns — abstraction, inheritance, and polymorphism — QP enables statechart modeling directly in C or C++ through another fundamental meta-pattern: the hierarchical state machine (HSM). Currently, the fundamental HSM pattern is an external add-on to C++ or C, but there is no reason it couldn’t be natively supported by a quantum programming language in the same way that abstraction, inheritance, and polymorphism are natively supported by OO programming languages. (Indeed, in Appendix A, you see that OOP can be supported as an “add–on” in procedural languages such as C by explicitly applying the three fundamental OO patterns. I subsequently use this augmented C, which I call “C+”, to develop the C implementation of the HSM pattern.)

The relationship between QP and OOP is interesting. On one hand, the most important aspects of QP, such as the HSM design pattern and the asynchronous communication paradigm, are orthogonal to OOP, which is an indication that these aspects of QP might be fundamental. On the other hand, however, these concepts work best when applied with OOP. In fact, a deep analogy exists between OOP and

---

1. For example, photons mediate the electromagnetic force, gluons the strong force, and bosons $W^+$ and $Z^0$ the weak interactions.
QP, which I discuss in Chapter 1. In this sense, QP builds on top of OOP, complements it, and extends it into the domain of reactive systems.

Most of the concepts that form QP are not new, but, rather, draw on a broad range of long-known techniques and methodologies from programming and other disciplines such as quantum field theory. Most inspiring to me was the real-time object-oriented modeling (ROOM) method described by Bran Selic and colleagues [Selic+ 94]. Specifically, the real-time framework, or Quantum Framework, first began as a radically simplified ROOM virtual machine. Other influences were the classical Gang of Four design patterns [Gamma+ 95], the UML specification [OMG 01] (especially the state machine package), the original works of Harel [Harel 87, Harel+ 98], and books by Bruce Douglass [Douglas 99, Douglas 99a]. The Douglass writings in many ways are on the opposite end of the spectrum from QP, because QP offers mostly alternative views and complementary techniques to those he describes. For example, the state patterns he pioneered rely heavily on orthogonal regions, whereas QP shows how to implement some of these more elegantly using state hierarchy (Chapter 5).

For over four years, I have been using and refining QP in real-life projects — for example, in hard real-time implementations of GPS receivers. I am excited and thrilled by the potential of this approach to the point that I wrote this book so I could share QP with the largest audience I can reach. However, I am realistic — I do not see QP as a silver bullet. QP does not promise you the royal road, as do some design automation tools; however, it offers arguably the fastest road to better designed, safer, and more efficient event-driven software, because nothing stands between you and the solution. When you start using QP, you’ll find, as I did, that your problems change. You no longer wrestle with convoluted if or switch statements; rather, you spend more time thinking about how to apply state patterns and how to partition your problem into active objects. Nonetheless, even with QP (or any CASE tool, for that matter), programming reactive systems remains difficult because it is by nature difficult. As Frederick Brooks [Brooks 87] notes in his classic essay “No Silver Bullet,” you can only attack the accidental difficulties and can’t do much about the essential ones, at least not in software alone. In this context, QP exactly follows Brooks’ advice — to attack and remove the accidental difficulties associated with developing event-driven software.

**QP versus XP and Other Agile Methodologies**

QP is not a programming methodology; it is a set of concrete techniques for modeling and implementing reactive systems. Nevertheless, the QP approach is an expression of a basic programming philosophy, which is closely aligned with the recent trends in software development known collectively as light or agile methodologies. Some of these technologies include eXtreme Programming (XP), Crystal methodologies, SCRUM, Adaptive Software Development, Feature-Driven Development, and
Agile Modeling. The basic philosophy behind the new approaches is best summarized in the “Agile Manifesto” [Fowler 01], in which the “seventeen anarchists” agree to value (1) individuals and interactions over processes and tools and (2) working software over comprehensive documentation.

In the context of QP, valuing individuals and interactions over processes and tools means putting emphasis on understanding the underlying implementations and mechanisms rather than on hiding the complexity behind a tool (the practice that Anders Hejlsberg [Hejlsberg 01] denounced as “simplicity–complexity wrapped in something simple”). Real-life experience has shown repeatedly that if an individual understands the underlying implementation model, then he or she can code more efficiently and work with greater confidence. For example, determining which actions fire in which sequence in a nontrivial state transition is not something you should guess at or discover by running a tool-supported animation of your statechart. The answer should come from your understanding of the underlying implementation model (discussed in Chapters 3 and 4). Even if you decide to use a design automation tool and even if your particular tool uses slightly different statechart implementation techniques than those I discuss in this book, you will still be a better, more productive, and confident user of the tool because of your understanding of the fundamental mechanisms.

In addition to putting value on individuals and interactions by explaining low-level fundamental software patterns, QP also offers powerful high-level metaphors, such as the quantum-mechanical and object-oriented analogies. A metaphor is valuable because it promotes the conceptual integrity of a software product and provides a common vocabulary, which dramatically improves communication among all of the stakeholders. Agile methodologies recognize the importance of such metaphors (e.g., XP proposes the development of a metaphor as a key practice).

As an elaborative approach, QP values working software over comprehensive documentation. In fact, QP offers nothing but the working code. I have made every attempt to provide only executable code, so that you can try out virtually every listing and code snippet you find in this book, as well as the code available only on the accompanying CD-ROM. Because only executable code is testable, this aspect of QP goes hand-in-hand with the requirement for continuous testing, which is inherent to all agile methodologies.

In addition to offering techniques for creating executable code, QP also offers highly readable, self-documenting code. For example in Chapter 4, I give directions on how to make the structure of a statechart clearly apparent from the code and almost equivalent to a UML state diagram. This is not to say that QP abandons UML diagrams or makes them obsolete. To the contrary, in this book, you will see quite a few diagrams that follow UML notation strictly (although because I used a simple drawing tool, they cannot be called UML-compliant). When it comes to diagrams
and other visual models, QP shares the commonsense view of Agile Modeling [Ambler 01]. The most important role of visual models is to help you think through the designs and communicate them to programmers, customers, or management. For that purpose, simple tools like paper and pencil, whiteboard, or sticky notes are usually sufficient. It is also OK to discard the visual models after they have fulfilled their purpose. The specific value of visual modeling lies in tapping the potential of high bandwidth spatial intelligence, as opposed to lexical intelligence used with textual information.

Incomplete, disposable visual models, however, can’t be used for code synthesis. In this respect, the agile approach fails to take advantage of the constructive aspect of some visual representations, such as UML statecharts. QP complements agile methodologies by enabling high-level modeling directly at the code level. With the concrete, ready-to-use building blocks provided by QP, you can construct, compile, and execute concurrent state models rapidly, even if they are nothing more than vastly incomplete skeletons. As you will see in Chapter 4, you can change the state machine topology (e.g., add, remove, or rearrange states and transitions) at any stage, even late in the process, by changing a few lines of code and recompiling. Then you can test your executable model on the host or target environments. Plenty of such executable models are included throughout this book. In that way, you can quickly try out many alternatives before committing to any one of them. This process is rightly called modeling, rather than coding, because your goal isn’t the generation of a final product or even a prototype, but rather the fast exploration of your design space.

Admittedly with such lightweight modeling, you lose the benefits of spatial intelligence. As mentioned earlier, modeling at the code level does not preclude using UML diagrams or low-fidelity sticky notes as models of user interfaces. Indeed, spatial intelligence is best at grasping high-level structures and patterns when the models are relatively high level and uncluttered. As the models become more detailed, lexical intelligence usually takes over anyway because, in the end, “programming is all about text” [Hejlsberg 01].

**Audience**

This book is intended for the following computer professionals interested in reactive, or event-driven, systems.

- Embedded programmers and consultants will find practical advice, explanations, and plenty of code that they can use as is or modify to build event-driven software.
- Real-time systems designers will find a lightweight alternative to heavyweight CASE tools for modeling real-time systems. The Quantum Framework, combined
with a preemptive RTOS, can provide deterministic behavior and can be embedded in commercial products.

- Users of design automation tools will better understand the inner workings of their tools, helping them to use the tools more efficiently and confidently.

- GUI developers, interactive Web page designers, and computer game programmers using C or C++ will find nontrivial, working examples of how to code and integrate UML statecharts with GUI environments such as the Microsoft Windows API.

- Hardware designers exploring the extension of C or C++ with class libraries to model SoC (System on Chip) designs will find one of the most succinct and efficient implementations of hierarchical state machines.

- Graduate-level students of Computer Science or Electrical Engineering will learn many design patterns that are backed up by numerous examples and exercises.

This book is about extending object-oriented techniques to programming reactive systems in C++ and C. I assume that you are familiar with fundamental object-oriented concepts and are reasonably proficient in one of these two languages. To benefit from Part II of the book, you should be familiar with fundamental real-time concepts. I am not assuming that you have prior knowledge of the UML specification in general or statecharts in particular, and I introduce these concepts in a crash course in Chapter 2.

**Guide to Readers**

This book has two main parts. In Part I (Chapters 1–6), I describe state machines — what they are, how to implement them, and the standard ways or patterns of using them. This part is generally applicable to any event-driven system, such as user interfaces (graphical and otherwise), real-time systems (e.g., computer games), or embedded systems. In Part II (Chapters 7–11), I describe the Quantum Framework, which is a software architecture designed specifically for embedded real-time systems.

Surveys of programmers\(^2\) consistently indicate that C and C++ clearly dominate embedded systems programming. The vast majority (some 80 percent) of embedded systems developers use C; about 40 percent occasionally use C++. Consequently, every practical book genuinely intended to help embedded systems programmers should focus on C and C++. For that reason, I consistently present two complete sets of code: C++ and C. The C++ implementations are typically more succinct and natural because the underlying designs are fundamentally object oriented. In Appendix A,

\(^2\) For example, the Embedded Systems Programming survey (published annually by *ESP* magazine) or the Annual Salary Survey (published by *SD* magazine).
I present a description of “C+” — a set of three design patterns (abstraction, inheritance, and polymorphism), which I've used to code object-oriented designs in portable ANSI C. The “C+” patterns appear in many nontrivial examples throughout the book contrasted side-by-side with C++ implementations of the same designs. If you are interested only in C++, you can skip the C implementations on the first reading. However, I found that understanding the underlying mechanisms of implementing object orientation vastly improved my OO designs and allowed me to code them with greater confidence. For that reason, you might want to study the C code, concentrating primarily on the “C+” specifics. Conversely, if you are interested only in C, you should still read the explanations pertaining to C++ code. Often, I don’t repeat clarifications for design decisions because they are the same for C++ and C. As a C programmer, you should have no problems understanding my C++ implementations because I use only very straightforward inheritance and hardly any polymorphism (except in Chapter 6).

My goal is not only to give you fish (i.e., the source code) but to teach you how to fish (i.e., to model reactive systems). Unfortunately, if you want to learn to fish, you should be ready to sweat a little. I try to provide you with executable implementations whenever possible. I believe that nothing builds more confidence in a new technique than actually executing a piece of example code. Sometimes, I ask you to step through a few instructions with a debugger; other times, I suggest that you make alterations to the code and rerun it. In the end, however, it is up to you to actually do it.

Most of the chapters in this book contain exercises. The exercises are intermixed with the text, rather than grouped at the end of a chapter, to put them closer to the relevant text. Most of the time, the exercises are not intended to test your comprehension of the chapter, but rather to suggest an alternative solution or to introduce a new concept. I provide a complete set of answers in order to pass on as much information as possible. If you usually skip over exercises, at least consider looking at the answers provided on the CD-ROM so that you don’t miss the guidelines or techniques I introduce there.

I describe the CD-ROM that accompanies this book in more detail in Appendix C and the HTML browser included on the disc. Here, I want to mention that the examples (although written in portable C++ or C) are designed to run under Microsoft Visual C++ v6.0 on a 32-bit Windows machine (9x/NT/2000). If you don’t have Visual C++, I recommend you get a copy so that you will be able to run the examples without redoing the makefiles, libraries, and so on. It is also important to have a good, easy-to-use debugger.

The source code for this book is available on the CD-ROM and can be freely distributed to students by accredited colleges and universities without a license. You can use the code as is or modify it to embed in your products, but you must obtain a
Source Code Distribution License to distribute QP source code. I may choose to assess a license fee for such situations, and you need to contact me for pricing (see below).

I intend to continue advancing QP and am interested in any constructive feedback you may have. I have opened a Web site devoted to promotion of this book and QP at the URL http://www.quantum-leaps.com. I plan for this site to contain application notes, ports of QP to different platforms, state patterns, useful links, bug fixes, frequently asked questions and much more. Please feel free to contact me via e-mail at miro@quantum-leaps.com.

Acknowledgments

I would like to thank Robert Ward, my acquisitions editor at CMP Books, in whom I found an excellent editor, extremely knowledgeable software professional, and a resonating mind. I feel very lucky that Robert accepted my book for publication and offered his guidance throughout this book’s lengthy birthing. I couldn’t have dreamed of a better editor for this book.

The team of CMP Books has been wonderful. It was a great pleasure to have Michelle O’Neal as the managing editor. I’d like to thank Julie McNamee, Catherine Janzen, and Rita Sooby, for putting up with my writing. I’m amazed how many times they managed to turn my convoluted technical jargon into something actually readable. Reviewing their corrections was for me the best possible lesson in lucid writing. I also thank Justin Fulmer, for polishing my drawings and integrating them into the final version of the book.

I am indebted to Jeff Claar, for the technical review of the manuscript and for scrutinizing the accompanying code. Jeff’s contributions include the complete C++ state machine implementation compliant with multiple inheritance, which is available on the companion CD-ROM.

I’d like to thank Michael Barr, for letting me publish my earlier article on the subject in the Embedded Systems Programming magazine, and for reviewing an early copy of this book.

This book has had a long gestation, of which the actual writing was merely the final step. I owe a lot to my former colleagues at GE Medical Systems, specifically to John Zhang, for infecting me with his enthusiasm for design patterns and state machines. In addition, I’d like to acknowledge my current team at IntegriNautics Corporation—one of the highest concentrations of Ph.D.s per square foot in Silicon Valley. I am particularly grateful to Paul Montgomery, for brainstorming many of the ideas, unforgettable pair-programming sessions, and for patiently enduring all my early iterations of the Quantum Framework.
I wholeheartedly thank my parents, who from over 7,000 miles away were still able to provide plenty of support and encouragement. The perseverance they instilled in me was critical for completion of this project. Quite specially, I thank my sister Barbara, for her faith in me and for always taking care of her “little brother”.

Most of all, however, I would like to thank my wife Kinga and our lovely daughter Marie for tolerating far too many nights and weekends, which I spent in front of a computer rather than with them. Without their love, help, and continuous encouragement this book could never be finished. I love you so much!

Miro Samek
Palo Alto, California
April 2002
State machines are a superb formalism for specifying and implementing event-driven systems that must react to incoming events in a timely fashion. The UML statecharts represent the current state of the art in state machine theory and notation.

Part I of this book introduces the concept of statecharts, describes concrete techniques of coding statecharts directly in C and C++, and presents a small catalogue of basic statechart-based design patterns. You will learn that statecharts are a powerful way of design that you can use even without the assistance of sophisticated code-synthesizing tools.
Chapter 1

Whirlwind Tour of Quantum Programming

I have found out there ain’t no surer way to find out whether you like people or hate them than to travel with them.
— Tom Sawyer Abroad [Mark Twain]

The triumph of the graphical user interface has been one of the most impressive developments in software during the past three decades.¹ Today the concept is so familiar as to need no description. Although from the beginning, windows, icons, menus, and pointing have been intuitive and easy to grasp for users, they remain a challenge for programmers. The internal GUI architecture baffles many newcomers, who often find it strange, backwards, mind-boggling, or weird. GUI programming is different because unlike traditional data processing, it is entirely event-driven. Events

¹ The concept of the windows, icons, menus, and pointing (WIMP) interface was first publicly displayed by Doug Englebart and his team from the Stanford Research Institute at the Western Joint Computer Conference in 1968 [Englebart+ 68].
can occur at any time in any order. The application always must be prepared to handle them. GUI is an example of a complex reactive system.

You don't need to look far to find other examples of reactive systems. In fact, CPUs of all PCs, Macs, and other general-purpose computers consume only about 1 percent of the worldwide microprocessor production. The other 99 percent of microprocessor chips sold every year end up in various embedded systems, which are predominantly reactive in nature. Yet, in spite of this ubiquity, the code found in most of these systems is notoriously difficult to understand, fix, and maintain. Theoretical foundations on how to construct such software have been around for more than a decade; however, these ideas somehow could not make it into the mainstream. Quantum Programming (QP) is an attempt to make the modern methods more approachable for programmers. QP is a set of straightforward design patterns, idioms, concrete implementations, and commonsense techniques that you can start using immediately without investing in sophisticated tools.

1.1 The Ultimate Hook — Anatomy of a GUI Application

The early GUI designers faced a formidable task. On the one hand, a GUI application must be virtually infinitely customizable to allow anything from nonrectangular windows to splash screens and dazzling screen savers. On the other hand, the system ought to impose a consistent look and feel, and applications content with this standard behavior should be simple. How would you reconcile such conflicting requirements?

Today the problem seems easy — the trick is to use the “Ultimate Hook” [Petzold 96]. The idea is brilliantly simple. The GUI system (e.g., Windows) dispatches all events first to the application (Windows calls a specific function inside the application). If not handled by the application, the events flow back to the system. This establishes a hierarchical order of event processing. The application, which is conceptually at a lower level of hierarchy, has a chance to react to every event; thus, the application can customize every aspect of its behavior. At the same time, all unhandled events flow back to the higher level (i.e., to the system), where they are processed according to the standard look and feel. This is an example of programming-by-difference because the application programmer has to code only the differences from standard system behavior.

Independently, David Harel applied the same idea to the finite state machine formalism [Harel 87]. Around 1983 he invented statecharts as a powerful way of specifying complex reactive systems. The main innovation of statecharts over classical finite state machines was the introduction of hierarchical states. To understand what it means, consider the relation between the nested state (substate) and the surrounding state (superstate) depicted in Figure 1.1a. This statechart attempts to process any
event, first, in the context of the substate. If the substate cannot handle it, the statechart automatically passes the event to the next higher level (i.e., to the superstate). Of course, states can nest deeper than one level. The simple rule of processing events applies then recursively to an arbitrary level of nesting.

Harel’s semantics of state hierarchy is at the heart of the Ultimate Hook design underlying GUI systems; in other words, the statechart supports the Ultimate Hook pattern directly. This becomes obvious when renaming the superstate to GUI_system and the substate to GUI_application, as shown in Figure 1.1b. Now this is an interesting result: The fundamental Ultimate Hook design pattern turns out to be a very simple statechart! This powerful statechart is unusually simple because, at this level of abstraction, it does not contain any state transitions.

Traditionally, the hierarchy of states introduced in statecharts has been justified as follows.

As it turns out, highly complex behavior cannot be easily described by simple, “flat” state-transition diagrams. The reason is rooted in the unmanageable multitude of states, which may result in an unstructured and chaotic state-transition diagram — [Harel+ 98].

Certainly, hierarchical diagrams are often simpler and better structured than traditional flat diagrams. However, this is not the fundamental reason for the significance of state hierarchy, merely one of the side effects. State hierarchy is fundamentally important even without the multitude of states and transitions, as demonstrated clearly by the GUI example. The powerful statechart shown in Figure 1.1b contains only two states and not a single state transition; yet, it is powerful. The only essential feature is state hierarchy, in its pure form.

Figure 1.1b is so unusually simple because it shows only the highest level of abstraction. All nontrivial GUI applications have many modes of operation (states) with typically complex rules of switching between these modes (state transitions), regardless of whether you use a statechart, a classical flat state transition diagram, brute force, or any other method. However, designs based on the statechart formalism seem to be the most succinct, robust, and elegant, if for no other reason than their direct support for programming-by-difference (Ultimate Hook pattern).
Although statecharts in some form or another have gained almost universal acceptance as a superior formalism for specifying complex reactive systems, their actual adoption into mainstream programming has been slow. In particular, GUI designs traditionally have not used statecharts. You will not find them in standard GUI architectures such as Microsoft Foundation Classes (MFC), Borland’s Object Windows Library (OWL), or the more recent Java Abstract Window Toolkit (AWT). You also will not find support for statecharts in rapid application development (RAD) tools such as Microsoft Visual Basic or Borland Delphi. Among the many reasons, the most important is that statecharts have been taught as a high-level, visual language, mandating the use of sophisticated computer-aided software engineering (CASE) tools, rather than as a type of design. This has created many misconceptions in the industry and has resulted in a lack of practical advice on how to code statecharts in mainstream programming languages such as C or C++.

1.2 A Better Way of Programming — A Calculator That Works

Coding a statechart directly in C or C++ is not that hard. This section shows you how. I decided to include this example early in the text so you could start experimenting with it as soon as possible. The example comes from the book Constructing the User Interface with Statecharts by Ian Horrocks [Horrocks 99]. The author presents a desktop calculator application distributed with Microsoft Visual Basic. He first identifies a number of problems with the original implementation and then proposes an alternative statechart design. Here, I will pick up where he left off by actually implementing his statechart in C++ and integrating it with the Windows GUI (Figure 1.2). Although this section concentrates on a C++ implementation, the accompanying CD-ROM also contains the equivalent version in C.

1.2.1 Shortcomings of the Traditional Event–Action Paradigm

Before getting into the implementation; however, I’ll examine some problems that Ian Horrocks found in the Visual Basic Calculator because they turn out to be emblematic of inconsistencies in handling modal behavior. Most of the time, the
calculator correctly adds, subtracts, multiplies, and divides. However, in certain cases, the application provides misleading results, freezes, or crashes altogether.

**Exercise 1.1** After familiarizing yourself with the contents of the accompanying CD-ROM (see Appendix C and the HTML browser on the disc) find the Visual Basic Calculator example and launch it. Try the following sequence of operations and watch the application crash: 1, /, −, =, 2, =. Try the sequence 2, ×, CE, 2, =, and observe that Cancel Entry had no effect, even though it appeared to cancel the ‘2’ entry from the display. Try different ways of breaking the calculator or of producing misleading results.

The Visual Basic Calculator often has problems dealing with negative numbers. This is because the same button (−) is used to negate a number and to enter the subtraction operator. The correct interpretation of the ‘−’ event, therefore, depends on the context, or mode, in which it occurs. Likewise, Cancel Entry (CE button) occasionally works erroneously. Again, Cancel Entry makes sense only in a particular context, namely, just after the user has entered a number or operator. As it turns out, the calculator tries to handle it in other contexts as well. At this point, you probably have noticed an emerging pattern. Just look for events that require different handling depending on the context, and you can break the calculator in many more ways.

The faults just outlined are rooted in the standard bottom-up implementation of this application. The context (state) of the computation is represented ambiguously as a group of flags and variables, so it is difficult to tell precisely in which mode the application is at any given time. There is no notion of any single mode of operation, but rather tightly coupled and overlapping conditions of operation. For example, the calculator uses DecimalFlag to indicate that a decimal point has been entered,
OpFlag to represent a pending operation, LastInput to indicate the type of the last key press event, NumOps to denote the number of operands, and several more state variables. With this representation, determining whether the ‘–’ key should be treated as negation or subtraction requires the following conditional logic (in Visual Basic).³

```
Select Case NumOps
  Case 0
    If Operator(Index).Caption = "-" And LastInput <> "NEG" Then
      ReadOut = "-" & ReadOut
      LastInput = "NEG"
    End If
  Case 1
    Op1 = ReadOut
    If Operator(Index).Caption = "-" And LastInput <> "NUMS" And
       OpFlag <> "=" Then
      ReadOut = "-"
      LastInput = "NEG"
    End If
  ...  
```

Such an approach is fertile ground for bugs for at least two reasons: examining the current mode requires evaluating a complex expression, and switching between different modes requires modifying many variables, which can easily lead to inconsistencies. Expressions like these, scattered throughout the code, are not only unnecessarily complex but expensive to evaluate at run time. They are also notoriously difficult to get right, even by experienced programmers, as the bugs still lurking in the Visual Basic Calculator attest.

### 1.2.2 Calculator Statechart

The good news is that there is a better way of approaching reactive systems. Statecharts, like the one depicted in Figure 1.3, eliminate the aforementioned problems by design. Arriving at this statechart was definitely not trivial, and you shouldn’t worry if you don’t fully understand it at the first reading (I wouldn’t either). At this point, my goal is just to introduce you to the statechart approach and convince you that it isn’t particularly hard to code in C or C++.⁴ I want to walk you quickly through the main points without slowing you down with full-blown detail. I promise to return to

---

³ The complete Visual Basic source code for the calculator application is available on the accompanying CD-ROM.

⁴ Here, I discuss a C++ implementation. However, the implementation in plain C is available on the accompanying CD-ROM.
this statechart on more than one occasion in the following chapters, for a closer study of statechart design, and to discuss concrete implementation techniques later.

The calculator statechart from Figure 1.3 contains 15 states5 (the topmost Windows system state is not included) and handles 11 distinct events: IDC_0, IDC_1_9, 

---

5. At first, you might be disappointed that the statechart for such a simple calculator is so complicated. After analyzing the problem, I feel that the diagram in Figure 1.3 represents the complexity of the problem just about right. Section 2.3.1 in Chapter 2 explains in more detail the reasons for complexity in this case.
IDC_POINT, IDC_C, IDC_CE, IDC_PLUS, IDC_MINUS, IDC_MULT, IDC_DIVIDE, IDC_EQUALS, and IDCANCEL.

Exercise 1.2  Find and launch the Quantum Calculator application (try both the C++ and C implementations). Try to understand the behavior of the application by comparing it with the statechart from Figure 1.3 (you might find the current state display handy). Try to break it or produce misleading results. Test the Quantum Calculator to see how it handles the ‘–’ and ‘+’ unary operators.

This state machine takes advantage of hierarchy in several places. For example, Cancel (the ‘C’ button) is handled explicitly only in the highest level state, calc (look for the arrow labeled “C” at the top of Figure 1.3). This event triggers a self-transition in the calc state. You can understand how the statechart handles the Cancel event based solely on the Ultimate Hook semantics of state nesting introduced earlier. Assume, for example, that when the user clicks the ‘C’ button, the active state is op Entered. This state doesn’t “know” how to handle the Cancel event, so it automatically passes this event for processing to the next higher level state, that is, to calc. The calc state knows how to handle the Cancel event by executing the aforementioned self-transition. This causes exit from calc followed by entry, first to calc, then ready, and finally begin, by the recursive execution of initial transitions. At this point, the calculator ends processing of the Cancel event and waits for the next event.

To restate: The statechart started in op Entered and ended in begin. Actually, the statechart could have been in any of the other 11 substates of calc (refer to Exercise 1.3) and would still end up in begin. The classical flat state machine would require specifying each of the 11 transitions explicitly. The statechart allows reusing one transition 11 times. The gain is not only the drastic reduction in the sheer number of transitions but also a more accurate representation of the problem at hand. There is only one Cancel transition in the calculator problem. A natural language specification might read as follows: Whatever state the calculator is in at the time the user clicks the ‘C’ button, the calculator should clear its display and other internal registers and become ready for another computation. The statechart represents this specification faithfully, whereas the classical flat state machine would add repetitions and artificial complexity.

Exercise 1.3  Not all 15 substates of calc can be active. For example states ready, operand1, and operand2 can never become active. Explain why.
The following implementation of the calculator statechart is straightforward because all the state machine functionality is inherited from the \texttt{QHsm} (quantum hierarchical state machine) class. Listing 1.1 shows the C++ declaration of the calculator statechart.

**Listing 1.1** Declaration of the calculator statechart; the unusual indentation of state handler methods (lines 14–29) indicates state nesting

```cpp
#include <windows.h>
#include "qf_win32.h"  // include the Quantum Framework (QF)

struct CalcEvt : public QEvent {
    int keyId;  // ID of the key depressed
};

class Calc : public QHsm {  // calculator Hierarchical State Machine
public:
    Calc() : QHsm((QPseudoState)initial) {}  // calculator Hierarchical State Machine
    static Calc *instance();  // Singleton accessor method
private:
    void initial(QEvent const *e);  // initial pseudostate-handler
    QSTATE calc(QEvent const *e);  // state-handler
    QSTATE ready(QEvent const *e);  // state-handler
    QSTATE result(QEvent const *e);  // state-handler
    QSTATE begin(QEvent const *e);  // state-handler
    QSTATE negated1(QEvent const *e);  // state-handler
    QSTATE operand1(QEvent const *e);  // state-handler
    QSTATE zero1(QEvent const *e);  // state-handler
    QSTATE int1(QEvent const *e);  // state-handler
    QSTATE frac1(QEvent const *e);  // state-handler
    QSTATE opEntered(QEvent const *e);  // state-handler
    QSTATE negated2(QEvent const *e);  // state-handler
    QSTATE operand2(QEvent const *e);  // state-handler
    QSTATE zero2(QEvent const *e);  // state-handler
    QSTATE int2(QEvent const *e);  // state-handler
    QSTATE frac2(QEvent const *e);  // state-handler
    QSTATE final(QEvent const *e);  // state-handler
private:
    void clear();  // action methods...
    void insert(int keyId);
    void negate();
    void eval();
    void dispState(char const *s);
private:  // data attributes
    HWND myHwnd;
    char myDisplay[40];
```

```
Events dispatched to the calculator are represented as instances of the CalcEvt class (Listing 1.1, lines 4–6). This class derives from QEvent and adds the keyId event parameter, which represents the ID of the key entered. As mentioned before, the calculator hierarchical state machine Calc declared in lines 8 through 46 derives from QHsm. The Calc class contains several attributes that keep track of the computation (the attributes constitute the memory of the state machine). Please note, however, that the attributes are not used to determine the state of the application. Rather, the QHsm superclass keeps track of the active state, which is crisply defined at all times. In fact, the calculator GUI displays it for you, so that you can easily correlate calculator behavior with the underlying statechart from Figure 1.3. You also can recognize all states declared as state handler methods in lines 14 through 29. The unusual use of indentation indicates state nesting.

### 1.2.3 Integration with Windows

For simplicity, this example uses the raw Win32 API rather than a higher level wrapper like MFC. The calculator GUI is a dialog box, so it declares friendship with the corresponding Windows dialog procedure (Listing 1.1, lines 44–45). Because Windows is an event-driven (reactive) system, it already provides a complete environment within which a state machine can execute and needs only minor customizations for this particular application. The main Windows procedure, WinMain(), performs only basic initializations and then invokes the dialog procedure.

```c
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, PSTR cmdLine, int iCmdShow)
{
    InitCommonControls();                       // load common controls library
    locHinst = hInst;                                  // store instance handle
    DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG), NULL, calcDlg);
    return 0;                       // exit application when the dialog returns
}
```

The dialog procedure (Listing 1.2) starts the state machine (by invoking the init() method in response to the WM_INIT_DIALOG Windows message), translates the Windows events to calculator events, and dispatches the events for processing (by invoking the dispatch() method) in response to the WM_COMMAND Windows message.
Listing 1.2 Initializing and dispatching events to the Quantum Calculator statechart from a dialog procedure

```c
static HINSTANCE locHinst;                                  // this instance
static HWND locHwnd;                                         // window handle

BOOL CALLBACK calcDlg(HWND hwnd, UINT iMsg,
                      WPARAM wParam, LPARAM lParam)
{
    CalcEvt e;
    switch (iMsg) {
        case WM_INITDIALOG:
            Calc::instance()->myHwnd = locHwnd = hwnd;
            SendMessage(hwnd, WM_SETICON, (WPARAM)TRUE,
                         (LPARAM)LoadIcon(locHinst, MAKEINTRESOURCE(IDI_QP)));
            Calc::instance()->init();                  // take the initial transition
            Calc::instance()->isHandled = TRUE;
            Calc::instance()->dispatch(&e);                      // take one RTC step
            return TRUE;
        case WM_COMMAND:
            switch (e.keyId = LOWORD(wParam)) {
                case IDCANCEL:
                    e.sig = TERMINATE;
                    break;
                case IDC_1:
                case IDC_2:
                case IDC_3:
                case IDC_4:
                case IDC_5:
                case IDC_6:
                case IDC_7:
                case IDC_8:
                case IDC_9:
                    e.sig = IDC_1_9;
                    break;
                case IDC_PLUS:
                case IDC_MINUS:
                case IDC_MULT:
                case IDC_DIVIDE:
                    e.sig = IDC_OPER;
                    break;
                default:
                    e.sig = e.keyId;
                    break;
            }
            Calc::instance()->isHandled = TRUE;
            Calc::instance()->dispatch(&e);                   // take one RTC step
            return Calc::instance()->isHandled;
    }
    return FALSE;
}
```
1.2.4 State Handler Methods

State handler methods perform the actual work of the application. As members of the `Calc` class, state handlers have direct access to all the attributes. A state handler takes a pointer to an immutable event instance (`QEvent const *`) and returns either 0, if it handled the event, or the superstate (more precisely, the pointer to the superstate handler method), if not.

Listing 1.3 contains handlers for states `calc`, `ready`, and `begin` (you can refer to the accompanying CD-ROM [Appendix C] for the code of the other state handlers). The structure of all state handlers is similar: They all start with an identical `switch` statement (with an event signal `e->sig` used as the discriminator) and all end in the same way by returning the superstate (i.e., a pointer to superstate handler method).

For example, state `begin` returns `ready`, `ready` returns `calc`, and `calc` returns `top` (the ultimate superstate that contains the entire state machine). The body of the `switch` statement contains all signals that the corresponding state handles, coded as separate `case` statements. For example, the `begin` state handles signals `Q_ENTRY_SIG` and `IDC_OPER` (Listing 1.3 lines 41–50). Most `case` statements end with `return 0` to indicate that the state handler processed this event. State transitions are coded using the `Q_TRAN()` (quantum transition) macro. For example, event `IDC_OPER` in state `ready` triggers a transition to state `opEntered`, so you code it as (line 33) `Q_TRAN(&Calc::opEntered).

Listing 1.3 State handlers for `calc`, `ready`, and `begin` states. Boldface indicates housekeeping code (see the last paragraph of this section)

```cpp
QSTATE Calc::calc(QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: dispState("calc"); return 0;
        case Q_INIT_SIG: clear(); Q_INIT(&Calc::ready); return 0;
        case IDC_C: clear(); Q_TRAN(&Calc::calc); return 0;
        case TERMINATE: Q_TRAN(&Calc::final); return 0;
        } if (e->sig >= Q_USER_SIG) {
            isHandled = FALSE;
        }
        return (QSTATE)&Calc::top;
    }

QSTATE Calc::ready(QEvent const *e) {
    switch (e->sig) {
        case Q_ENTRY_SIG: dispState("ready"); return 0;
        case Q_INIT_SIG: Q_INIT(&Calc::begin); return 0;
        case IDC_0: clear(); Q_TRAN(&Calc::zero); return 0;
        case IDC_1_9: clear();
    }
```
As you can see, the housekeeping code \(^6\) (i.e., state machine declaration and state handler skeletons, indicated in boldface in Listing 1.3) you need to write to translate a statechart to C++ is almost trivial. In fact, it is not more complicated than the code you need to write to translate a Unified Modeling Language (UML) class diagram to C++ (i.e., class declaration and method skeletons). You probably don’t even think that you translate a class diagram to C++; you simply code an object-oriented system directly in C++. This is so because C++ provides the right (object-oriented) level of abstraction. The practical techniques for implementing statecharts raise the level of

---

6. Douglass [Douglass 99] uses the term “housekeeping code” to denote sections of code that are used repetitively to represent common constructs such as states, transitions, events, and so on. Some CASE tools can automatically generate such representation invariants from statechart diagrams.
abstraction further (to the “quantum” level) and, in the same sense, enable direct modeling of reactive systems in C++.

### 1.3 Object-Oriented Analogy

I hope you experienced déjà vu when you read about programming-by-difference and reuse of behavior in the context of statecharts. Haven’t you encountered similar concepts before? Doesn’t a state hierarchy resemble an object-oriented taxonomy of classes? As this section explains, hierarchical states don’t simply resemble classes of objects in object-oriented programming (OOP); the analogy is deep and fundamental. Such a close analogy has many practical implications.

#### 1.3.1 State Hierarchy and Class Taxonomy

One of the cornerstones of OOP is the concept of class inheritance, which allows you to define new classes of objects in terms of existing classes, and consequently enables you to construct hierarchically layered taxonomies of classes. A hierarchy of states introduces another type of inheritance, which is equally fundamental. I will call it behavioral inheritance [Samek+ 00].

To understand how state hierarchy leads to inheritance and how it works, consider again the statechart depicted in Figure 1.1a. This time, however, suppose that the substate is completely empty, with no internal structure (no transitions and no reactions). If such a state becomes active, it will automatically forward all events to the superstate. Therefore, the behavior of such a substate will be externally indistinguishable from the superstate — the empty substate inherits the exact behavior from its superstate. This is analogous to an empty subclass, which does not declare any methods or attributes. An instance of such a subclass is, in every respect, equivalent to an instance of its superclass. The child class is indistinguishable from the parent class because everything is inherited exactly.

Although checking the corner case of exact inheritance is instructive, inspecting ways in which nested states can differ from their ancestors is more interesting. As class inheritance allows subclasses to “adapt” to new environments, behavioral inheritance allows substates to “mutate” by adding new behavior or by overriding existing behavior. Nested states can add new behavior by adding new state transitions or reactions for events that are not recognized by surrounding states. This corresponds to adding new methods to a subclass. Alternatively, a substate may also process the same events as the surrounding states but will do it in a different way. Thus, the substate can override the inherited behavior, which corresponds to a subclass overriding a method defined by its parents, which leads to polymorphism.

In a typical class taxonomy, classes lower in the hierarchy are more specialized than their ancestors; conversely, classes higher in the hierarchy are generalizations of...
their descendants. The same holds true in state hierarchies. For example, consider a hypothetical “failed” state that turns on an alarm bell upon entry (as part of its entry action) and turns it off upon exit (as part of its exit action). If this state has a substate, say “unsafe,” and this substate becomes active, the alarm bell will ring because being in the unsafe state also means being in the failed state. If the system is in the unsafe state, it also is in the failed state and, recursively, is in every ancestor state of failed. The is in (is-in-a-state) generalization of states corresponds to the is a (is-a-kind-of) generalization of classes.

1.3.2 Entering/Exiting States and Instantiating/Finalizing Classes

In the previous example, the entry action executed automatically upon entry to a state (turning on the alarm bell), and the exit action (turning off the alarm bell) executed automatically upon exit from the state. These actions are analogous to class constructors and destructors. Instantiation of a class is very much like entering a state. Conversely, class finalization is like exiting a state. In both cases, special actions are invoked in a predetermined order: Constructors are invoked starting from the most remote ancestor class (destructors are invoked in reverse order). Entry actions are invoked starting from the topmost superstate (exit actions are invoked in reverse order).

1.3.3 Programming-by-Difference

Class inheritance is commonly used for programming-by-difference. This programming style is the essence of reuse: A subclass needs to define only the differences from its superclass and otherwise can reuse (share) implementation defined in the superclass.

Behavioral inheritance is identical in this respect. A substate needs to define only the differences from its superstate and can otherwise reuse the behavior defined in the superstate. In other words, supporting programming-by-difference behavioral inheritance enables reuse of behavior.

1.3.4 Behavioral Inheritance as a Fundamental Meta-Pattern

OOP can be viewed as a consistent use of three fundamental concepts — abstraction, inheritance, and polymorphism — that are actually meta-patterns because they provide the underpinnings for all other object-oriented (OO) design patterns [Gamma+ 95]. QP introduces and implements another, equally fundamental, meta-pattern: behavioral inheritance. The meta-pattern is truly enabling because it raises the level of abstraction to allow direct modeling of complex state behavior in C or C++ in the same way that fundamental OO meta-patterns (natively supported in OO languages) enable direct OO modeling in C++, Smalltalk, or Java.
As you will see in Chapter 4, the implementation of the behavioral inheritance meta-pattern often uses the object analogy and borrows many solutions from the C++ object model implementation. Again, this is a direct application of the OO analogy.

### 1.3.5 State Patterns

As Gamma and colleagues [Gamma+ 95] write: “One thing expert designers know not to do is solve every problem from the first principles.” The maturity of object technology shows through the emergence of OO design patterns that capture, name, and catalog proven OO designs. In analogy, state patterns, which are concerned with useful ways of structuring states rather than objects, have begun to appear [Douglass 99], reflecting the increasing maturity of statechart technology.

By providing a concrete implementation for the behavioral inheritance meta-pattern, QP enables a much more precise description of state patterns than the traditional graphical statechart notation alone, in the form of concrete, executable code. In this respect, QP acts much like an OO programming language, which also captures OO patterns in the form of concrete executable code. In both cases, bubbles and arrows of graphical representation, although very helpful, are not sufficient to capture all the details necessary to understand and successfully apply a pattern. Bertrand Meyer summarized eloquently the shortcomings of graphical-only descriptions when he said [Meyer 97a]:

> the good thing about bubbles and arrows, as opposed to programs, is that they never crash.

State patterns in QP revolve predominantly around the central concept of behavioral inheritance, rather than the orthogonal component. Therefore, they represent alternatives to the solutions presented elsewhere (e.g., Douglass [Douglass 99]). Chapter 5 presents a minicatalog of quantum state patterns.

### 1.3.6 Refactoring State Models

Another aspect in which state models and OO models are similar is their evolution during the software life cycle. Both state hierarchies and class hierarchies undergo similar development phases, and both, at some point of their life cycle, need restructuring to continue to evolve.

The main objective of software restructuring, or refactoring (see e.g., [Opdyke 92], [Fowler+ 99]), is not to change how the software behaves — indeed, the changes should be transparent to black-box testing. The goal of refactoring is rather to actively counteract the natural increase in the degree of chaos (architectural decay)
that gradually renders any software system prohibitively expensive to maintain and modify.

Because of the similarities between behavioral inheritance and class inheritance, the same general refactorings are applicable both to OO systems and to statecharts.

- **Refactoring to generalize** — creating a common superclass—creating a common superstate
- **Refactoring to specialize** — deriving subclasses from a common base—nesting substates in a common superstate

In addition, like OO design patterns, state patterns capture many structures that result from refactoring state models. Using these patterns early in the life of a statechart design can prevent later refactorings. Alternatively, when restructuring becomes inevitable, state patterns can provide convenient targets for your refactorings.

### 1.3.7 Beyond Object-Oriented Programming

Recent years have seen several attempts to extend and augment traditional OOP. Trends that have gained particular attention are components, patterns, and frameworks. Software components are capable of encapsulating complete business functions and therefore are usually at a higher level of granularity than objects. OO design patterns try to capture and reuse proven patterns of collaboration among whole groups of objects. At a higher level still are frameworks, which are entire, albeit incomplete, applications. The common themes of all these developments are ways of combining many fine-granularity objects into systems. All these trends are examples of programming-in-the-large.

QP, based on behavioral inheritance, takes the opposite route. The traditional OO method stops short at the boundary of a class, leaving the internal implementation of individual class methods to mostly procedural techniques. Behavioral inheritance and the OO analogy allow many OO methods to be extended and applied inside classes.

### 1.4 Quantum Analogy

To help you understand how Quantum Programming fits with other trends, it is helpful to compare software developments to modern physics. Traditionally, OOP would correspond to classical mechanics: beautifully able to describe everyday experience but unable to accurately describe either very large or very small scale phenomena. Components, patterns, and frameworks try to expand the macroscale frontier. You could compare them to thermodynamics or general relativity pertinent to large-scale, complex objects like galaxies or black holes. In this picture, QP would correspond to quantum mechanics, because it expands the microscale frontier.
As described by the laws of quantum theory, microscopic objects have the following two most characteristic properties.

- Quantum objects spend their lives in strictly defined quantum states and can change their state only by means of uninterruptible transitions known as quantum leaps. Because of various symmetries, the quantum states are naturally hierarchical (degnerate in quantum terminology).

- Quantum systems cannot interact with one another directly; rather, every interaction proceeds via an intermediate artifact (intermediate boson). The various intermediate bosons are mediators of fundamental forces (e.g., photons mediate the electromagnetic force, gluons the strong force, and bosons W± and Z° the weak forces).

QP follows the quantum model quite faithfully. Part I of this book corresponds to the first characteristics of quantum systems — their discrete, statelike behavior. Part II, on the other hand, covers the second aspect of the quantum analogy — the interactions. The fundamental units of decomposition in QP are concurrently active hierarchical state machines (active objects). These software machines can interact with one another only by asynchronous exchange of various event instances.

1.5 Summary

This chapter provided a quick tour of Quantum Programming. QP is concerned with reactive systems, which are systems that continuously interact with their environment by means of exchanging events. Over the years, several techniques have evolved that can be used to design and implement such systems. One of the most powerful ideas has proved to be the concept of hierarchical event processing, which GUI programmers know as the Ultimate Hook pattern. Almost two decades ago, David Harel generalized this concept and combined it with finite state machines to create the formalism known as statecharts. Although statecharts have gained almost universal acceptance in software methodologies and modeling languages, like UML, their adoption into everyday programming has been slow. The main reason is the widespread misunderstanding that statecharts are only usable when supported by sophisticated CASE tools. The result is a lack of practical advice on how to efficiently hand-code statecharts in mainstream programming languages such as C or C++. However, as you saw in the Quantum Calculator example, you can easily implement the fundamental concepts of statecharts directly in C++ by applying the behavioral inheritance meta-pattern. This pattern is central to QP, just as abstraction, inheritance, and polymorphism are patterns central to OOP.

The analogy between QP and OOP goes deeper. They are both unified around the concept of inheritance. Just as class inheritance is a cornerstone of OOP, behavioral
inheritance is a cornerstone of QP. This analogy allows almost direct application of many OO techniques to state models, such as programming-by-difference, the construction of proper state taxonomies, the application of similar refactorings, or the use of exit and entry actions. In addition, the implementation of the behavioral inheritance meta-pattern shares many commonalities with the internal implementation of the C++ object model, which you can view as a native realization of the three fundamental OO design patterns: abstraction, inheritance, and polymorphism.

QP, like OOP, introduces its own (quantum) state patterns. These patterns are concerned with useful ways of structuring statecharts to solve recurring problems. QP, like an OO programming language, allows more precise descriptions of the patterns than can be achieved with graphical-only representation.

QP goes beyond traditional OOP by modeling the internal structure of reactive classes. The governing laws in this microcosm turn out to be similar to those of quantum physics, where objects spend their lives in discrete states, make uninterruptible state transitions (quantum leaps), and interact only by exchanging event instances (intermediate virtual bosons).
Chapter 11

Conclusion

*I would advise students to pay more attention to the fundamental ideas rather than the latest technology. The technology will be out-of-date before they graduate. Fundamental ideas never get out of date.*

— David Parnas

For many years, I have been looking for a book or a magazine article that describes a truly practical and reasonably flexible way of coding statecharts in a mainstream programming language such as C or C++. I have never found such a technique.

I believe that this book is the first to provide what has been missing so far — a flexible, efficient, portable, maintainable, and truly practical implementation of statecharts that takes full advantage of behavioral inheritance. This book is perhaps also the first to offer complete C and C++ code for a highly portable statechart-based framework for the rapid development of embedded, real-time applications.

My vision for this book, however, goes further than an explanation of the code. By providing concrete implementations of fundamental concepts, such as behavioral

---

1. I have never been satisfied with the techniques that require explicit coding of transition chains (see Chapter 3) because it leads to inflexible, hard-to-maintain code and practically defeats the purpose of using statecharts in the first place.
inheritance and active object–based computing, the book lays the groundwork for a new programming paradigm, which I call Quantum Programming (QP).

This last chapter summarizes the key elements of QP, how it relates to other trends in programming, and what impact I think it might have in the future.

### 11.1 Key Elements of QP

In the Preface, I defined QP as the programming paradigm based on two fundamental concepts: (1) hierarchical state machines and (2) an active object–based computing model. Although independent in principle, these two ideas work best together. You can realize these concepts in many ways; QP is one of them. Other examples include the ROOM method (considered independent of the ObjecTime toolset) and virtually every design automation tool for developing event-driven software.

What sets QP apart is its minimalist, code-centric, and low-level nature. This characterization is not pejorative; it simply means that QP maps the fundamental concepts directly to the source code, without intermediate layers of graphical representations. QP clearly separates essentials from niceties by implementing the former directly and supporting the latter only as design patterns. Keeping the implementation small and simple has real benefits. Programmers can learn and deploy QP quickly without large investments in tools and training. They also can adapt and customize the Quantum Framework (QF) easily to their particular situation, including to severely resource-constrained environments. They can understand, and indeed regularly use, all the features.

#### 11.1.1 A Type of Design, Not a Tool

The most important point of QP is that the hierarchical state machine (as any other profound concept in software) is a powerful type of design, not a particular tool. The issue here is not a tool — the issue is understanding.

Code-synthesizing tools can have heft and substance, but they cannot replace a conceptual understanding. For over a decade, various authors, in writing about state-charts, have been asserting that the days of manual coding are gone and that state-charts open a new era of automatic programming supported by visual tools. However, with such an era of truly widespread automatic code synthesis still nowhere near in sight, you are left today with no information on how to code state-charts practically. Worse, you cannot access the accumulated knowledge about state-charts because most of the designs exist only on paper, in the form of incomplete state diagrams or, at best, as high-level models accessible only through specific

---

2. That is, programmers still need to learn the concepts. There is no way around that. However, they can skip learning a tool.
tools. This diffusion of information is unfortunate because instead of propagating a true understanding of the technique, the tool-selling rhetoric creates misconceptions in the software community and makes statecharts, as a type of design, inaccessible to the majority of software practitioners.

The goals of QP are to dispel the various misunderstandings and make statecharts more accessible to programmers. Although tools can help generate code from state diagrams, they are not essential to take full advantage of the most fundamental statechart features. Indeed, it is relatively simple to code statecharts directly in C or C++ and to organize them into fully functional applications founded on a statechart-based application framework (the QF).

11.1.2 A Modeling Aid

Many software methodologists lament that programmers suffer from the rush-to-code syndrome: a pervasive urge to crank out code instead of analyzing, designing, modeling, documenting, and doing the other things that should precede and accompany coding. This syndrome is not necessarily evil. Typically, it reflects the natural and healthy instinct of programmers who want to engage in concrete development instead of producing artifacts whose usefulness they mistrust. Therefore, rather than fighting this instinct, QP helps jump-start the development process by rapidly building high-level, executable models.4 Such models allow you to perform analysis and design by quickly exploring the problem space; yet, because the models are code, no conflict exists with the rush-to-code syndrome.

QP supports rapid model building in several ways.

1. It lets you work at a high level of abstraction directly with hierarchical state machines, active objects, and events.

2. It has been designed from the ground up so that you can compile and correctly execute intentionally incomplete prototypes successfully. For example, the publish–subscribe event delivery of the QF does not require that you specify the recipients of events, so a prototype still compiles, even if some active objects (recipients of events) are missing. Similarly, automatic event recycling allows the correct execution of applications (without memory leaks), even if some published events are never received.

3. It lets you elaborate statecharts in layers of abstraction; that is, you can intentionally leave the internal structure of composite states unspecified.

---

3. As described in Section 2.2.9 in Chapter 2, state diagrams are incomplete without a large amount of textual information that details the actions and guards.

4. Such models correspond roughly to spike solutions in eXtreme Programming (XP).
4. It lets you modify state machine topology easily at any stage of development. A correctly structured state machine implementation is often easier to modify than the corresponding state diagram.

Through support for executable prototypes, QP offers a light-weight alternative to heavy-weight and high-ceremony CASE tools, for which rapid prototyping has always been one of the biggest selling points. In fact, QP imitates many good features of design automation tools. For example, the QF is conceptually similar to the frameworks found in many such tools. The only significant difference between QP and CASE tools is that the tools typically use a visual modeling language (e.g., UML), whereas QP uses C++ or C directly. In this respect, QP represents the view that the levels of abstraction available in the conventional programming languages haven't yet been exhausted and that you do not have to leave these languages in order to work directly with higher level concepts, such as hierarchical state machines and active objects.

11.1.3 A Learning Aid

Repeatedly, the experience of generations of programmers has shown that to code efficiently and confidently, a programmer must understand how the underlying concepts are ultimately realized.

From my own experience, I recall how my understanding of OOP expanded when I implemented object orientation from scratch in C. I had been using C++ for quite a long time in a very object-oriented (or so I thought) manner. Yet, OOP truly got into my bones only after I saw how it works internally. I started to think about OOP as the way of design, rather than the use of a particular programming language. This way of thinking helped me recognize fundamental OO concepts as patterns in many more systems, which, in turn, helped me understand and improve many existing implementations, not just those that are object oriented or coded in C++ or C (but, e.g., in PL/M).6

I repeated the experience again, this time with the concepts of hierarchical state machines and the active object–based computing model. I have studied ROOM and have built state models with various tools, but I truly internalized the concepts only after having implemented behavioral inheritance and the active object–based framework.

What worked for me might work for you too. You can use the code I've provided as a learning aid for understanding a concrete implementation of the fundamental concepts. I believe that this book and the accompanying CD-ROM will help you

5. See Appendix A and [Samek 97].
6. At GE Medical Systems, I had a chance to work with an embedded system with 500,000+ lines of code programmed mostly in PL/M.
through the process in a few short weeks, rather than several years — the time it took me. When you learn one implementation, you practically learn them all because you understand the concepts. Tools and notations come and go, but truly fundamental concepts remain.

### 11.1.4 A Useful Metaphor

QP owes its name to a powerful analogy between state machines interacting via asynchronous event passing and quantum systems interacting via the exchange of virtual particles. A critique of this analogy might be that programmers are not familiar enough with the physics concepts. However, the physics background necessary to benefit from this analogy is really at the level of popular science articles.

Only recently has the software community started to appreciate the role of analogies and metaphors in programming. A good metaphor is valuable in software for several reasons.

1. It can foster the conceptual integrity of the software.
2. It can improve communications by providing a common vocabulary.
3. It can improve the usability of the end product.
4. It can speed up the learning of new concepts.

Chapter 7 (Section 7.3.1) discusses aspects 1 through 3. Here, I would like to comment only on the last aspect: the role of the quantum metaphor in learning QP.

When people learn new things, they automatically try to map new concepts to familiar ones in the spontaneous process of making analogies. A problem occurs when these spontaneous analogies are incorrect. The new knowledge interferes with the old knowledge (learning interference), and the learning process is more difficult than it would be if the individual did not have the conflicting knowledge in the first place [Manns+ 96]. A correct analogy provided explicitly to the student can speed up the learning process in two ways: by providing correct associations to ease the integration of new concepts with familiar ones and by avoiding learning interference. In this sense, the quantum metaphor can help you learn the fundamental concepts of QP.

### 11.2 Propositions of QP

As I have indicated throughout this book, none of the elements of QP, taken separately, are new. Indeed, most of the fundamental ideas have been around for at least

---

7. If you are a C programmer interested in QP, you might need to go through the exercises exactly in the order I describe. First, study OOP in C (see Appendix A) and only then study QP in C.

8. Inventing a good metaphor is one of the key practices of eXtreme Programming [Beck 00].
Chapter 11: Conclusion

a decade. The contributions of QP are not in inventing new algorithms or new theories of design (although QP propagates a method of design that is not yet mainstream); rather, the most important contributions of QP are fresh views on existing ideas.

Challenging established views is important. An analogy from physics helps illustrate the point. Albert Einstein’s [Einstein 1905] famous publication marks the birth of special relativity, not because he invented new concepts but because he challenged the established views on the most fundamental ideas, such as time and space. However, and what is perhaps less well-known, in the very first sentence of his 1905 article, Einstein gives his reason for shaking the foundations — the asymmetry between Newton’s mechanics and Maxwell’s electromagnetism. Yes, the lack of symmetry was enough for Einstein to question the most established ideas. Ever since, the most spectacular progress in physics has been connected with symmetries.

In this sense, QP pays special attention to symmetries. The hydrogen atom example from Chapter 2 shows how nesting of states arises naturally in quantum systems and how it always reflects some symmetry of a system. This issue alone requires you to consider hierarchical states as fundamental, not merely a nicety, as some methodologists suggest. QP further observes that behavioral inheritance is the consequence of another symmetry — this time between hierarchical state machines and class taxonomies in OOP. Behavioral inheritance and class inheritance are two facets of the same fundamental idea of generalization. Both, if used correctly, are subject to the same universal law of generalization: the Liskov Substitution Principle (LSP) (see Section 2.2.2 in Chapter 2), which requires that a subclass can be freely substituted for its superclass.

The deep similarities among quantum physics, QP, and OOP allow me to make some predictions. The assumption is that QP might follow some of the same developments that shaped quantum mechanics and OOP.

11.2.1 Quantum Programming Language

OOP had a long incubation period. Although the fundamental concepts of abstraction, inheritance, and polymorphism were known already in the late 1960s,9 OOP came into the mainstream only relatively recently. Without a doubt, the main boost for the adoption of object technology was the proliferation of OO programming languages in the 1980s.10 These languages included Smalltalk, Object Pascal, C++, CLOS, Ada, and Eiffel [Booch 94].

QP might go a similar route. The fundamental concepts of hierarchical state machines and active objects (actors) were known already in the 1980s. From

---

9. The first OO language was Simula 67, created in Scandinavia in 1967 to aid in solving modeling problems.
10. Some of these languages are characterized as being object based rather than fully object oriented [Booch 94].
their inception, these ideas have been supported by visual tools, such as Harel’s [Harel+ 98] Statemate. However, as demonstrated in this book, the concepts are viable also with nonvisual programming languages.

At this time, behavioral inheritance and an active object–based computing model are just external add-ons to C++ or C. However, they lend themselves to being natively supported by a quantum programming language, in the same way that abstraction, inheritance, and polymorphism are natively supported by OO programming languages.

The rationale for such a language is the usefulness of QP concepts in programming reactive systems and the relatively low complexity of the implementation. Behavioral inheritance is no more difficult to implement than polymorphism and is probably easier than implementing multiple inheritance with virtual base classes in C++. Yet, language-based support for behavioral inheritance offers arguably many more benefits to programmers, especially to the embedded, real-time software community.

Integration of QP into a programming language could have many benefits. First, a compiler could check the consistency and well formedness of state machines, thereby eliminating many errors at compile time. Second, the compiler could simplify the state machine interface for the clients (e.g., remove some artificial limitations of the current QP implementation). Third, the compiler could better optimize the code.

Many possibilities exist for realizing such a quantum language. One option could be to loosely integrate the QF into a programming language, as with built-in thread support in Java.

### 11.2.2 RTOS of the Future

Rarely can you find a piece of software truly worth reusing, especially in the fragmented embedded software business. Perhaps the main reason is that reuse is expensive, and there simply are not that many truly general pieces of functionality to justify such expenses. One notable exception has always been a real-time operating system (RTOS). Indeed, as hundreds of commercial and other RTOS offerings can attest, the greatest demand for third-party software in the community is for the operating system.

More opportunities for the reasonable reuse of software exist in conjunction with the functionality traditionally provided by RTOSs. State machines and active object–based computing are truly general and need tight integration with an RTOS. In fact, an active object–based framework, such as the QF, can replace a traditional RTOS.

Benefits of such integration are at least threefold. First, active objects provide a better and safer computing model than conventional threading based on mutual exclusion and blocking. Second, the spareness of concepts necessary to implement the QF eliminates the need for many mechanisms traditionally supported in RTOSs.
Therefore, the integrated system would not be bigger than the RTOS itself, and my experience indicates that it would actually be smaller. Third, such an integrated RTOS would provide a standard software bus\(^{11}\) for building open architectures.

### 11.2.3 Hardware/Software Codesign

Advancements in microelectronics have recently enabled the integration of complete, complex systems on a single chip. To cope with the continuously increasing complexity of such systems, designers are considering C and C++ more seriously as languages for describing both hardware and software.\(^{12}\) The motivation for specifying hardware in C/C++ is at least twofold: (1) to manage the increase in the level of abstraction compared to traditional description languages (e.g., VHDL and Verilog) for hardware design; and (2) to reduce the programming language gap between software and hardware engineers working on the same system.

QP, especially if supported natively by a C-like language, is an ideal platform for uniformly representing both software and hardware, specifically because hardware systems are reactive and concurrent by nature. Although at this time hardware designs have not embraced the concept of hierarchical state machines, they almost inevitably will as hardware rapidly approaches the levels of complexity previously found only in software.

Conversely, increasing clock speeds, power dissipation issues, and the limited memory bandwidth of modern hardware call for a different approach to software. As clock cycles get shorter, some parts of a chip are no longer reachable in a single cycle, and it is increasingly difficult to hide this distributed nature from the software. Moreover, software seems increasingly important for intelligent power management (e.g., clock gating — shutting off the clock in parts of the chip that are not in use).

In many respects, modern hardware starts to resemble relativistic quantum systems, in which the speed of signal propagation from one part of the system to another is no longer instantaneous but limited by the speed of light. A quantum programming language that incorporates the quantum analogy has all the mechanisms to handle such signal latencies built in. A programming paradigm exposes the distributed nature of resources (hardware and software), instead of hiding them, as more traditional software paradigms do. Interestingly, exposing the latencies and resource distribution seems to be exactly what hardware experts are calling for [Merritt 02].

---

11. Section 7.3.3 in Chapter 7 discusses the concept of the QF as a software bus.
12. For example, SystemC is an emerging standard of C/C++ class libraries that also includes a simulation kernel that supports hardware modeling concepts (http://www.systemc.org).
11.3 An Invitation

This book, and even my speculative propositions, has only barely scratched the surface of possibilities that the widespread adoption of fundamental concepts such as behavioral inheritance and active object–based computing can bring. Just think of the explosion of ideas connected with OOP. QP is based on no less fundamental ideas and therefore will eventually make a difference in the software community.

If you are interested in advancing the QP cause, you can become involved in many areas.

- Port the QF to new operating systems and platforms, such as Linux, VxWorks, QNX, eCos, MicroC/OS, and others.
- Provide replacements for conventional RTOSs by tightly integrating the QF with schedulers.
- Use behavioral inheritance meta-pattern to capture and document new state patterns precisely.
- Implement QP in languages other than C and C++ — for example, in Java.
- Explore the possibilities of implementing a quantum programming language, perhaps by modifying an open-source C or C++ compiler.
- Publish reusable, active object components.
- And so much more.

I have opened the official QP Web site at http://www.quantum-leaps.com. I intend this site to contain ports, application notes, links, answers to frequently asked questions, upgrades to the QF, and more. I also welcome contact regarding QP through the e-mail address on this site.
Bibliography


IAR Systems visualSTATE® http://www.iar.com/Products/VS/

Interactive Software Engineering,


Microsoft. 2001. Microsoft Knowledge Base article Q96418.


OnTime. 2000. Refer to the On Time documentation on the accompanying CD-ROM.

OnTime. 2001. Refer to the On Time documentation on the accompanying CD-ROM.

Opdyke, William F., “Refactoring Object-Oriented Frameworks”, Ph. D. Thesis, University of Illinois at Urbana-Champaign, 1992. NOTE: this document is available in PDF from the accompanying CD-ROM (\Resources\Opdyke92.pdf)


## Index

### Symbols
- µC/OS 254, 267, 340

### A
- abstract class 340, 342, 344
  - UML notation for 354
- abstract method 137, 347–348
  - UML notation for 354
- Abstract Operating System 208
- abstraction 105, 336
  - and LSP 34
    - as meta-pattern 21, 81, 120, 333, 335–339
    - inheritance 32, 105
    - levels of 5, 15, 17, 131, 165, 328
    - state hierarchy 51, 184
- access control 338
  - private 338
  - protected 338
  - public 338
  - See also visibility of class members
- action 27–30, 41, 43, 49, 53, 55, 59, 67, 224, 253, 237
  - coding transitions 98–99
  - sequence of 38
  - UML notation for 356
  - undefined syntax of 41
  - See also entry/exit action
  - See also internal transition
- action listener 240
- active object 20, 190–191, 197, 248–253
  - as Observer 243
  - priority 241
- active object computing 191, 315
  - and programming discipline 316
  - flexibility of 315
  - See also µC/OS
- actor 190–191
  - See also active object
- ad hoc approach 103, 196, 290
- additive complexity 35

### ADDITIONAL CONTENT
- Adobe Acrobat Reader 362
- ADT (abstract data types) 336
- AECL (Atomic Energy of Canada Limited) 196
- AGILE! macro 222
- Agile Modeling xii
- Arm, Scott xii
- and-decomposition of states 34, 159
  - See also orthogonal regions
- angular momentum 49, 51–52
  - conservation of 50
- ANSI C 121
- application
  - framework 190
  - shutdown 255
- application intentionally incomplete 264
- architectural decay 27, 196–197
  - and guards 26
- ARM (Advanced RISC Machines) 350
  - ARM7TIMI 230
  - See also Atmel, AT91 microcontroller
- arming a timer 256
- ASSERT () macro 106–107, 222
- assertions 220
- asynchronous event
  - dispatching 151
  - exchange 20, 198
- Atmel Corporation
  - AT91 microcontroller family 229–230

### Heuristics for Programming
- rules for programming 315
- activity graphs 42
- actor 190–191
- See also active object
- ad hoc approach 103, 196, 290
- additive complexity 35
- Adobe Acrobat Reader 362
- ADT (abstract data types) 336
- AECL (Atomic Energy of Canada Limited) 196
- aggregate 63, 86–87
- aggregation 151, 355
- UML notation for 355
- agile methodologies xii
- Agile Modeling xii
- ALLEG() macro 222
- Amber, Scott xii
- and-decomposition of states 34, 159
  - See also orthogonal regions
- angular momentum 49, 51–52
  - conservation of 50
- ANSI C 121
- application
  - framework 190
  - shutdown 255
- application intentionally incomplete 264
- architectural decay 27, 196–197
  - and guards 26
- ARM (Advanced RISC Machines) 350
  - ARM7TIMI 230
  - See also Atmel, AT91 microcontroller
- arming a timer 256
- ASSERT () macro 106–107, 222
- assertions 220
- asynchronous event
  - dispatching 151
  - exchange 20, 198
- Atmel Corporation
  - AT91 microcontroller family 229–230
atomic
  change of state 91
  event processing 197
automatic code generation
  See code generation, automatic
automatic contract validation
  See also DBC
automatic event recycling 87, 237, 247–248, 327
automatic garbage collection
  See garbage collection
automatic objects 338
automatic programming 326
automatic variables
  See variables, automatic

B
background processing 283–284
bad idea 234, 247
behavior
  continuous 24
  inherited 17, 33
  modal 6
  sharing of 31
  simple 24
  See also behavioral reuse
  See also state behavior
behavioral inheritance 16–18, 32–34, 81, 331
  and inheriting state models 185
  and state hierarchy 16
  and state machine inheritance 167
  and state patterns 18, 34, 132
  and symmetry 52
  as meta-pattern 17, 81, 248
  tree 110
behavioral reuse 16–17, 31–33, 53, 132
  and class inheritance 82, 168, 185
  example of 48, 50
binary logarithm 286
binary semaphore 193
block device 342
blocking 230, 234–235, 247
  as mode of operation 314
  in active objects 315
  in traditional multithreading 314
Booch, Grady 32
Borland
  C++ compiler 175
  Delphi 6
  OWL (Object Windows Library) 6
bottom-up approach 7, 45, 197
  See also ad hoc approach
Brooks, Frederick x, 42, 206, 316, 336
bug
  See error

C
"C+" (object-oriented programming in C) xiv, 120, 336
  and polymorphism 177
  and statechart refinement 177
C++ 325
  exception handling 224
  object model 82, 336
  standard libraries 228
calculator statechart 9, 170
CallEvent 41
callMemberFn() method 182
canonical state machine
  See state machine, canonical
Cargill, Thomas 224
CASE (computer-aided software engineering) 44, 211–212
cfront compiler 336
chain of responsibility
  See design pattern
change manager 240
ChangeEvent 41
character device 342
choice pseudostate 40
  implementation of 76, 127
  UML notation for 357
ChorusOS 197
CISC (complex instruction set computing) 349
class 337
   diagram 353–355
   inheritance 16, 33
   UML notation for 354
classical FSM 4–5, 10, 28, 31–32, 41, 53, 55, 70, 83, 155
   and entry/exit actions 76
   and guards 76
   and pseudostates 76
   vs. HSM 105–106, 155
classical mechanics 19, 49, 330
   vs. OOP 199–200
cleanup 253–255
clock device 256
clock tick 256–258, 294–295
   interrupt 257
   ISR 259
code generation
   automatic vii, 15, 43–44, 211–212
   and application framework 44
   and QF 212
   and RTOS 44
code-synthesizing tools
   See CASE
coding statecharts
   See statechart
collaboration of classes 355
comment parser FSM 57
commonalities in frameworks 266
compatibility of behavior 168
complete event 142
complete shutdown 255
complete state handler 102, 104
complete transition 142
complexity 26, 29, 32, 160, 168
   additive 35
   and abstraction 32
   artificial 10
   multiplicative 35
   of hardware 332
   of QP implementation 331
   of state machine 103
   See also simplicity
code generation (continued)
   component state machine
       See state machine, component
composite aggregation
       UML notation for 355
composite state 31, 104–105
   and quantum mechanics 49
       See also quantum state
defining 96
   example of 14
   LSP-compliant 34
   UML notation for 356
conceptual integrity xi, 206–207, 329
concurrency model 151
concurrent objects
   UML notation for 357
   constructor in C 337–338, 341
   consumption rate 319
   container state machine
       See state machine, container
   controllability 198
   cooperative multitasking
       See multitasking, cooperative
correlated event production 321
counting semaphore
   See semaphore
coupling 317
CPU utilization 215
critical section 272–273
CRITICAL_SECTION 292
cross-development 301–302

D
dangling pointers 227–228
DBC (design by contract) 106, 220–223
   strategic contract example 240, 244
   tactical contract example 239
deadline 40, 215, 227, 231, 317, 320
   notation for 357
deadlock 194, 198, 232, 306, 314, 316
defer operator 144
defered event 41, 144
   See also state pattern
DEFINE_THIS_FILE macro 222
degenerate state 20, 52
   and symmetry 52
delete operator 225
design automation tools 43, 103, 190, 211, 318, 326, 362
   QP as alternative to viii
   See also CASE
design by contract
   See DBC
design pattern
   Chain of Responsibility 86, 108
   Facade 198
   Mediator 202
   Observer 202, 240, 243, 245
   Singleton 11, 59, 93, 155, 158, 169, 180, 258
   Singleton in “C+” 177
   State 65, 72–73
   Template Method 137, 183
   vs. state patterns 132
   See also Singleton design pattern
   See also state design pattern
desktop application 218
desktop metaphor 206
desktop-style programming 217–219, 225, 229, 263
destructor in C 337–338
deterministic execution 198, 231, 237, 264, 290, 295, 316
Device Mode idiom 155, 163
dialog controls 151
Dijkstra, Edsger 191, 205
dining philosophers problem
   See DPP
directory structure 270
disabling
   interrupts 192, 231, 296
task switching 193, 231
dispatch() method 56
   implementation 108
   dispatching events
   See events, dispatching
distributed system 209
DOS 283–290, 314
   device driver 342
   See also Microsoft, DOS
Douglas, Bruce x, 32, 132, 318
downcasting
   event pointer 89, 154
   this pointer 72
DPP (dining philosophers problem) 191, 202, 230, 245, 306
   Philosopher active object 311
   QF implementation of 306
   Table active object 308
   See also deadlock
Duby, Carolyn 78
dynamic binding 178, 183, 342
dynamic event allocation 237–240
dynamic memory allocation 219, 227–228
dynamic state transitions 110, 116, 164, 180

E
EC++ (Embedded C++) 258, 351
   and exception handling 224
edit–compile–execute cycle 291
Eiffel 222, 330
eight-bit micros 290
electromagnetism 200, 330
electron spin 52
embedded C
   See EC++
embedded Linux 217
embedded real-time system 4, 187, 190, 209, 215–218
   and exception handling 224
   memory management 225, 229
   vs. general-purpose computer 216
enabling interrupts 296
encapsulation 191, 336
INDEX 375

END_CALL macro 348
END_VTABLE macro 347
energy conservation 235
ENSURE() macro 106, 222
EnterCriticalSection() 292
entry action 36–37
and identity of state 37
coding 98
implementation of 76
order of execution of 37
vs. class constructor 17, 37
vs. class initialization and finalization 17
error 218
event 26, 306–315
action paradigm 7
altering the sequence of 144
annihilation of 247
anonymous 142
bursts in production 249
changing type 156
deferring 41, 144
dispatching 99
explicit 156
failure to enqueue 244
initialization 154, 156
kinds of 40
life cycle of 27
production 321
sinks of 322
static allocated 250
synchronous 151
See also asynchronous event
event handler 59
event object 86
event parameters 27, 315
event passing viii, 53, 86, 197, 201, 216, 218, 235–237, 244, 247, 264, 286, 318, 323
event pool 237, 273–277, 322
multiple 238
See QEPool
sizing 322
event processor 62, 83, 106–118
implementation of 106
event queue 27, 249, 298–300, 319–322
ring buffer 278
See QEQueue
sizing 319–320
event-driven systems 1, 3, 12, 83, 86, 132, 139, 159, 187, 191, 197, 326
exception handling
and EC++ 224
state based 224
exception handling policy 220
exceptional condition 219, 223
exclusive access 192
executable
code 132
model xii, 43, 249, 317, 327
execution
profile 284
thread 152, 298–300
exercises, answers to 362
exit action 36–37
and identity of state 37
coding 98
implementation of 76, 98
order of execution of 37
vs. class constructor 37
vs. class destructor 17, 37
vs. class initialization and finalization 17
extended state 25
extended state machine 25
extended state variable 25, 101, 196
vs. orthogonal components 159
extension points 190, 212

F
Facade design pattern 198
fairness 194
Feynman diagrams 200
vs. sequence diagrams 200
Feynman, Richard 200
FIFO 143
final state 30, 48
explicit 135, 255
UML notation for 356
fine granularity 267
finite state machine
  See FSM
flowchart 42, 76
  vs. statechart 41
focus of control
  UML notation for 357
foreground processing 283–284
fork pseudostate
  See pseudostate, fork
fragmentation
  lengthy processing 143
  of heap 226
framework extension points 212
free () 225, 231
FSM (finite state machine) 24
  and repetitions 32
  Optimal 69, 83, 88
  vs. HSM (hierarchical state machine) 155
  See also state machine
fudge factor 320
functor 181
G
  Gamma, Erich 18, 33
garbage collection 227, 236
generalization 354
    arrow 345, 354
general-purpose computer
    vs. embedded system 216
global namespace 258
GNU gcc
    -fvtable-thunks option 175
GPS receiver x, 207–209, 242
graceful shutdown 255
graphical notation 353
guaranteed cleanup 92, 148
  and class destructor 36
  and exit action 36
guaranteed event delivery 210, 245
guaranteed initialization 92, 148
  and class constructor 36
  and entry action 36
guard 26, 41, 43, 48, 50, 76, 102–104, 154, 159, 183–185, 193, 212, 311, 327
  and architectural decay 26
  implementation of 76, 99
  sequence of evaluation 43
  UML notation for 40, 356
GUI (graphical user interface) 3

H
  hard real-time 209
  Harel statecharts 24
  Harel, David x, 4, 24, 41–42
  has a (has-a-component) relationship 248
heap
  fixed-block-size 237
  infinite 228
heap problems 225–228
  block allocation overheads 226
  fragmentation 226, 228
  heap as a shared resource 226
  mass oversizing heap 226
  nondeterminism 226
  priority inversion 226
  sharing 226
Heisenberg uncertainty principle 51
Hejlsberg, Anders xi–xii
heuristics for active object–based systems 318
Hewitt, Carl 191
hierarchical states 4, 31–32
  semantics of 5
  symmetries 32
high-water mark 279
history
  clearing 164
  See also pseudostate
history mechanism 39, 127
Horrocks, Ian 6, 78
host machine 301
housekeeping code vii, 15, 43, 211–212
HSM (hierarchical state machine) 31–32, 155
  and opportunity for reuse 32
  malformed 126
  See also QHSM class
See also state charts
See also state machines
hydrogen atom 49, 52

I
IAR
VisualState 44
I-Logix
OXF (Object Execution Framework) 190
Rhapsody 44, 190, 208
Statemate 44, 331
incomplete prototypes 317
inconsistent configuration 196
inheritance
aggregate 87
and class instantiation 37
and programming-by-difference 17
as cornerstone of OOP 20, 34, 81
as meta-pattern 17, 81, 350
benefits of 33
exact 16
for classes 16, 33
UML notation for 353
for states
See behavioral inheritance
in C 120, 178, 335, 339
example of C 340
multiple (MI) 75, 89, 181–182
of entire state models 85, 92, 138, 168–169, 186
of state machines 168
of statecharts 167
overhead of 349
state transition 101, 111
UML notation for 354
initial pseudostate
See pseudostate, initial
initial transition 10, 30, 56, 93, 98–99, 143, 154, 202, 252, 261, 288, 310
and Init() method 56, 59, 83
coding of 98
hard-coding of 63
implementation of 89
reuse of 46
topmost 94
UML notation for 356
vs. class constructor 95
See also QHsm::init() method
initialization 253–255
of an HSM 99
initialization of an event
See event, initialization
installation 361–362
instantiation 154
Intel
8089A 297
x86 processor 266, 273, 296, 349, 361
intermediate
bosons 20
vector bosons 199
internal transition 39, 102, 134, 145, 149, 310
coding 98
UML notation for 356
vs. self-transition 39
interrupt latency 231, 273
and critical section 273
interrupt service routine (ISR) 227, 283, 342
interrupts, enabling 296
intertask synchronization 207
INVARIANT() macro 106, 222
IRQ 297, 342
is a (is-a-kind-of) relationship 17, 33, 248, 354
element of (QActive is a QHsm) 248
is in (is-in-a-state) relationship 17, 31, 33, 354
IS_IN() operator 159–160
ISR
See interrupt service routine

J
Java 227, 331
and "C++" 340
AWT (Abstract Windows Toolkit) 6
event model 240
native interface 336
preventing subclassing with final keyword 105
jitter, notation for 357
join pseudostate
    See pseudostate, join
junction pseudostate 40
    implementation of 76
    See also pseudostate, junction

L
Labrosse, Jean 340
Latch state pattern 159
LCA (least common ancestor) 38, 92, 110
LeaveCriticalSection() 292
legacy systems 198
Leveson, Nancy 195
lexical intelligence xii
licensing xv
LIFO 143, 274
Linux 217
look and feel 133
    consistency 4, 133–134, 182
low-water mark 275
LSP (Liskov Substitution Principle) 34, 168, 185, 330
    and compatibility of behavior 168
    and quantum mechanics 49, 52
    and reactive classes 183
    and state machine inheritance 183
    for classes 34
    for states 34, 207

M
malloc() 219, 225, 231
manual coding vii
many-to-many interactions 202
map file 229
Mars Pathfinder mission 232
master–slave 158
Mathworks Stateflow 44
me pointer 337
Mealy and Moore state machines 43
Mealy automaton 28
Mediator 240
Mediator design pattern 202
Mellor, Steve 32
memory leak 219, 227, 236, 264, 327
memory partition 237, 273
memory pool 237, 273, 298
memory, conserving 322
message mailbox 194, 207, 250, 298, 316, 340
message pump 249
message queue 250, 277, 298, 340
meta-pattern, abstract 17
metaphor xi, 206
Meyer, Bertrand 220, 223
MI (multiple inheritance) 75, 181, 361
    and pointers to members 75
    compatible HSM 181
MicroC/OS-II
    See µC/OS
microkernel architecture 197
Microsoft
    ActiveX 211
    C/C++ compiler 175
    Developer Studio 266
    Document/View architecture 211
    DOS and QF 265
    MFC (Microsoft Foundation Classes) 6, 12, 211
    Visual Basic Calculator 6
    Visual C++ xiv, 211, 266
    See Win32
    See Windows
Microsoft Foundation Classes
    See Microsoft, MFC
minimal communication 317
modeling
    at the code level xii
    early stages of 247
Moore automation 28, 30, 43, 53
    and entry/exit actions 36, 89
    See also Mealy automation
Moore, Gordon E. 209
Moore’s Law 208–209
multicasting events 245–247
multitasking, cooperative 290
multithreading 208
  and deadlock 194
  and fairness 194
  and nondeterminism 194
  and race conditions 192
  and starvation 194
  and system utilization 194
object based 208
Murphy, Niall 217
mutex 226, 231–232, 237, 251, 314, 324
semaphore
  See mutex

N
namespace, global 258
naming convention 78
NASSERT macro 222
nested states 16, 31–34, 38, 135, 145, 168
  See also hierarchical states
nested switch statement 57–60
network processor 210
neutrino 201
new operator 219, 225
nondeterminism 194, 215, 226–227, 284, 295, 323

O
object based multithreading 208
object class 343
  ObjectAbstract() method 345
  ObjectCtor_() method 345
  ObjectIS_FIND_OF() macro 345
  ObjectXtor_() method 345
  vptr__ attribute 345
object composition 150
object oriented analogy ix, 16–19, 82, 94–95, 336
ObjecTime toolset 326
observability 198
observables 51
Observer design pattern
  See design pattern
observers 240
OMG UML specification
  See UML
onAssert__() function 222
one-shot timeout 256
one-to-many interactions 202
OnTime
  RTKernel-32 and QF 266
OO programming languages 330
OOP
  costs of 349–350
  vs. classical physics 199
opaque shell 198
open architecture 209
Optimal FSM
  See FSM, Optimal
or-decomposition of states 34
orthogonal component
  See orthogonal region
orthogonal regions 34, 36, 150, 159
  and concurrency 36
  and order of event dispatching 36
  and state patterns 132
  UML notation for 35, 357
orthogonality
  approximate 35
OS_EVENT 340
OSF/Motif 336
overhead
  of adding states 27
  of behavioral inheritance meta-pattern 129, 176
  of C++ exception handling 224
  of heap 226, 229
  of OOP 349–350
  of priority inheritance 231
  of RTOS abstraction layers 208
  of UML statecharts 82
OXF (object execution framework) 208
See also I-Logix, Rhapsody
OXF (object execution framework) 190
overhead 208
package scope
- header file 271–272
parallel computing 210–211
parent–child 158
Parnas family 266–267
Parnas, David 266, 350
parsing numerical expressions 45
passing events 235–237
PATH environment variable 361
periodic timeout 256
platform dependence 267
pointer-to-member function 62, 76, 169
- as an aggregate 174
- syntax 172
pointer-to-virtual-member function 169, 173–176
Polling state pattern
See state pattern, polling
polymorphic call 348
polymorphic event triggering 41
polymorphism 168, 335, 341–348
- as meta-pattern 17
- realized in hardware 342
port.h 271
preemptive 230, 295
preemptive model 28
preemptive priority-based scheduler 230, 246, 295
- priority 230
- priority based kernel 295
- priority ceiling 232–233
- mutex 232
- protocol 232
- priority inheritance 232–233
- mutex 232
- priority inversion 231, 233, 237, 292
- priority numbering in QF 251
- priority of active object 241, 251
- production rate 319
- programmable interrupt controller 297
- programming-by-difference 4, 17, 92, 134, 173
- and reuse of behavior 16
- programming-in-the-large 19
- prototype, intentionally incomplete 327
- prototypes
- incomplete 317
pseudo-code
- UML notation for 354
pseudostate 39–40
- deep-history 40, 127, 160
- fork 40
- history 40
- initial 40, 92–93
- join 40
- junction 40
- shallow-history 40
- UML notation for 357
See also choice pseudostate
public scope header file 269–271
publish-subscribe 201
- mediator 240
- model 240–245
- observers 240

Q
Q_DEFINE_CALL_MEMBER_FN() 182
Q_EMPTY_SIG signal 91
Q_ENTRY_SIG signal 14, 90–91, 95, 98, 107, 116
Q_EXIT_SIG signal 90
Q_INIT() macro 84, 90, 93, 98, 127
Q_INIT_SIG signal 90, 95, 98, 104, 107, 116
Q_NEW() macro 240, 277, 308–309
Q_STATE_CAST() macro 107
Q_TRAN() macro 84, 91, 99, 109, 127, 161, 180
- transition sequence 91
Q_TRAN_DYN() macro 84, 109, 161, 181
Q_USER_SIG signal 90, 307
QActive class 202, 204, 249, 268, 287
- enqueue() method 288
- myEqueue attribute 268
myThread attribute 268
postFIFO() method 144, 261, 288
postLIFO() method 144
postLILO() method 288
QActive() constructor 261
run() method 246, 252, 288
start() method 229, 242, 250–251, 254, 261, 288, 294
stop() method 261, 288
qassert.h header file 107, 221
QEPool class 237, 274–275, 277, 298
data structure 274
get() method 238, 275–276
init() method 238, 275
myEvtSize attribute 274
myFree attribute 274
myNfree attribute 275
myNmin attribute 275, 322
myNtot attribute 274
put() method 238, 275–276
QEQueue class 277, 282
data structure 278
declaration 277
get() method 280
init() method 279
myEnd attribute 278
myFrontEvt attribute 278, 282
myHead attribute 278
myNmax attribute 279, 319–320
myNtot attribute 279
myNused attribute 279
myStart attribute 278
myTail attribute 278
putFIFO() method 278, 281
putLIFO() method 278, 282
QEvent class 12, 84, 86–87, 202, 204, 308
poolId attribute 239, 248
useNum attribute 244, 246, 248
QF (Quantum Framework) 190
and Design by Contract 223
API 258–263
application, incomplete 247
as software bus 210
change manager 240
design of 215
error and exception handling policy 220
event queues 250
integration with I/O 323
memory management policy 230
time management 217, 220, 255–258, 294
vs. RTOS 208
QF class 258
add() method 242
background() method 259, 285
cleanup() method 259
create() method 239–240, 259, 277
definition() method 259
init() method 56, 242, 253, 259, 276
poolInit() method 239, 253, 259
propagate() method 246
publish() method 241, 245, 259, 309
subscribe() method 241–243, 259, 309
tick() method 259, 300, 313
unsubscribe() method 241, 243, 259
QF_EQUEUE() macro 268
QF_EQUEUE_INIT() macro 279
QF_EQUEUE_ONEMPTY() macro 280, 287
QF_EQUEUE_SIGNAL() macro 281–282, 287, 292
QF_EQUEUE_WAIT() macro 280, 287, 292
QF_ISR_PROTECT() macro 292
QF_ISR_UNPROTECT() macro 292
QF_PROTECT() macro 273
gf_rtk32.h 299
QF_THREAD() macro 268
QF_UNPROTECT() macro 273
qfpkg.h 271
QFsm class 88, 153
vs. QHsm class 156
QFsmState 88
QHsm class 11, 83, 85, 202, 248
constructor 93
dispatch() method 99, 108, 163, 252
define() method 163
in C 337
in "C+" 121–122
  init() method 99–100, 107, 252
  isIn() method 109, 159
  mySource attribute 111
  myState attribute 84, 111, 163
  subclassing 95
top() method 92
tran() method 111–112, 181
  tranSetup() method 118
  tranStat() method 117, 180
  Tst statechart 95
  virtual destructor 85
QNX Neutrino 197
QP (Quantum Programming) x, 4, 207, 326
  and behavioral inheritance 81
  and internal structure of classes 19
  Language 330–331
  mission of viii
  Web site 333
QPseudoState 88
QSignal (quantum signal) type 87
QState (quantum state) 84, 88–89
  type 84, 88
  upcasting to 99
QTimer class 204, 256, 262, 311
  disarm() method 263
  fireEvery() method 256, 262
  fireIn() method 256, 262, 312
  rearm() method 263
quantum analogy x, 19–20, 49, 199–201, 207
quantum calculator 44–48
  design of 44
  extended version 168
  implementation of 10–16
quantum field theory 200, 207, 235
Quantum Framework
  See QF
quantum leap 20, 29, 207
quantum mechanics ix, 19
quantum number 52, 78
quantum programming
  See QP
quantum state 20, 51–52, 207
quantum vacuum 202, 236

R
race condition 192, 195–196, 290
RAD (Rapid Application Development) 6
  model building 327
  prototyping 328
Rational Corp. Development Studio 44
reactive base class 173, 177, 180, 183
reactive class 79, 167–168, 183, 185
reactive component 155, 159
reactive system 4, 6, 8, 16, 20, 23, 52, 54, 73, 132–133, 144, 177, 187, 197, 207, 331
ready list 285
realloc() 231
real-time
  framework 43
  multitasking kernel 295
  operating system
    See RTOS
  See also hard real-time
  See also soft real-time
reentrancy 226
refactoring 18
  of state models 18–19
regular transition 99
reinterpret_cast 107
reminder 138–143
Remainder state pattern 143, 158, 290
  and state-based exception handling 224
reparenting 185
REQUIRE() macro 106, 222
resetting a state machine 135
resource allocation 192
resource sharing
  avoiding 317
  responsiveness 317
reuse
  and class inheritance 336, 341
  and design pattern 132, 159
Index 383

and orthogonal region 150
  design pattern 19
  essence of 17
  high cost of 331
  in application frameworks 190
  in class libraries 190
  in embedded systems 331
  in frameworks 190
  in HSM 32
  of behavior
    See behavioral reuse
  of code 34, 72, 182, 185
  of legacy code 256
  See also programming-by-difference
reverse inheritance 185
  ring buffer
    and event queue 278
RISC 350
  role in collaboration 355
ROOM (real-time object-oriented modeling) x, 103, 149, 164, 191, 317, 326
  ROOMchart 24, 77
  virtual machine 149, 190
RTC (run to completion) 28–29, 143, 151, 234, 249
RTKAllocMemPool() 298
RTKCreateMailbox() 299
RTKCreateThread() 299
RTKDisableInterrupts() 296
RTKDisableIRQ() 297
RTKEnableInterrupts() 296
RTKEnableIRQ() 297
RTKernel-32 266, 295–302, 314
RTKernelInit() 300
RTKFreeBuffer() 298
RTKGetBuffer() 298
RTKMailbox() 299
RTKPutCond() 300
RTKPutFrontCond() 300
RTKTaskHandle 299
RTKTerminateTask() 300
RTOS (real-time operating system) 202, 207–208, 331
  abstraction layer 208
  and conventional multithreading 208
RTOS-32
  evaluation kit 361
  manual 362
RTTARGET environment variable 361
RTTarget-32 289
RTTI (run-time type identification) 345
Rumbaugh, James 150
run to completion
  See RTC
  rush-to-code syndrome 327
S
  scalability 198
  scheduler
    preemptive priority-based 230, 246, 295
  scheduling algorithm 207
  self-transition 39
    vs. internal transition 39
  Selic, Bran x, 317–318
  semaphore 193, 231
  sequence diagram 306, 317, 357
    vs Feynman diagram 200
  sequential multicast 246
    See also multicasting events
  sequential programming 194
  serial port 234
  SetEvent() macro 292
  shallow history 160
    pseudostate 40
  shared behavior 31
  shared data 152
  shared memory 283
  shared resources 203
  shutdown, graceful 255
  signal 306–315
    granularity 103–104
    latencies 332
    propagation delays 200
  SignalEvent 40, 84
signals
  consolidation of 46
  enumerating 95
  uniqueness of 307
simple behavior 24
simple state 31, 104
simplicity xi
Simula 67 330
single inheritance 340
Singleton 311
Singleton design pattern 59, 158, 180, 258
  example in “C+” 177
  example in C++ 169
sinks of events 322
sizing event pools 322
sizing event queues 319–320
sleep mode 255
  Sleep () 295
soft real-time 209
  requirements 227
  systems 247
software bus 208, 210, 332
source code 361
source state 38
spatial intelligence xii
specialization 354
spike solution 327
Standard Template Library 228
starvation 194
state
  based exception handling 224
  behavior 20, 24, 34, 49, 53, 72, 149–150,
  158, 167, 182
  combinatorial explosion 35
  configuration 38
  design pattern 65
  explosion 35
  identity of 37
  in quantum mechanics 49
  most recent configuration 160
  mutual consistency in 245
  or-decomposition 34
  removing 185
  simple 31
  116, 184, 356
  UML notation for 356
  variable 73–74
  See also composite state
  See also degenerate state
  See also final state
  See also hierarchial state
  See also nested state
state design pattern 65–69
state handler
  defining 96–99
  incomplete 102
  malformed 128
  signature of 89
state handler method 12, 59, 72
  defining 96
  example of 14
state hierarchy 31, 77–78
  abstract 31
  and quantum mechanics 49
  circular 128
  LSP-compliant 34
state machine 154
  as remedy to highly conditional code 23
  canonical 134
  component 151
  container 150
  extended 25
  interface 56–57
  resetting 135
  terminating 48, 135, 148, 254–255, 261
state models 185
  malformed 104
state pattern 18, 132, 165
  and behavioral inheritance 18, 34, 132
  and orthogonal regions 18, 132
  deferred event 144–149
Orthogonal Component 150–160
polling
Transition to History 164
Ultimate Hook 137
  and exception handling 224
See Reminder state pattern
state table 60–64
  optimal FSM 75
state transition 5, 27, 91–92, 101, 112
  and RTC step 29
  and sequence of actions
    See transition execution sequence
diagram 5, 30, 32
    See also statechart diagram
  UML notation for 356
in Mealy/Moore automata 28
reuse of 10, 31
semantics of 38
UML notation for 356
uninterruptable 21
  vs. quantum leap 21
state variable 25, 57, 59, 64, 72, 75, 79, 95, 101, 196
  vs. orthogonal components 159
statechart vii, 4
  and active objects 197, 248
  and class inheritance 167
  and even queue 249
  as a type of design 326
  automatic code synthesis 43
coding of 10, 98
constructive nature of 212
diagram 5, 9, 30, 32, 39, 45, 47–49, 98, 170, 204, 356–357
essence of 30
example in C 124
executable model of 43
overhead of UML 82
refactoring of 19
refined 170
refining through inheritance 168, 178, 183
semantics vs. notation 41
UML notation for 356–357
  vs. flowchart 41
    See also UML, statechart
static binding 178
static initializer 338
static objects 338
static transitions 110
strategic contract 244
strict inheritance 184
  in state models 184
struct in C++ 86
subclass 339
subclassing
  Calc1 169, 179
  framework classes 211
  preventing 105
  QActive 202
  QEvent 84, 240, 308
  QHsm 95, 102, 128
  StateTable 63
subevent 41
subscriber list 242
  lookup table 254
  direct 31
    See also transitive substate
subtyping 184
  in state models 184
super pointer 340
superclass 339, 346
superloop 207
superstate 31
  designing the 99
  symmetry 207
  and state hierarchy 49
synchronizing
  access 192
  orthogonal regions 35
synchronous event dispatching 151
system utilization 194
SystemC 332

T
target machine 301
target state
  See state, target
Index

task-level response 284, 295
Telelogic ObjectGeode 44
Template Method design pattern 137, 183–184
testability 198
test-and-set operations 193
testing 197
and executable models xi
Therac-25 195–197
“Thin wire” style of communication 198
this pointer 337
THIS_FILE__ 222
thread of execution 251–253
thread routine 252
thunk technique 175
time management
See QF
timed blocking 255, 277, 298
TimeEvent 40
timeliness
of event delivery 245
requirements 216
timer 256
arming 256
disarming 256
phasing of periodic 256
rearming 256
timing diagrams 357–358
top state 31, 92
Tran class 58, 85, 116
tran() method 111
Tran_class in C 122
tranSetup() method 118–120
transition
action functions 63, 78
coding initial 98
coding regular 99
execution sequence 37–39, 91
in UML 38
removing 185
sequence of QP 92
to history 160–164
UML notation for 356
See also action
See also initial transition
See internal transition
tranStat() method 116
trigger (triggering event) 27
TRIGGER() macro 107
typical clock rates 256

U
Ultimate Hook 4, 6, 133–137
state pattern 137
state-based exception handling 224
UML (Unified Modeling Language) 15, 39
and well-formedness rules 104
meta-model 77
noncompliant statecharts 105
notation
abstract class 354
abstract method 354
aggregation 355
choice point pseudostate 357
class 354
concurrent objects 357
final state 356
focus of control 357
history pseudostate 357
inheritance 354
initial transition 356
internal transition 356
orthogonal regions 357
pseudocode 354
state 356
transition 356
specification x, 24, 39, 41, 92, 104, 362
See Appendix B
state machine meta-model 77
statechart 24, 30–44
uncertainty principle 200, 235
unfixable 196
variabilities
  in frameworks 266
  See also variables, automatic
variables
  automatic 250, 322
  vcall thunk 174
  VCALL() macro 348
  VHOOK() macro 347
  virtual bosons 201
  virtual destructor 345
  virtual memory 227
  virtual particle 200, 235
  virtual photon 200, 235
  virtual pointer 177, 343
  virtual state handler 75, 171
  virtual table 175, 177, 343
  visibility of class members 354
  Visio™ stencil 362
  Visual Basic Calculator 196
  visual modeling language 328
  visual programming vii, 41
  visual tools 211, 326
  visualSTATE engine 190
  VMETHOD() macro 346
  VPTR (virtual pointer) 121, 343
  VPTR() macro 178, 348
  VTABLE 121–122, 125, 178, 343
  VTABLE() macro 346
  VxWorks
  See WindRiver, VxWorks

WaitForSingleObject() 292
watchdog timer 256
well formedness 331
Win32 290–295, 314
  API 12, 193
  console application 295
Windows xiv, 83, 132, 136
  9x 292
  and QF 266
  as reactive system 12
  CE 217
  NT 292
WindRiver Systems 252
  BetterState 44
  VxWorks 230, 232, 252
WinMain() 12, 56, 86, 136, 295
  WM_COMMAND 12, 104
  WM_TIMER 295
worst-case pool utilization 275
worst-case queue utilization 279

XP (eXtreme Programming) xii, 57, 206, 327, 329

Zeeman effect 49
zero overhead principle 82
What’s on the CD?

The companion CD-ROM contains all the source code and executable images mentioned in the book, including several ports of the Quantum Framework (QF). The disc also includes answers to the exercises scattered throughout the book, the Evaluation Version of On Time RTOS-32 v4.0, Visio™ stencils used to create the diagrams in this book, and several references in Adobe Portable Document Format (PDF).¹

The CD-ROM is designed for maximum usefulness, even without installing any of it on your hard drive. In particular, you can browse the source code, execute examples, and read PDF documents directly from the CD.

The disc comes with an HTML-based index page, index.htm, in its root directory. It is automatically activated if the CD autoinsert notification is enabled on your system. You need a Web browser (e.g., Microsoft Internet Explorer or Netscape Navigator) to view the index.

For more information on the source code structure, installation, answers to exercises, or other CD resources, see Appendix C, beginning on page 359.

---

1. A copy of Adobe Acrobat Reader™ is included on the CD-ROM for your convenience.