The Trading-shim Manual

Bill Pippin

September 26, 2008

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Given the appropriate exterior program, the shim provides a command-line and dbms augmented interface to the classic three functions of any brokerage interface, manual or automated: access to market data, execution of market transactions, and account reporting.

What program provides the market services? The Interactive Brokers Trader Workstation, IB tws client, or simply tws, consists of a socket-based api and java gui that together provide access to both external markets and the user's IB account. The shim provides access to a running instance of the tws, and so these services, via the socket api.

1.1 Outline

This manual provides both tutorial and reference information about the shim. Although the tutorial, Part I, is intended to be most useful to the trader or researcher who is writing downstream programs that would use the shim, and the reference, Part II, to the developer looking to understand or modify the source code of the shim, there are many cross references between the tutorial and reference, so that they are combined here into one manual.

The tutorial describes installation, setup, and startup, in Chapter 3; use of the shim command language, in Chapter 4; modifications to the database, in Chapter 5; and a number of other application-specific topics, collected together in Chapter **??**. The reference has been commented out for now, about which more in the next section.

1.2 Caveats

The trading-shim is currently under active development, as is the IB tws, so that documentation for the shim aims at a moving target. Please understand that this manual, as so the shim, is currently a work in progress. Much of the text is incomplete, particularly in the reference section, where it has been collected from many shorter works written over the course of development, and not yet revised for consistency. That being said, most of the tutorial is newly written for the benefit of the first-time user, if only to answer support questions from the mailing list, and I hope the tutorial will be of immediate use to readers. In addition, many of the diagrams in the reference section are automatically generated, and these figures should be of great help to anyone tackling the sources for the first time.

As an aid to the reader in estimating the amount of text in each chapter, and to help one avoid repeatedly drilling down to empty sections, Table 1.1 gives the page counts and status of each chapter. In that table, the nil sign [] indicates a chapter that is either empty or nearly so, and a checkmark, one that has reached final draft status; those are essentially complete outside of updates as either tables listed immediately prior to the checkmark, or features listed in Chapter 11, are completed.

Ch	Title	Pages	Status	
1	Introduction	6	Table 1.1	✓
2	Why the Shim?	7		✓
3	Installation, Setup, and Startup	23		
4	Using the Command Language	9		[]
5	Adding Info to the Database	**??**		[]
??	Using the Shim	5		[]
A	Related Command Scripts	5		[]

Table 1.1: Chapter page counts and completion status

Feedback from readers has indicated that the partially completed chapters of the manual were confusing, and so they have now been commented out. Those chapters that are checked off in Table 1.1 are essentially finished, and are included herein; those with a blank status entry are in the process of being written, and may be included; and those with a status of nil, [], are just placeholders. *Readers who are not sure what the rest of this paragraph means should skip over it secure in the confidence that they would not care even if they fully understood.* Developers who wish to look at the unfinished parts of the manual, in particular the reference, may compare the files `manual.tex`, `manual.full.tex`, and `manual.part.tex` in the `doc` directory, using say the `diff` utility, after which case the steps to build a bleeding-edge version of the manual should be obvious. Or you can ask on the list, and I'll be glad to help.

1.3 Notation

Newly introduced terms are typically emphasized using an *italic* font the first time they appear. Command line dialogues, source code fragments, and in some cases identifiers borrowed from program source code are printed using a `monospaced` font.

Tables or figures that are referred to but not yet defined are indicated by double question marks, ??, in the text, and a small square box in the online version; such missing references can be found in the document sources by searching for the string `FIXME`.

1.4 Acknowledgements

The trading-shim would not exist if Russ Herrold had not had the confidence and determination to take risks by starting a partnership to develop it. I owe him not only for the opportunity to write the shim, but also for constant encouragement, and especially the many hours we've spent debating design issues. Our IT skills are complimentary, and the shim is a much stronger program as a result of our dialogues. I'm fortunate indeed to have been able to work closely with Russ these last two years.

Though Russ was the first user, there are now others as well. I'd like to acknowledge here my debt to those who use the shim and have contributed feedback. Thank you, those who have downloaded the shim, or joined the mailing list, or submitted questions, whether to the list, or via private email. Every time I stop to think about some question from a new user, I gain new ideas of how to make the shim widely useful. Don't ever discount the value of your own experience or knowledge; your fresh perspective is precious. As an experienced developer, I well know how to implement software; but it's the comments and questions from *you* that teach me how to make it useful to the community. I want to reemphasize this point: feedback from users — your feedback — is critical to us. It plays an irreplaceable role is shaping our plans, design, and especially roadmap. Please let us know what you think.

I'd also like to mention other debts, to: the free software ecosystem, § 1.4.1; tools and references I've used during development, § 1.4.2; authors on finance, § 1.4.3; before, last, considering more personal debts, in § 1.4.4.

1.4.1 Free Software

I have too many debts for the universe of free software I use to ever hope for a comprehensive acknowledgement.

I'll note, then, just two men who stand out, both in achievement and dedication. Richard Stallman has been widely recognized and attacked alike for his extraordinary contributions to the free software ecosystem, in particular the *General Public License*, the value of which becomes more and more apparent over time; his critics are themselves proof that he is a man of justice. Jon Postel is perhaps less widely known, yet spent his life to create and protect a free Internet, as proven by the body of RFCs.

1.4.2 Developing the Shim

I have Stroustrup [1] at my side while I code; his book is a superb starting point from which to learn C++. That language is in my opinion the first and still the best multi-paradigm [2] [3] programming language to become widely available, and I'm fortunate to be able to write programs with it.

Beyond my tremendous intellectual debt to the originator of C++, a number of other texts were of direct help in the design and implementation of the shim.

There is no other algorithms text comparable to Cormen [4], and I would not have been able to write the library components for the shim without the lessons I learned from that text back in graduate school. Daniel Sleator [5] provided the initial code for the splay tree component § C.4.3, and Bob Jenkins [6] [7] the initial code for the hash code algorithm § C.2.2. In each case only their generosity in releasing the sources without restriction made it possible for me to borrow from their work, and any remaining problem with those components reflects on my implementation only, and not in any way on their original work. Stevens [8] is the ideal reference for Unix system calls; it was much harder to write programs to the Unix api before his text appeared. Comer has not only written a superb introduction to TCP, but in addition his applications text [9] is a great starting point for the development of any program that uses TCP sockets.

For anyone wanting to write programs using the MySQL C client library, the Doorstop [10] by Dubois is an invaluable help; he's even been kind enough to put the critical chapter online. The transaction processing text by Gray and Reuter [11] provided a critical contribution to our efforts to eliminate dbms-related race conditions.

When you need `make` to work right, there is no good alternative to the O'Reilly text by Mecklenburg [12]. I look forward to acknowledging at some future time a working debt to the New Rider's Gnu autotools [13] text, though at this point we don't yet have a `./configure` build process in place; it's on our list of tasks not yet done.

It will be obvious to any reader who's compiled the documentation, or even recognizes the Computer Modern fonts, yet I'll say it anyway: this manual wouldn't exist without the extraordinary expenditure of time and effort Knuth [14] devoted to TEX.

I use other tools to build on this foundation: Lamport's `latex` [15] macro system for TEX, Emden Gansner's `dot` [16] graph layout program to create figures, and Drakos' `latex2html` [17] converter to produce the hypertext version of this manual.

I mentioned Stallman earlier, and his invention of the GPL, and how recent events prove its importance more than ever. Closer to home, the licenses for `latex2html` and `gnuplot`, whether considered truly free or not, (their terms seem problematic to me when considered in the light of the last two of the *Four Freedoms*) demonstrate how the General Public License is the right way to protect free software. In this vein I'd also like to offer my thanks here to Eben Moglen, and his efforts on behalf of version 3 of the GPL. Those of us who write — or, for that matter, use — free software owe a great debt to the activists who struggle on our behalf.

1.4.3 Learning About Finance

Malkiel [18] was the first book on investing I ever read; it was published starting in 1973, and is now in the ninth edition. No matter how others may overstate the efficient market hypothesis, still his book is a welcome antidote to snake-oil indicator tracts. In my opinion, the most important book I have yet read on markets is by Mandelbrot [19]; without it I would be hard-pressed to point out the flaws in the strong form of the efficient market hypothesis. As a developer recently departed from academia, I found the autobiography by Derman [20] fascinating for its personal look at software development.

The rest of the texts that follow are much newer to me, though many readers will have encountered them long ago. I list them below to note the debt my colleague, Russ Herrold, acknowledges to them.

Graham [21] is famous as an influence of Buffet, and Lynch [22] [23] is a more accessible starting point for the fundamentalist. Any technician needs encouragement and warnings alike; war stories provide both, and the books by Schwager have them in abundance: [24] [25] [26]. The portrait [27] of Livermore forms a class by itself. At some point, the strategy researcher must consider actual trading algorithms, and here there is much to be said for the classics, in particular the following invaluable works by Keltner [28] Ainsworth [29] and Wilder [30].

1.4.4 Personal Thanks

Any reader who carefully examines this manual, the web site, or the component library will realize I have definite ideas about how to reduce software complexity. I'm greatly indebted to a fellow researcher, Jung Choi, for extensive discussions before I even began work on the shim. We had hoped to carry out research together, and though funding issues prevented it, I'd at least like to acknowledge here the importance of our free-ranging discussions on software engineering, programming languages, and free software.

The creation of free software has its own unique challenges, and my friends have been great encouragement while I grappled with them. As the first user, Russ Herrold has been such a good friend. In addition, outside of the workplace, Rich Burgan, Derek

Yang, Satya Pattanaik, and Marc Flynn have been sympathetic listeners to my blow-by-blow accounts of the development process, showing much more interest than the topic probably deserved.

My family also has been unstinting in their encouragement, my sisters Tina, Sonny, and Tricia always there to listen, and my mother Paula Pippin steadfast and encouraging over a very long time, not just for the shim, but before that with my graduate school work also, which took much longer.

I have too many debts from grad school to mention here, and so for the most part I'll leave the acknowledgements in my dissertation [31] to carry that burden. A few, however, are so far-reaching that I'll repeat them here. A Buddhist teacher, Daisaku Ikeda, with his emphasis on the value of life-long learning, provided the encouragement that sent me back to school even after some time in industry, and my debt here is far, far beyond any explanation. My first advisor, Spiro Michaylov, and second advisor, Neelam Soundarajan, were each excellent teachers, and I owe much of my achievements in grad school to them.

Chapter 2

Why the Shim?

Given the IB tws api, and the desire to use it, there are any number of possible software designs with which to meet that end, and you may ask: why a standalone program, an interpreter, with a database, and written in C++?

2.1 Background

Programs that talk to the shim, the shim itself, the tws client, and its connected and indirect servers together make up a chain where every interior node is both a client and server, and although data feeds back in loops, there is still a clear directionality from the user to the external markets, and so throughout this text I'll refer to the user-facing node in this chain as the downstream, and the external servers as the upstream, so that the chain becomes downstream-shim-tws-upstream.

Api and related IO objects are referred to collectively as *events*, and these events are partitioned into four categories: *commands*, from the downstream to the shim; *requests*, from the shim to the IB tws; *messages*, from the tws to the shim; and *comments*, generated internally by the shim and sent downstream.

The IB tws api is complex, with about 50 request and message events defined, and the largest of these, the place-order request, consisting of more than 50 attributes in the latest version. No formal specification for this protocol language has been publicly released by IB; instead, the source code for a Java program, the *sample client*, is the best available guide to the request and message formats, which I refer to as *wire* formats, reflecting their use in a socket protocol. Note also that the request-message language is fundamentally asynchronous: some data is requested via subscription; the initial response time for order events, though excellent for the common case, is unbounded; and the children of bracket orders may fire at any time.

2.2 Motivation

To suggest the design, it's useful to consider a simpler alternative first. The minimum library interface of transparent wrapper procedures is surprisingly complex, and requires significant additional functionality to be useful; providing these features brings us to the shim's design.

The same approach can be taken to justify the implementation of the shim — start with an easily sketched, naive program architecture, then solve issues of reliability and maintainability to arrive at the shim's implementation architecture.

2.2.1 Why Not a Lightweight Library?

The minimum library interface is a strawman, brought up to be discarded. A library could provide procedures to: open a socket to the IB tws; send and receive handshake information; send data aquisition, order, and account related requests in accordance with the api, and in response to procedure calls; and store the resulting messages as events, to be forwarded, again in response to a library call. This buys us little, and costs us much.

To see how little we gain, consider a transparent standalone program for the IB tws api, one that reads and forwards handshake and other requests upstream, and similarly messages to the downstream, transferring both as is. It adds an extra layer, and no additional functionality whatsoever; clients must have full knowledge of the tws api to use it, and given that, could just as well connect to the IB tws directly.

Note, then, the two problems that arise to the extent that we strive for a "thin" interface: little or no additional functionality, and at the cost either of yet another new binary api on top of the old, or at best a pointless interface layer. Obvious though these problems may be, they're hard to avoid: any programmatic interface to the IB tws api must be carefully designed if it is to simplify access to the api without sacrficing functionality.

The minimum library would replace a protocol api, admittedly binary but still a language of requests and messages amenable to formal specification and open to programs in any language, with request and message procedures for each protocol event, and with an interface every bit as large as the initial IB tws api. Clients need nearly full knowledge of the base api, and there is a maintenance headache as well. The event procedure parameter objects and the primitives to lay their attributes out on the wire must accord with the api, and yet testing those procedures for api conformance is hard, so hard in fact, that other free software projects taking this approach have bogged down. There are twin problems of scale, and efficiency: testing the correctness of the parts is individually straight-forward but collectively onerous; and there is little to the library besides the (large) interface, so that downstream clients are unnecessarily complex, due both to the bulk of library-facing code, and the need to manage low-level api state.

I claim that maintainable interface software to the IB tws api must include table driven specification of the protocol events, with the request send and message receive processing driven by those table entries; that the downstream client must be able to offload knowledge of api details to stored, shared tables, i.e. a database; and that the interface must shield the client from api state, and take responsibility for api event state where feasible. In any case, I refused to wade into the maintenance swamp of one-per-event api send and receive procedures when I started the shim, and I reject it even more firmly today, after having watched other projects struggle with this headache.

2.2.2 What Features Do We Want?

Request abstraction is the first and most critical feature, and the other features either follow directly, or are closely related. The abstractions used to represent otherwise complex request parameter lists must be defined somewhere, and a database system is the best approach.

The mimimum library provides exactly three features besides api access. There is message buffering; a shared address space, though that either at the cost of writing downstream code in the same language, or passing requests and messages through a foreign language api; and procedures that encapsulate the low-level format of the requests, and provide some kind of “recv-msg’ abstraction to perform message parsing.

Considering each of these features in turn: Message buffering is necessary so the IB tws won’t stall, and is part of any useful design. A shared address space might be useful, to allow various parts of the library to be freely reused, though it should be an option to the downstream, not a requirement. The last goal, however, of request and message abstraction, provides the true justification for *any* api interface, and most of the features of § 2.3 serve that end.

2.3 The Shim Architecture

The shim is a command interpreter: it provides a simple command language by which clients can control the more elaborate IB tws api. The simplification is feasible since a database defines contracts and orders, and commands may use keys to stand for the related record values. Request state is tracked by the event router, which is meant to detect errors where feasible, typically by managing timers to detect request failures.

A Command Interpreter

Once simplified via database abstraction, the command layer may be formally defined as a language of events, that language specified to be textual, and the system that accepts this language be implemented as a command interpreter.

The shim is such a system; it accepts text commands from the downstream client, and so simplifies implemenation of that client over the alternative of a direct socket connection; uses a schema-driven parser that type-checks attribute values, constructs events according to those schema, and recovers from message errors using a conservative algorithm that discards only one token per mismatch; and simplifies debugging, exposing the tws api protocol languages by echoing binary and text data to various output channels.

A Database System

The api wire formats imply elaborate contract and order objects, and the permanent definitions for these may be conveniently stored in a database, and referred to by name. This notion of persistent contract and order records is the key to simplifying the command language used by the downstream to make requests. Since for the number of symbols of interest, persistent data can be obtained from the database all at once, there need be no time cost for contract and order database access once past initialization, and in fact the shim has been implemented to slurp the entire database into memory at startup.

Once given the fundamental purpose of supporting shared definitions of contracts and orders between downstream clients and the shim, the database provides in addition a store for historical data, a journal for orders, and a means to dynamically update the shim during process execution. Though such access does cost time, it is either relatively infrequent, as in the case of order journal entries, or can be offloaded to a subprocess, as when history data is saved.

At this time history query writes are still performed by the main process. Although this is not a problem for small, recent queries, e.g., the last hour's worth of 15-second data, it is probably best for the time being to use distinct process instances of the shim for large history queries and orders, respectively. Since, one, there should be no need to perform large history queries as part of a high-frequency trading strategy, and two, up to four instances of the shim can share the same IB tws server, as it accepts up to eight api socket connections, and the shim uses at most two, then it is perfectly feasible to partition bulk data collection and order computation in this way.

An Event Router with Timing

The shim must read from multiple inputs at arbitrary times, at a minimum both the downstream client and the upstream IB tws, and so it can not block for input from either one. Process threads are widely used for such multiplexed IO.

From the viewpoint of the downstream user, the client-shim system necessarily includes time-dependent control. Timeouts, the simplest case, are required if we are to detect lack of response, as when requests are ignored, and other requirements can lead to significantly more complicated reactive control. Such control is difficult to get right, and critically important; consider, e.g., a downstream client program updating a graph from the results of recurring history queries, where the program must either have current data, or signal that the data feed has dried up. Again, process threads are widely used for such reactive control.

Quite often, as here, threads save no time whatsoever, and in that case I prefer to use the `select()` system call, both for IO multiplexing, and as the foundation of reactive control. Once having abandoned the unecessary complexity of threads, a single thread of control and straight forward domain analysis lead naturally to a scheduling/dispatching control object, for the shim named the *Router*.

The Router plays a leading role in the read-route-write loop that provides the high-level control for the shim, controlling either directly or by its data members: the wait for input, using `select()`; the task scheduler, responsible for timeouts and periodic events, such as recurring history queries; and event dispatch, or *routing*, once events have been read from input.

The Router and its included children work efficiently to keep track of time by using multiple time scales of: seconds, 20 millisecond intervals, and the processor clock rate itself. The first time scale is from the downstream application domain, e.g. for timeouts; the second, from the maximum allowed sustained frequency for IB tws requests; and the third is chosen for implementation reasons.

Each of these clocks is read by the shim, and though none is ideal by itself, used together judiciously they provide accurate time-keeping and precise timestamps at low time cost. In brief, the `select()` timeout is set with the IB tws request period; the `localtime()` system call is used, for seconds, once the select call returns, to detect timeouts, and 1 second clock ticks; and the processor read time stamp counter instruction, also `rtsc`, provides fine granularity, both for event timestamps, and to check the select timeout. This test is to ensure that 20 milliseconds have actually passed, since although the processor `rtsc` result may rollover, or lose time depending on power-save settings, it can be trusted in any case not to be fast, and the 1 second timer serves as backup if it has been stopped by sleep.

Conclusion

The shim uses a database to offload complex api request details; a text command language to provide convenient, simple access to the request api; interleaved text output of commands, requests, messages, and endogenous comments; and a single-threaded router architecture for task scheduling and event processing.

Perhaps most critical for maintainability, it uses a table driven parser, with *schema* tables for each kind of event, and where the rows of those tables are attribute type vectors for each particular event, so that much of defining a new request or message event boils down to declaring type symbols for any new attributes, and adding a row of the appropiate attribute symbols, named for the event, to the related schema. The resulting design has proved remarkably robust and flexible.

Part I

Tutorial

Chapter 3

Installation, Setup, and Startup

Once you use the IB tws to connect to the outside world, you become part of the much larger market, formed from traditional exchanges and electronic crossing networks (ECNs), other market customers and brokerages, and of course IB and its systems.

The subsystem consisting of your system together with the IB-provided services it connects to is itself complex, consisting of your IB tws account and its servers, the network links to those servers, one or more instances of the IB tws program that use the net links, a dbms server for the use of the shim, most likely multiple databases installed on the server, multiple roles for the shim, and, what justifies the entire system, the downstream programs you use to access the markets.

Setup for your market access system involves more than just program configuration for the trading-shim program by itself, requiring that you set up a networked system, including in particular the IB tws, the shim, and its database, leaving aside any downstream programs you might also use with the shim. To use the shim, you will need to locate resources § 3.1; compile the shim, and configure the shim and IB tws § 3.2; and create and load the database § 3.3. At this point you should be ready to run the shim § 3.4.

3.1 Resource Requirements

You'll need the trading-shim sources, which are at http://www.trading-shim.com/. In addition, you'll need a network connection (§ 3.1.1); one or more computers setup to run Linux, and with MySQL (§ 3.1.2) and the IB tws (§ 3.1.3) installed; and one or more accounts with IB (again, § 3.1.3). Note that you may have multiple database servers and IB tws accounts; that the trading-shim, MySQL, and IB tws are all network aware; and that, in consequence, you have many possible configuration choices for your market access system (§ 3.1.4).

3.1.1 The Network

The IB tws program needs network access. Although you may well want to have more than one network link to improve system reliability, the configuration and security issues involved with obtaining network connections to the outside world are beyond the scope of this tutorial, and in any case have little impact on shim configuration, since the IB tws serves as an intermediary to the upstream net.

This by no means detracts from the importance of such issues. Although network configuration planning will not be considered furthur herein, it would nevertheless seem prudent to have multiple network connections if you intend to risk real money by making trades with the IB tws. Please also keep in mind that network connections to the IB tws program from the trading-shim require that the tws accept a network socket connection, and that such connections are not themselves secured by password access, so that both your network and your IB tws host must be secure.

3.1.2 The Trading-shim Database

If you look at source files, you'll see near the top a line that includes the following text: *shim: dbms-augmented command interpreter for Interactive Brokers' tws api*. Mention of the dbms is by design. The supporting database is fundamental to use of the shim, since it's the only way that the shim command language can be simplified beyond that used by the tws api.

Currently the only dbms system that works with the shim is MySQL. Although this may change in the future, for now, you must have mysql installed on your system to use the shim. Given a reasonable linux distribution, MySQL is just a package away, so what follows expects you to have a MySQL server available to you.

Current development of the shim takes place against a 5.0 series MySQL, which is important due to its support for foreign key dependency checking with InnoDB tables. The version of your server should not be an issue unless you are running a very much out-of-date system, in which case you should consider upgrading your MySQL server, and realize otherwise that use with a 4.x server is unsupported.

3.1.3 The IB tws

The Trader Workstation (tws) is available from http://www.interactive.brokers.com/. If you have not done so already, download the IB tws system from their site, and install it according to their directions. The IB tws requires Java 1.5 or better, so you will need to have Java installed as well.

Although you do not provide IB tws account information to the shim, you will of course need some form of account information to start the IB tws. You may choose to use the demo account, with user name `edemo` and password `demouser`, though please note that important features of the IB tws api are then either untrustworthy (market data) or do not work (history queries). You may have already set up and funded a trading account with IB, and obtained in return a user name and password that may be used to start the IB tws. Finally, you may have obtained as well a paper account, which allows you to connect with the same market data and history query privileges as your real account, though without the risk to your account balance.

Please, if you have a real account, obtain the paper account it entitles you to as well, and limit your initial use of the shim to environments where the only available IB tws program running is one that has connected to its upstream through the paper account. You alone are responsible for the use of your IB account, including indirect use via the shim. *Avoid using your real account with the shim unless and until you are certain that you understand and accept the related risks.*

3.1.4 Resource Selection

Each component of a market access system provides opportunities for redundancy to provide flexibility and improve reliability, from the network connection, through the tws process, dbms server and database, to the shim process itself. Issues related to configuration planning for redundant hardware resources, and in particular fault tolerant systems [11] are, however, outside the scope of this manual.

I'll focus instead on the flexibility and complexity that stem from multiple accounts, databases, and shim process instances. Note first that the shim has two primary roles, one of collecting market data and history queries, the other of submitting orders and watching the positions that result. These two roles are referred to as *modes*, about which more later, and for now it's enough for them to have names, `data` and `risk` respectively. Recall that for each real account you open with IB, you may also open a paper trading account, and that given such an account pair, and in the event you want to have both open at one time, you will need to run two instances of the IB tws, since each tws process connects to just one account. (Even though using both accounts at once may not be the common case, it may well be useful on occasion, e.g., if development and operational use coincide.) In this case the account information for the paper and real accounts would probably correspond to test and operational data, and distinct order journals should be used for each case. Such operation leads to two shim processes, one for each of the modes, each using a dedicated account and database, and each driven by separate downstream scripts, giving us e.g., the process-database diagram of Figure 3.1. The configuration problem here boils down to ensuring that programs talk to the correct counterpart and share an otherwise private database in common between themselves.

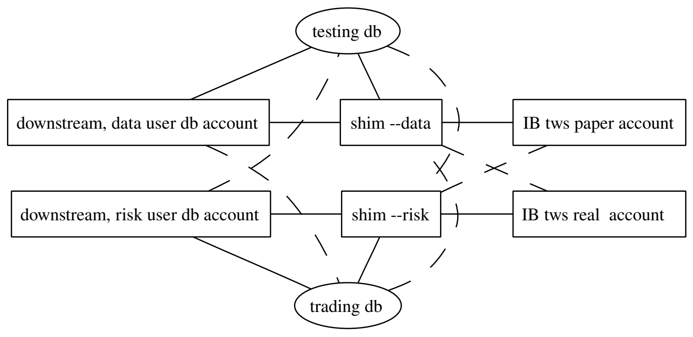


Figure 3.1: Configuration choices

Consider the case where data is being collected under the paper account, and live trading is going on with the real account. The downstream scripts will naturally talk to the correct shim process since they or a parent script probably took responsbility for starting those shims in the first place, which leaves two issues. There is a need for, one, database synchronization between downstream and the shim, and two, configuration control over shim connections to database server and IB tws. Both can be met if information is shared between downstream and shim, either via a shared configuration file, or an event sent from downstream to shim, and in fact either may be used, about which more in § 3.2.3. The result with respect to Figure 3.1 is that only solid edges would connect processes and databases, and that the dashed edges would be elided for this example.

The scenario above motivates one up-front resource choice and a couple of design decisions. There is a real need for multiple IB tws accounts, and even if multiple real accounts prove inconvenient to obtain, this need can be met at least in part by real-paper account pairs. There is a need also for multiple copies of a single database structure, which can be met with sql create-table scripts run in the context of distinct database names; and an easily shared format for the trading shim configuration file, since downstream programs need to read it too. Shim configuration is described in § 3.2.3, and cloning the database, in § 3.3.2.

3.2 Program Configuration

In brief, download, unpack, and compile the trading-shim § 3.2.1; configure the IB tws to accept incoming socket connections § 3.2.2; and configure the connection parameters by which the shim connects to the database and IB tws 3.2.3.

3.2.1 Download and Compile the Shim

If you haven't already, download the sources from the trading-shim site. Choose a directory in which to place the tarball, and unpack the sources, using the tar command, Figure 3.2. E.g., `tar xzpvf shim-070810.tgz`, where `x` means extract; `z`, uncompress; `p`, preserve permissions; `v`, verbose file listing; and `f` refers to the tarball file name, here a daily from August tenth.

```
src$ tar xzpf shim-070810.tgz
src$ ls shim-070810
bin      dep  INSTALL  log       mk.patch  pdf     ROADMAP  src
COPYING  doc  lib      Makefile  NEWS      README  sql      www
src$
```

Figure 3.2: Unpacking the sources

Compile the sources. The Makefile is currently set to use g++ directly, although you may also use `distcc` if you have it installed. From the directory into which `tar` unpacked the sources, in the example above `shim-070810`, from the shell command prompt, simply type `make`. For those uncertain about what to expect from this step, there is more information about the Makefile script in § A.1.

You can verify that the compile succeeded by trying to run the shim, from the shell command prompt, simply typing the (incomplete) command `./shim`. The shim should start and quit, with the usage message of Figure 3.3, in which case you know that the compile succeeded.

```
shim-070810$ ./shim

Usage: shim <mode> [optional feature list]
Modes:

    --help     # print this explanation and list the optional features

               # real modes, requiring access to an IB tws:
    --data     # process subscriptions and log resulting tick stream events
    --risk     # accept full command set, send requests, and log all events

               # test modes, with no connection to the tws:
    --play     # read events from the image file and send text to stdout
    --unit     # for internal use; unstable though otherwise harmless
```

Figure 3.3: The `shim` usage message, after typing `./shim`

3.2.2 Allow Connections to the IB tws

Once you are able to start the IB tws, you will need to configure it to accept connections from the shim, by enabling api clients, and defining the shim's IP address to be trustworthy. From the *configure → api* menu option, check *Enable SocketX and API Clients*; and via the *configure → api → Trusted IP Addresses* dialogue, enter the shim's host IP number to be a trusted address. E.g., for the shim connecting as localhost, the IP number `127.0.0.1` must have been entered as a trusted IP number.

This last step, of defining the shim's host IP address to be trusted, is critical. Otherwise, the accept incoming connection dialogue will pop up, the shim's connection attempt will timeout, and the shim will giveup and quit.

3.2.3 Provide the Connection Parameters

Connection parameters for the shim may be provided via any of the following means, in increasing order of priority: the hardwired default connection parameters in `data.c`; or a configuration file, the `.shimrc` file, in the home directory or current directory; or command input via the dbms and feed commands, triggered by the `init` option.

The connection parameters are the most important of the configuration parameters that a non-programmer user might still reasonably choose to alter. All the parameters are named values, and so are defined by key-value pairs, about which the next section, after which the three means of connection control — config file, init option, and source code patch — will be considered.

The Configuration Key-Value Pairs

The configuration keys known to the shim are listed as the second column of Figure 3.4. Each key has the default value hard-coded into the source code of the shim, and may also appear in a configuration file, in which case the default is overridden. The linkage parameters are also overridden via the `dbms` and `feed` commands if you use the `init` option.

The `DbmsName` and `FeedName` values are currently limited to the default values, fixed as `mysql` and `tws`, respectively, can't be changed, and so may be ignored. They exist mainly as placeholders within the two link commands `dbms` and `feed`, so that the format of the commands allows for additional types of dbms and upstream market feed in the future.

The `DbmsHost` and `FeedHost` values are critical, since without a waiting database server and IB tws program at the indicated machine locations, the shim will quit. You may use either domain names or ip addresses, and if you have mysql and the IB tws running on the local machine, the default value of localhost is what you want.

Category	Key name	Default value	Type
database linkage	`DbmsName`	mysql	string
	`DbmsHost`	localhost	”
	`TableSet`	testing	”
	`UserName`	shim	”
	`Password`	0	”
upstream linkage	`FeedName`	tws	”
	`FeedHost`	localhost	”
	`FeedPort`	7496	”
log file names	`ShimText`	ShimText	”
	`CmdEvent`	cmdinput.txt	”
	`ReqEvent`	shim2tws.bin	”
	`MsgEvent`	tws2shim.bin	”
timeouts (secs)	`InitTime`	20	number
	`FeedTime`	3	”

Figure 3.4: Key-value names and defaults

The `TableSet` value determines which database you are asking the dbms server to give you access to, and if you have run the database setup script of § 3.3.2 as is, the possible values are `testing` and `trading`. The `UserName` chooses the account name within that database, and again, `shim` is an account name provided by the dbms setup script. You need not have a password for your database account unless you so choose — the setup script does not select one — and the default value of zero stands for no password.

The log file names are with respect to the current directory, and will be discussed furthur in § 3.4.3. Of the timeouts, the value of `InitTime` determines how long you are given to enter the `dbms` and `feed` commands if you use the `init` option to provide the link parameters, about which more in the next section, while the `FeedTime` timeout is considered as part of troubleshooting in § 3.6.

In Figure 3.4, the fourth column, labelled “type”, indicates what kind of values may be provided; strings are sequences of non-blank but otherwise arbitrary characters, while the numbers used for timeouts must be non-empty digit strings with neither sign nor decimal point.

For those curious about the use of the string type for the IB tws port number, it is also possible to use a service name, an entry from `/etc/services`, as the value for the `FeedPort` key. If the entry name you choose is not already used in that file, and you have added an new entry at the end of the file, e.g., the line in Figure 3.5, then you could use the service name, here `tws`, for the `FeedPort` value.

Though possible, this is definitely not desirable for the individual user for reasons of security; why would you want to publicize access to your private IB account? It’s conceivable that such a service-based approach might be of use on a corporate lan,

```
# Local services
tws             7496/tcp                        # IB tws
```

Figure 3.5: A possible service entry for the IB tws in `/etc/services`

with individual servers providing paper accounts as a published service, and the ports for real accounts being less well known. The administrators for such a system must understand the security issues involved in publicizing the IB tws port, and such configuration is not supported.

Controlling the Configuration Parameters

If you set up the database to use the default tableset and account names provided by the setup script, about which more in § 3.3.2, and both your database server and IB tws process are running on the local machine (`localhost`), then you may well be able to accept the defaults as is. Othewise, you will have to provide corrected connection values to the shim.

```
shim-070810$ ls -a .shimrc
.shimrc

shim-070810$ cat .shimrc
DbmsHost        localhost
UserName        shim
TableSet        testing
FeedHost        localhost
FeedPort        7496
```

Figure 3.6: The default values for the shimrc file

The shim will attempt to read the `.shimrc` configuration file, hereafter referred to simply (and imprecisely) as the "shimrc" or config file, if it occurs in the home or current directory. Since a default shimrc file is provided with the distribution, then the shim will see the config file if you run it in the same directory where you unpacked the sources, as in Figure 3.6.

The shim program accepts a reasonably flexible format for the config file. As shown in Figure 3.6, the file consists of name-value pairs, and those pairs may be in any order. Not all pairs need be provided; if the shimrc file is incomplete, missing values are filled in by the defaults in the source code, from the file `data.c`. Each pair must be on a line by itself, and the file must have left-aligned pairs only, that is without comments, blank lines, or leading whitespace.

You may also provide the connection parameters as input if you start the shim with the `init` option, that is if that option name occurs on the command line. In that case, the shim will prompt for the `dbms` and `feed` commands, e.g. Figure 3.7.

```
Enter the dbms connect parameters via the dbms command, using the format:
dbms DbmsName DbmsHost TableSet UserName Password;
dbms mysql xps400 testing shim 0;
Ok

Enter the upstream connect values via the feed command, using the format:
feed FeedName FeedHost FeedPort;
feed tws localhost 7496;
Ok
```

Figure 3.7: An `init` option connect dialogue

Note that the program prompts for the commands one at a time, giving the format of the command using the key names to stand for values, then echoing the input, and finally confirming the input syntax as “Ok” after command validation by the parser. Note also that there is at this point no assurance that the connection parameters are valid, only that they provide the expected number of strings.

There is a total of `InitTime` seconds, by default 20, for you to provide input in response to the command prompts, after which the shim will quit. Since for the common case such commands would be sent to the shim via a controlling script, the timeout value isn’t critical, but if you’re experimenting with manual input for the `init` option, add an entry for the timeout to your config file, and feel free to adjust it as needed.

Although you may prefer in the future to use the init option as your standard means of connection parameter input, for the purposes of this guide, and your first attempt at running the shim, please either check that the default connection parameters will work for your configuration, or else edit the default config file as needed, since the text of § 3.4 expects you to either have valid defaults, or a valid config file.

3.3 Database Creation

Given that you have a MySQL database server installed, the following sections explain how to configure it to run in ANSI mode, § 3.3.1, and set up the trading-shim databases `testing` and `trading`, § 3.3.2. The testing database is intended for use with an IB paper account and the regression test scripts that accompany the shim, while the trading database may be used with a real account, if you so choose to take this risk.

3.3.1 Set the Dbms Isolation Level

For safe operation, the shim program requires that the sql server run with the ANSI sql isolation level set to `SERIALIZABLE` in order to prevent phantom reads, that is that the mysql default sql isolation level be changed from the default and oxymoronic level of `REPEATABLE-READ` to the safer level of `SERIALIZABLE`. This is necessary to ensure that transactions are *ACID*, that is atomic, consistent, isolated, and durable. The trading-shim checks for this property, and, by design, will not start without it.

There are various ways to control the isolation level; for mysql it may be set: at the session level; on the server command line; via a command, e.g., "`set global transaction isolation level serializable;`" as by past versions of the setup script; and in the server configuration file.

Partial change, on a per session basis, misses the point that *all* database application write access should be safe and transactional, so that the data can be trusted, and so about which no more. Command operation unfortunately does not persist, so that although placing the command in the setup script solved the problem temporarily, the change was lost after the first server restart. The apparently most straight-forward approach might seem to be to edit the `mysqld` command line in order to add the argument `--sql-mode=ANSI`, so ensuring conformance with a number of `ANSI` sql requirements, including the isolation level. [1] Most systems start the mysql server automatically, when the system starts up, and so doing this would require changing the init or startup script for the `mysqld` program. Since these scripts vary from one platform to the next, and are somewhat opaque even on Linux, this leaves the last option, modification of the server configuration file.

Edit the `/etc/my.cnf` file to insert the line `transaction-isolation = SERIALIZABLE` in the `mysql` stanza. Figure 3.8 gives a patch against that file, and in fact the patch may be found as the file `sql/mysql.iso.patch` in the distribution. Note that applying this change will require root authority.

3.3.2 Create the Databases

Use the setup.sql script in subdirectory sql to create the initial set of user acccounts, the testing and trading databases, and their tables, as well as to load the tables with initial values; or else treat the setup script as a starting point to see how to build a database according to your own naming and security policy, and use the script create.sql, also in the sql subdirectory, to create and load the database tables.

[1] If you choose to set the sql-mode via `sql-mode=ANSI`, whether from the command line or in the configuration file, be aware that this approach is neither recommended nor supported. It disables a number of mysql extensions to the syntax of `ANSI` sql, including the freedom to use double quotes in place of single quotes for strings that have embedded single quotes, as with some of the database symbol load scripts. Perhaps more critically, there are also reports of additional restrictions on the use of sql reserved words as identifiers.

```
*** my.cnf-OLD 2007-04-20 16:10:13.000000000 -0400
--- my.cnf 2007-04-20 16:10:13.000000000 -0400
***************
*** 9,14 ****
--- 9,18 ----
  user=mysql
  basedir=/var/lib

+ [mysqld]
+ transaction-isolation = SERIALIZABLE
+
+
  [mysqld_safe]
  log-error=/var/log/mysqld.log
  pid-file=/var/run/mysqld/mysqld.pid
```

Figure 3.8: Changing `my.cnf`

Using the `setup.sql` Script to Create the Databases

More precisely, and given that you accept the naming and security choices in the setup.sql script, do the following:

1. Obtain the mysql root account password; and note that the mysql root account is distinct from the linux OS root account. Login access for the mysql root account is often restricted to prevent network logins, and in that case you will also need login access to the machine where the database server runs as well, although here an ordinary user account should work just fine.

2. Ensure that the shim tar ball has been unpacked on the machine from which you intend to login to the mysql database, or else copy the subdirectory sql with all its contents and subdirectories to that machine. For reasons already noted in (1), the machine you use may well be the one that the mysql server itself runs on.

3. From the directory sql, and given read access to the sql directory, its subdirectories, and files, start the mysql interpreter as the [mysql] root user:

   ```
   mysql -u root -p
   ```

   The -p option will cause the mysql program to prompt you for the mysql root password, which you must have obtained previously, in (1).

4. You should see text approximately like the following, indicating that you have obtained a mysql interpreter prompt:

   ```
   mysql -u root -p
   ```

```
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 928 to server version: 5.0.27

Type 'help;' or '\h' for help. Type '\c' to clear
the buffer.

mysql>
```

5. Enter the following command in order to create user accounts, the testing and trading databases, create the tables for those databases, and populate them with initial values. You should see *many* status messages fly by (on the order of a couple of hundred; in the dialogue below, the ellipses indicate many rows scrolling past), and ending with the mysql prompt:

```
mysql> source setup.sql

... ... ... ... ... ... ... ... ... ...

Query OK, 12 rows affected (0.00 sec)
Records: 12  Duplicates: 0  Warnings: 0

Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

Query OK, 0 rows affected (0.00 sec)

mysql>
```

6. To verify that the testing and trading databases have been created, you may list the databases (note the semicolon):

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| lost+found         |
| mysql              |
| testing            |
| trading            |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

7. To verify that tables have been created, you may set the database to testing, and list the tables (again, ellipses indicate elided material):

```
mysql> use testing
Database changed
mysql> show tables;

+--------------------+
| Tables_in_testing |
+--------------------+
| AtomTag            |
| BarSize            |
| Bool               |

...  ... ... ...  ...

| Underlying         |
| Version            |
| Volatility         |
| WatchSets          |
+--------------------+
50 rows in set (0.00 sec)

mysql>
```

8. Type quit at the prompt to end the mysql session:

```
mysql> quit
Bye
sql$
```

At this point the shim databases have been created, tables for those databases created and populated with default values, and user accounts created for the four roles of shim program operation, downstream data collection, downstream orders, and offline maintenance and programming; the account names are listed in Table 3.1.

Name	User	Intended Role
`shim`	program	database connection by the trading-shim program
`data`	program	downstream programs that run the shim in data mode
`risk`	program	downstream programs that run the shim in risk mode
`code`	person	interactive access and database maintenance

Table 3.1: Default user accounts

Please understand that although the testing database has default order information, so that the regression scripts can demonstrate simple reversing orders, the trading database has not yet been populated with the order lineitems referred to by downstream order commands, about which more in Chapter 4. Although databases have been set up at this point, you should not yet be trying to submit orders from downstream programs to the shim for transmission through the api to the IB tws.

Using the `create.sql` Script to Recreate the Tables

You may find it convenient to use the create script as part of a custom database setup process, e.g., if you would rather create the databases via manually entered commands instead of running the setup script mentioned previously. You do not need to run the create program if you have already run the setup script, as the tables have already been created as part of that process.

You should also feel free to recreate the testing database tables as needed, and in any case, you will need to do this to upgrade your table design as new versions of the shim are released. The create.sql script, in particular, is provided to enable you to rebuild the tables without again obtaining mysql root access; you should only have to run the setup.sql script once, while the create.sql script may be executed many, many times during development.

If you are considering table recreation after some period of program operation, you should realize that the OrderJournal table contains accounting information about the orders, if any, that you have made, so that if you use a real IB account with the trading database, as opposed to the paper account that should be used with the testing database, you will need to first preserve the logical contents of that table.

In the case that you wish to recreate the tables and load their initial values, recall that the setup script above created a user account `code`, meant for general purpose programming and maintenance of your trading-shim databases. Use this account to connect, and run the create.sql script, as illustrated below. Note that in the example here the account is not protected by a password; and again, the ellipses indicate missing material, as many rows scroll by, with values similar but not identical to those for the "`source setup.sql;`" step above.

```
sql$ mysql -u code testing
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 928 to server version: 5.0.27

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> source create.sql;

... ... ... ... ... ... ... ... ... ...

Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

Query OK, 1 row affected (0.00 sec)

Query OK, 3 rows affected (0.01 sec)
```

```
Records: 3  Duplicates: 0  Warnings: 0

mysql>
mysql> quit
Bye
sql$
```

3.4 Shim Startup

Given a co-operative mysql server and IB tws, properly compiled shim, and valid connection parameters, whether via defaults or config file, at this point you should be able to start the shim (§ 3.4.1). Perhaps more to the point you want to see how it works (§ 3.4.2) and in particular view the output (§ 3.4.3).

3.4.1 Run the Shim

Recall from § 3.2.1 that as part of startup, the shim must have a mode value on the command line, typically either `--data` or `--risk`. For now we'll use `--data` mode, which is well suited to collecting market data, but does not allow you to submit orders. Although there are other modes, and also a number of optional features, we'll put those aside for now, leaving them for the next chapter. From the directory where you unpacked and compiled the shim, enter the command `./shim --data`.

```
shim-070810$ ./shim --data

The trading shim has finished program initialization, including the
construction of successful connections to the database and IB tws.

quit;
```

Figure 3.9: Starting and stopping the shim

Ideally you should see the connect message of Figure 3.9, after which you can stop the shim by entering the `quit` command. If you mistype it, just re-enter it, each time on a line by itself, ending with the semicolon, and lastly pressing return. Or, feel free to type `Ctrl^C`; it won't hurt anything here.

3.4.2 Run the Test Scripts

The primary test script, `bin/regress`, can be run either as part of making the shim, via the make target `make test`, or directly. It runs the shim in data mode, so that no orders can be made, and sends a variety of subscription and query requests to the IB tws.

From the directory where you unpacked the shim, type the command `bin/regress`, as in Figure 3.10. The shim should connect, and after the connect messages bits and pieces of cruft may appear on the screen; these would be debug output that occur as part of ongoing development and are sent to the `stderr`, and may for the most part be ignored. Much more to the point, however, an additional `konsole` window should pop open on the screen and remain open until the shim exits.

```
shim-070810$ bin/regress

The trading shim has finished program initialization, including the
construction of successful connections to the database and IB tws.

Call   1
ScimInputContextPlugin()
Hcmd:   3  15
Info:  0xb713effc  0xb6fcc424
Info:  0xb714731c  0xb6fcc6b4
src$ ~ScimInputContextPlugin()
```

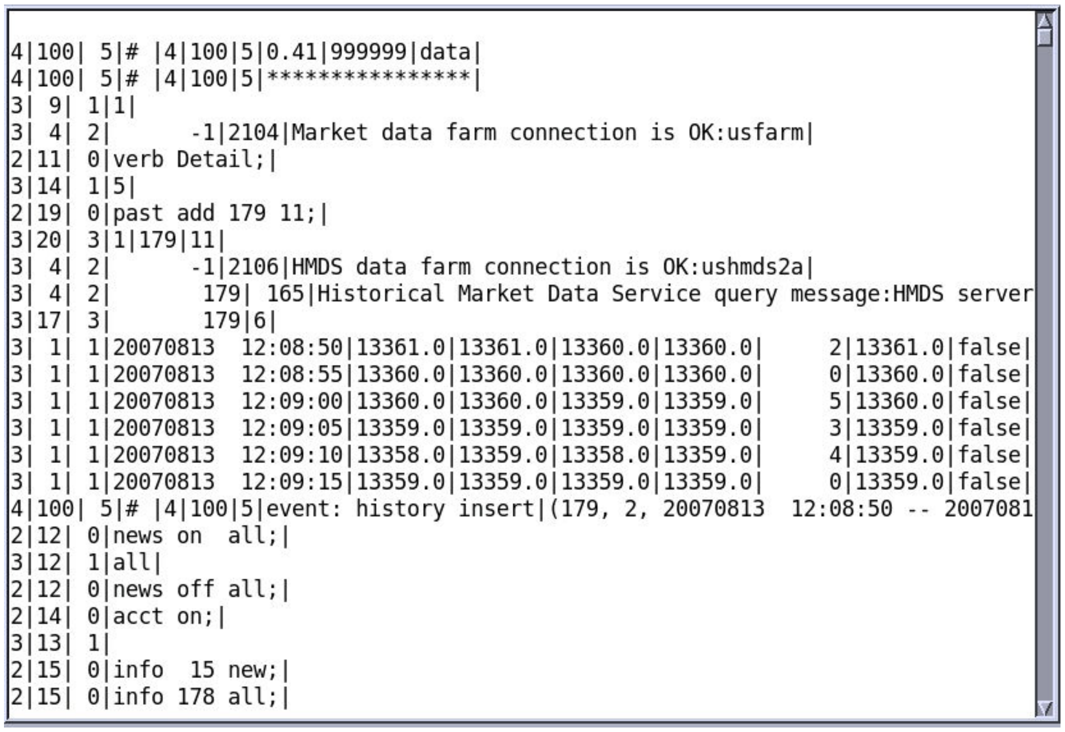
Figure 3.10: Running the primary test script

Figures 3.11 and 3.12 are screenshots from the begining and end of the traced output from `bin/regress` script. The text in the figures consists in essence of the log output file `ShimText` echoed to a `konsole` window by the Unix utility `tail`, and you can look at the scripts `bin/tail.window` and `bin/log.filter` to see what is happening.

In Figure 3.11 we see the progran banner, market data and history farm status messages from the IB tws, a history query and its answer, the status message as that result is inserted into the database, a news check, and a subscription to account data.

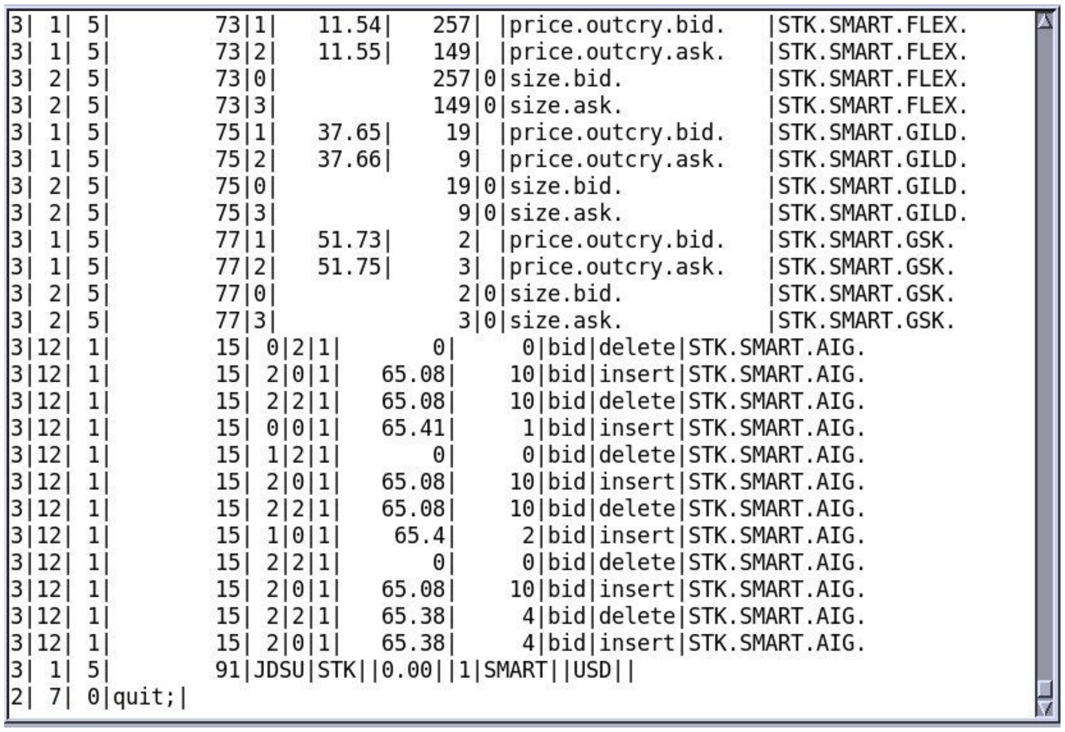
Figure 3.12 consists of tick data for a number of symbols that were subscribed to earlier in the session via the bulk subcription command, the `quit` command, and one lonely tick message bringing up the rear, since the quit is not acted on until all events from the current time tick have been logged.

Although there are a mass of details not yet considered, you can see that the text log is in essence a catenation of the command, request, and message events that occur during a shim session.



```
4|100| 5|# |4|100|5|0.41|999999|data|
4|100| 5|# |4|100|5|****************|
3| 9| 1|1|
3| 4| 2|      -1|2104|Market data farm connection is OK:usfarm|
2|11| 0|verb Detail;|
3|14| 1|5|
2|19| 0|past add 179 11;|
3|20| 3|1|179|11|
3| 4| 2|      -1|2106|HMDS data farm connection is OK:ushmds2a|
3| 4| 2|       179| 165|Historical Market Data Service query message:HMDS server
3|17| 3|       179|6|
3| 1| 1|20070813  12:08:50|13361.0|13361.0|13360.0|13360.0|     2|13361.0|false|
3| 1| 1|20070813  12:08:55|13360.0|13360.0|13360.0|13360.0|     0|13360.0|false|
3| 1| 1|20070813  12:09:00|13360.0|13360.0|13359.0|13359.0|     5|13360.0|false|
3| 1| 1|20070813  12:09:05|13359.0|13359.0|13359.0|13359.0|     3|13359.0|false|
3| 1| 1|20070813  12:09:10|13358.0|13359.0|13358.0|13359.0|     4|13359.0|false|
3| 1| 1|20070813  12:09:15|13359.0|13359.0|13359.0|13359.0|     0|13359.0|false|
4|100| 5|# |4|100|5|event: history insert|(179, 2, 20070813  12:08:50 -- 2007081
2|12| 0|news on  all;|
3|12| 1|all|
2|12| 0|news off all;|
2|14| 0|acct on;|
3|13| 1|
2|15| 0|info  15 new;|
2|15| 0|info 178 all;|
```

Figure 3.11: Screen shot of history, news, account data and contract data queries



```
3| 1| 5|        73|1|   11.54|   257| |price.outcry.bid.   |STK.SMART.FLEX.
3| 1| 5|        73|2|   11.55|   149| |price.outcry.ask.   |STK.SMART.FLEX.
3| 2| 5|        73|0|           257|0|size.bid.           |STK.SMART.FLEX.
3| 2| 5|        73|3|           149|0|size.ask.           |STK.SMART.FLEX.
3| 1| 5|        75|1|   37.65|    19| |price.outcry.bid.   |STK.SMART.GILD.
3| 1| 5|        75|2|   37.66|     9| |price.outcry.ask.   |STK.SMART.GILD.
3| 2| 5|        75|0|            19|0|size.bid.           |STK.SMART.GILD.
3| 2| 5|        75|3|             9|0|size.ask.           |STK.SMART.GILD.
3| 1| 5|        77|1|   51.73|     2| |price.outcry.bid.   |STK.SMART.GSK.
3| 1| 5|        77|2|   51.75|     3| |price.outcry.ask.   |STK.SMART.GSK.
3| 2| 5|        77|0|             2|0|size.bid.           |STK.SMART.GSK.
3| 2| 5|        77|3|             3|0|size.ask.           |STK.SMART.GSK.
3|12| 1|        15| 0|2|1|       0|     0|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 2|0|1|   65.08|    10|bid|insert|STK.SMART.AIG.
3|12| 1|        15| 2|2|1|   65.08|    10|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 0|0|1|   65.41|     1|bid|insert|STK.SMART.AIG.
3|12| 1|        15| 1|2|1|       0|     0|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 2|0|1|   65.08|    10|bid|insert|STK.SMART.AIG.
3|12| 1|        15| 2|2|1|   65.08|    10|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 1|0|1|    65.4|     2|bid|insert|STK.SMART.AIG.
3|12| 1|        15| 2|2|1|       0|     0|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 2|0|1|   65.08|    10|bid|insert|STK.SMART.AIG.
3|12| 1|        15| 2|2|1|   65.38|     4|bid|delete|STK.SMART.AIG.
3|12| 1|        15| 2|0|1|   65.38|     4|bid|insert|STK.SMART.AIG.
3| 1| 5|        91|JDSU|STK||0.00||1|SMART||USD||
2| 7| 0|quit;|
```

Figure 3.12: Screen shot of market and market depth data, ending with shim exit

3.4.3 Locate and View Output

3.5 The Shim Command Line

The command line for the shim consists of the program name itself, presumably `shim`, and spelled out as `./shim`, about which more later; the required mode, § 3.5.1; and following the mode, a possibly empty list of options, § 3.5.2. The location of your output (§ 3.5.3) is controlled in part by those options, and also by how and where you run the shim, § 3.5.3.

3.5.1 Choosing the Mode

A command mode is required for normal operation. If you leave it off the command line, you'll get the mode message of Figure 3.13 to explain your available choices. I'll consider the real modes in this section, and the help mode in the next one, leaving aside the test modes as of use only to developers.

```
Usage: shim <mode> [optional feature list]
Modes:

    --help      # print this explanation and list the optional features

                # real modes, requiring access to an IB tws:
    --data      # process subscriptions and log resulting tick stream events
    --risk      # accept full command set, send requests, and log all events

                # test modes, with no connection to the tws:
    --play      # read events from the image file and send text to stdout
    --unit      # for internal use; unstable though otherwise harmless
```

Figure 3.13: The mode message

For all productive use of the shim you'll use either `--data` or `--risk` mode.

Data mode protects you from unintentional orders, since the commands used for orders are not accepted by a shim running in that mode; it literally can not understand them, and will treat such as a syntax error. Data mode has the virtues of its defect, since denial-of-service issues that might be critically important as orders are transmitted become much less troublesome when all that is at stake are market data subscriptions and outstanding history queries. E.g., the bulk subscription command, `load`, which is convenient for rapidly changing from one set of subcriptions to another, is best suited to data mode; even if you run up against your market data subscription limit by mistake, you can compensate for lost access to a particular symbol by an ad hoc history query.

As mentioned previously, you can submit orders to the IB tws in risk mode, and this is the only mode that accepts the commands to submit and cancel orders. All other commands are also supported in this mode, including the bulk subscription load command, since you might need to monitor market conditions over some large, rapidly changing set of symbols. You should be careful, however, to control the number of market data subscriptions that you have in place at any one time, since IB limits you to at most 100 per account. This and other IB tws resource limitations are listed in Figure 3.14, and each of these limits is enforced as well by the shim, to avoid disconnect-reconnect delays.

Limit	Resource
50	api requests per second
100	market data (tick) subscriptions, at any one time
3	market depth (level 2) subscriptions, ditto
6	history queries per minute
1	history queries in flight at any one time

Figure 3.14: Resource limits to the IB tws

The shim feeds requests to the IB tws at most once every 20 milliseconds, to avoid breaking the api request rate limitation. It counts market data and market depth subscriptions already submitted, and queues new ones up until the number of active subscriptions has declined to allow the requests to be sent. Only one history query is sent out at a time, and at most six of those will be submitted per minute. In each case over-eager downstream commands that would surpass some rate or counted limit are simply queued up until conditions allow later submission.

For market data, and in the absence of subscription cancellations, the resulting delay is unbounded, so that careless use of the load command might easily block data subscriptions for critical, position-related contracts. It is up to the user or programs that control the shim — i.e., *you* — to make the resource allocation decisions for history queries and market data subscriptions that are necessary to stay within the limitations that IB has placed on their system. Otherwise, early requests will starve later ones.

3.5.2 Choosing Options

If you choose `--help` mode, i.e. with the command `./shim --help`, then you'll see not only the text of Figure 3.13, but also, following it, the explanation of the command line options, as in Figure 3.15.

The first of these, the `cmds` option to `--help` mode, displays also a brief explanation of the shim command set, about which more in Chapter 4. Of the other options, most provide some kind of control over shim output. In particular, the `file`, `cout`, and `logd` options select logging to a file, the standard out, or the system logger, respectively. The file option is the default if none is given, and it is also enabled if the `pane` option is used. The output options may be combined, so that if, for instance, you want to send the event text to both a file, and also the system logger, feel free.

```
          # Options may be listed following the mode.
          # help mode only:
cmds      # describe the syntax and semantics of the shim commands

          # real modes; the output options write all events to the:
file      # file with name also the ShimText name-value pair value
cout      # standard output -- warning, sprays all over the screen
logd      # system logger, to provide routing via /etc/syslog.conf
          # output options may be combined and file is the default

          # other real mode options:
init      # prompt for the dbms and link commands during startup
pane      # pop a konsole windowpane to scroll cmds, reqs and msgs
load      # load SubRequest at startup even without a load command
save      # cmds-reqs-msgs image file partition for later playback

          # test mode only:
fast      # cut startup time; this saves just one second, works with
          # Linux only, and is used primarily to speedup the offline
          # parts of the test suite
```

Figure 3.15: The help message for options

Two other real mode options affect message output. The `pane` option pops up a kde `konsole` window, and scrolls the text of all the events there, as they are written to the log file. The `save` option is used in debugging, and copies command, request, and message data to the files `cmdinput.txt`, `shim2tws.bin`, and `tws2shim.bin`.

Of the remaining parameters, the `init` option triggers prompts for the connection parameters, as already explained in § 3.2.3, and in particular Figure 3.7, while the `load` option forces a read of the bulk subscriptions table at startup, so that market data subscriptions may be sent to the IB tws as if via some batch job, without any command input whatsoever.

3.5.3 Deciding Where to Run the Shim

3.6 Troubleshooting Connect Problems

Chapter 4

Using the Shim

This chapter introduces the reader to the shim command set, and documents the format of the events that result. It includes both introductory sections for the new user as well as detailed explanations and tables for the downstream script programmer.

Recall from Chapter 1 that shim and api protocol language statements are referred to collectively as *events*, and these events are partitioned into the four categories of: *commands*, from the downstream to the shim; *requests*, from the shim to the IB tws; *messages*, from the tws to the shim; and *comments*, generated internally by the shim and sent downstream. The command language is explained in § 4.1, while the related requests and resulting messages, collectively *api protocol events*, are covered in § 4.2. The new user will probably want to treat the command set as an introduction to the api protocol, and read selectively in § 4.1, leaving the details of § 4.2 for later, while the downstream programmer should expect to cover all of § 4.1 and § 4.2.

4.1 The `trading-shim` Command Set

Commands consist of: an initial verb; the parameters, if any; a terminating semicolon; and finally a newline. They may be usefully divided between those that control the shim, § 4.1.1, and trigger api requests, § 4.1.2.

Recall from § 3.5.2 that `--help` mode supports the `cmds` option, which adds a brief explanation of the shim command set to the initial description of the modes and options. That text is recapped in Figures 4.1 and 4.2.

4.1.1 Commands to Control the Shim

- initialization

```
Command function -- a brief guide:

    help     Display the text you are reading now

    ping     Check connectivity between downstream and the shim
    next       "     "            "     the shim and the tws
    read     Add newly inserted entries of the database to the shim
    load     Load the SubRequest table, adding and deleting subscriptions
    list     List current subscriptions

    wait     Add an n-second bubble to the shim's request queue
    wake     Clear the bubble counter, restarting   "       "
    quit     Terminate the shim (after waiting on bubble, if any)

    verb     Set the tws log level
    news     Subscribe and unsubscribe to news bulletins
    open     Query for open orders
    acct       "    "  the (label, value, currency, account) tuple list
    data       "    "  contract data

    tick     Subscribe and unsubscribe to market data
    book       "       to market depth (cancel currently broken)
    past     Query for historical data (cancel not yet implemented)

    wire     Create, modify, submit or cancel an order
    cash     Exercise an option

The following command verbs are synonyms:

    acct and account
    past  "  history
    wire  "  order
    cash  "  exercise

Note that the load and individual subscription commands overlap in
functionality, and that it is safest for now to use one or the other type,
but not both, within a session.  That is, if you use load, consider avoiding
tick, book, and past, and vice versa.
```

Figure 4.1: The Command Function Help Text

```
Command notation -- a brief guide:

Commands begin with the command verb, followed by the parameters, if any,
and terminated -- always -- by a semicolon.

The simplest command verbs have no parameters:

    help    next    list    wake
    open    read    load    done

The following command verbs allow whitespace and arbitrary comment text
to follow up to the terminating semicolon -- don't forget it!

    ping    quit

The parameter syntax for most of the remaining commands is minimal:

    wait N;
    verb Level;
    data Cid;
    tick Req Cid I;
    book Req Cid I;
    past Req Cid I;
    cash Act Cid Q Force;

    N    : the number of seconds
    Q    :  "  quantity
    Cid  :  "  contract id, a database uid attribute value of Contract
    I    :  "  configuration id, a database table uid, one of, by verb:
           tick: TickConfig
           book: DepthLimit
           past: PastFilter
    Level: one of (System, Error, Warning, Info, Detail)
    Req  :  "  "  (add, del)
    Act  :  "  "  (exercise, lapse)

The order command, due to the number of api parameters, is more complicated.
Much of the complexity is hidden in the LineItem record, but modifiable
parameters must be provided on the command line.

    wire(Oid,Type,Op,Q,P,Aux,T);

    Oid : the line item id, a database uid attribute value of LineItem
    Type: an order type, e.g., MKT, LMT, STP, or TRAIL
    Op  : one of (Create, Submit, Modify, Cancel)
    Q   : the quantity
    P   :  "  limit price
    Aux :  "  auxiliary price
    T   :  "  timeout (just a dummy for now, not yet used)

For examples, and in any case if this crib sheet is not sufficient,
see the regression tests, in particular the program text of bin/includes.

Note that the execution report, market scanner, option modelling, and all
financial advisor related requests and messages are not yet supported.
Please subscribe to the mailing list and let us know if you need these
features, see:

    http://www.trading-shim.org/mailman/listinfo
```

Figure 4.2: The Command Notation Help Text

- runtime
  - control
  - database

4.1.2 Commands that Trigger Requests

- api protocol
  - risk
  - data
    - ad hoc
    - itemized

4.2 IB tws api Protocol Events

4.2.1 Requests to the IB tws

4.2.2 Messages from the IB tws

The sample client sources provide the best available guide to the requests and messages that make up the language of api events. The files of particular interest are `EClientSocket.java` for requests, and `EReader.java` for messages, both in the directory path `IBJts/java/com/ib/client` once the sources have been unpacked. The IB tws api documentation has also been useful, especially in determining the domain values for the various event attributes; start at IB's main page, pull down SOFTWARE, and drill down through FIX/API to one of the API topics, in particular User's Guide, Beta Notes, or Release Notes.

Under the IB tws api protocol, both requests and messages begin with a numeric code, and consist of null-terminated strings.

4.2.3 Requests

The initial codes for the IB tws api request events are listed in Table 4.1.

Tag	Description	Class name
1	request market data	ReqMktData
2	cancel market data	EndMktData
3	place order	PlaceOrder
4	cancel order	CancelOrder
5	request open orders	OpenOrders
6	request account data	AccountData
7	request executions	Executions
8	request next id	RequestIds
9	request contract data	ReqConData
10	request market depth	ReqMktBook
11	cancel market depth	EndMktBook
12	request news bulletins	ReqBulletin
13	cancel news bulletins	EndBulletin
14	set IB tws log level	SetLogLevel
15	request auto open orders	AutoOpens
16	request all open orders	AllOpens
17	request managed accounts	ManagedAccts
18	request financial advisors	FinAdvisor
19	replace financial advisor	ReplaceFa
20	request historical data	ReqHistory
21	exercise options	ExerciseOpts
22	request market scan	ReqScanSub
23	cancel market scan	EndScanSub
24	request scan parameters	ReqScanParms
25	cancel historical data	EndHistory

Table 4.1: IB tws api request names and codes

Table 4.2: IB tws api message names and codes

4.2.4 Messages

4.3 The Downstream Text Protocols

Downstream programs talk to the shim via a simple verb-operand command language, § 4.3.1, while the text format used for shim output works to encapsulate all the various event types, § 4.3.2.

4.3.1 The Command Language

4.3.2 The Shim Output Format

Since the shim output language is large, with 80 event types currently defined, it is structured to simplify message selection by the downstream, with events having a common prefix, including numeric event codes suitable for switching. Following the prefix, the rest of the event text is next echoed with near literal precision, after which selected message types have appended context.

The Output Prefix

The output prefix is that initial portion of each output record that has the same structure as every other, and consists of six fields. The first three are generated by the shim, possibly with help from the system logger, but without reference to the event type. The last three, the `src-tag-ver` triple, are event specific; in the case of requests and messages, the `tag` and `ver` are defined by the IB tws api.

Table 4.3 gives an example of the output prefix as captured by the system logger; the text of the table differs from that actually logged in that the vertical bars that originally separated the fields are here indicated by column rules, and a heading has been added for clarity.

There are actually two forms of output prefix, as the shim performs IO directly to a file descriptor, or this task is delegated to the system logger via the `syslog()` standard library call, in which case that procedure prepends additional text. Here the prepended text consists of the date, formatted time, hostname, and following colon. Note that although the textual value of the initial field differs, the number of columns in the table, and number of fields that would be counted by splitting on vertical bars, remains the same.

Considering the leftmost three fields, the process id, here `27865`, can be used to group log records by session. The seconds since midnight and μsecs values (see § 8.6, § 9.2.1, and § C.2.3) are primarily of use in benchmarking performance, although they in addition provide a unique key for the events of that day. Given 37674 seconds since

prepended log text and process id	seconds	μ secs	src	tag	ver
May 8 10:27:54 pippin : 27865	37674	1010583	4	100	5
May 8 10:27:54 pippin : 27865	37674	1010590	4	100	5
May 8 10:27:54 pippin : 27865	37674	1010598	4	100	5
May 8 10:27:54 pippin : 27865	37674	1010643	3	9	1
May 8 10:27:54 pippin : 27865	37674	1343459	3	4	2
May 8 10:27:54 pippin : 27865	37674	1343500	2	2	0
May 8 10:27:54 pippin : 27865	37674	1343515	2	7	0

Table 4.3: Examples of the output format prefix

midnight and noting that $37674 = 60 \times (60 \times 10 + 27) + 54$, we confirm that the session occurred at 10:27:54 am. Although duplicative here, direct output to a file would lack the logger prepend, so that the seconds field is not in general redundant.

The three rightmost fields are probably more widely used than the first three, since together they map one-to-one to the event type, so that they enable event type switching, about which more in the next section.

Message Switching

The `src` code partitions the events into categories such as command, request, message, or comment; the `tag` matches according to event; and, for events originally occurring on the upstream side of the shim, the `ver` code corresponds to the IB tws api request or message version. A common idiom for downstream scripts splits output text on vertical bars, and then switches on the source and tag codes.

Since the `tag` and `ver` values for requests and messages are defined by the IB tws api, their values are found back in Tables 4.1 and **??**. The request and message tags are every bit as stable in the design for the shim as for the IB tws api, so hardwiring them into downstream scripts is probably reasonable.

The command and comment codes, on the other hand, are specific to the shim, and have changed a number of times over the course of development. For this reason, I recommend that downstream developers match directly on the command verb text if they need to switch on command types, Table **??**. The internally generated events are the least stable part of the event design **??**, and so are not covered here yet.

The Encapsulated Body

For flat events, the attributes in the body of a log format record map one-to-one to the fields of the related command, request, message, or comment. History queries are flattened, so that the header has a distinct log record, and the detail lines follow immediately, one to a line. In some cases data is white-space formatted to aid column alignment, and the dynamically chosen tick and order ids are mapped back to the related contract or order lineitem index key. Otherwise, request and message event data is printed exactly as the characters appear on the network socket.

Chapter 5

Adding Info to the Database

Currently, the only section in this chapter, below, is a mailing list post for a question that has come up a number of times now, of how people in Europe can add new symbols and contracts to the database. In what follows, mixedcase names are database tables, and paths are relative to the directory sql of an unpacked tarball. The post is motivated with a rhetorical question:

Where is the symbol and contract data, and how does it get there?

5.1 Adding Symbols and Contracts to the Database

In [very] brief, there are three primary source tables that are loaded, and these are then used to populate the table Underlying. Then the table Symbol is loaded, first from Underlying, and second from ProductMap. Finally Symbol is filtered by LocalMap into Contract to give the set of contracts.

The contracts are the entities that can be subscribed to for current market data, queried for history data, and in some cases bought and sold.

5.1.1 Data for the table Underlying

In reading the load.sql script, focusing on the primary symbol data, and working backwards from goal to means, note, 1st, the insert statements in load.sql that populate the Underlying table; 2nd, the table names those insert statements select from, and 3rd, the data sources that are src'd by load.sql in order to populate the three primary source tables.

In brief, Underlying is populated from Currency, Miscellany, and Stock, and those src'd from req/Currency.sql, mod/Miscellany.sql, and mod/Stock.sql. At this point Underlying is filled with type, home, name, and description quadruples (see syms.sql for the create table statement for Underlying). Looking at the values for the src'd tables, Underlying can be seen to be a union of security, more precisely, product, tuples, labelled by type of security.

5.1.2 Data for the table Symbol

Once given the contents of Underlying, inserts in the load.sql script fill Symbol from Underlying and ProductMap, respectively.

Underlying fills Symbol

Script load.sql first copies from Underlying to Symbol, giving us an initial set of symbols. Keep in mind that a symbol is an abstraction, not just a string like SUNW, or JAVA. Those are names, and a symbol has in addition a type, e.g., stock or future; and a home, since different exchanges may use the same name for different symbols, that is underlying products.

Although Symbol now has an initial population originally derived from: the primary currencies supported by IB, USD, AUD, CAD, CHF, EUR, GBP, HKD, JPY, MXN, and SEK; miscellaneous index, commodity, and paper names from Miscellany; and the stocks from, you guessed it, Stock, currently mostly NYSE and NASDAQ, and some AMEX — still, there are any number of possible derived products not yet in symbol. In particular, for the entries in Miscellany, we may want to use the same name for related indices, futures, and options.

ProductMap adds to Symbol

The values in the three primary source tables are treated as base case values, and the ProductMap table allows us to multiply them into new Symbol tuples as desired. Note from the create table statements in syms.sql that Underlying and Symbol have nearly the same structure, being in essence type-exchange-name triples. Each product mapping gives directions for creating a new, derived symbol triple from an underlying, and incidently stands for the derivative relationship between the two, since that is not directly visible except by matching on the long name comment attributes, a tedious and fragile business at best.

5.1.3 Data for the table Contract

Recall that the table LocalSet controls how contracts are created from symbols. Note that as delivered, the load script for the database results in, as this is written, more than 5000 symbols, but less than 200 LocalSet tuples and contracts. This indicates one of the reasons for LocalSet, to eliminate uninteresting symbols from contract consideration; another is to provide the additional information needed to fully define contracts.

The role of LocalSet

The entries in LocalSet consist of key values related to key indices in Symbol, and three additional attributes, the key values for route, unit of currency, and variant part tag index.

The first two are straight-forward, and have already been encountered in a different context, as symbol home exchanges, and currencies to be traded. Here the route refers to the floor or ECN where the trade is to take place or cross, while the currency is used as a unit of denomination.

The tag index varies in interpretation with the security type, and as it implies additional variant part data such as expiry and right. For stocks and indices, where there is no variant part, the tag is always zero, while for futures and options it stands for a tuple index into either of the tables FutDetail and OptDetail.

Adding new contracts

The proper way to add contracts to the Contract table is by adding new data to the source and intermediate load files from which it was originally defined, and reloading the database by running the create.sql script. Note that for production use, you will want to first unload history data records and order journal entries in order to save them, since in reloading the database tables are first dropped, then recreated, and finally reloaded. In what follows, when I speak of adding an entry to a table, I mean that an entry is added to the load file, so that the changes you make will persist the next time you recreate the database.

In brief, the preferred way to add a new contract to the Contract table is to add an entry to the LocalSet load file mod/LocalSet.sql, add supporting entries as needed to primary and intermediate load files used to populate Symbol, and then recreate the database.

In the worst case, for a new Underlying not yet appearing in the database, you will have to add to one of Currency, Miscellany, or Stock, then, if deriving from that, to ProductMap, and in any case, add to LocalSet. If the underlying already exists, but some derivative of interest does not, then you need add to only ProductMap and

LocalSet; and if the security of interest occurs in Symbol, you only need select it by adding to LocalSet.

To summarize, in order to add a new contract, add a new LocalSet load file entry, and add as well supporting entries as needed to meet foreign key constraints, to the source load files req/Currency.sql, mod/Miscellany.sql, mod/Stock.sql, and mod/ProductMap.sql; and then, from the directory sql, and in the mysql interpreter, source the create.sql script.

The table LocalSet, although used by the load script, is never referred to by the shim, and you are free to add or delete entries up to the point the database is used for production, after which existing entries must not be deleted, and adds should be at the end.

Part II

Reference

Chapter 6

Languages and IO

One of the primary requirements for the shim is that it shield the downstream from the binary IB tws protocol by converting that format to a convenient textual form. There are then at least two protocol languages that the shim must speak, binary for upstream and text for downstream.

Once given a textual language with which to express events, they must be copied somewhere to be of use, and so there are several channels to which the shim can echo events, though one in particular is always primary, currently either a text file in the current directory, or else the system logger.

The format of requests and messages is described in § 6.1; that of commands and the output stream in § 4.3; and the names and other attributes of the output channels to which the output stream can be directed, in § 6.2.

6.1 The Binary Upstream Protocols

The IB tws socket api works over a TCP/IP socket, so that bytes are delivered reliably and in order as long as the socket is working. Since, however, the tws is threaded, and talks to multiple upstream servers, the messages delivered over TCP are themselves delivered non-deterministically, that is with uncertain order.

Note that here as with other aspects of the IB tws api, there is no formal specification, and the IB tws does not itself have freely available sources, so that all of this section must necessarily be tentative. Note also that the IB tws api is currently under active development, so that the material here is very much subject to change, whether by the replacement of existing events with new versions, or the addition of new request and message types entirely.

The rest of this section will focus on: the interpretation of bytes read from the TCP socket, § 6.1.1; common features of the IB tws binary protocols, § 6.1.2; the initial handshake, § 6.1.3; api requests to the tws, § **??**; and api messages from the tws, § **??**.

6.1.1 Portability and the Tranfer Encoding

The shim has been designed to work with UTF-8 or other byte-oriented tranfer encodings, and database setup uses UTF-8 as the default table encoding. Use of the shim with systems having larger transfer encoding unit sizes is not supported, and this includes in particular connection to an api server running in a UTF-16 environment.

The database table load files in the distribution use only the ASCII subset of UTF-8, and in our environment the IB tws api server likewise confines itself to ASCII. Given those limitations, platforms with other byte-oriented transfer encodings have been used to host the IB tws, see e.g., Table 6.1.

os	locale	language	file encoding	default	bits
Linux	en_US	en	UTF-8	UTF-8	8
Windows 2000 [a]	en_US	en	Cp1252	Cp1252	8
Mac OS X	en_US	en	MacRoman	MacRoman	8

Table 6.1: Examples of java locale and default encoding parameters

[a]We use Windows for one legacy application only, and so the locally available Windows box is quite antiquated.

The shim uses a table-driven scanner to tokenize input, and its character classification tables are designed to work with bytes, with bytes outside the 7-bit ASCII range treated as alphabetic. Beyond this, the shim merely passes bytes around, so that the interpretation given to those bytes by the downstream applications or the IB tws is outside its control.

Although the shim has been observed to successfully connect to, make requests of, and parse the messages of IB tws api servers on platforms with a byte-oriented Java transfer encoding other than UTF-8, e.g., Cp1252 or MacRoman, such operation is the sole responsibility of the user. Such encodings have ASCII as a common subset with UTF-8, and their apparent interoperability is an artifact of the restriction in practice of symbol and other character text data to the ASCII subset.

6.1.2 Common Features

There is no record separator or terminator for either requests or messages, so that recovering from parse errors is nontrivial. The IB tws depends on a correct parse of the previous request to determine the record boundary, and such requests must therefor be correct in type and number. The shim, in contrast, includes logic to resynchronize the parser by discarding fields one at a time until a successful record match occurs. There is good reason why the tws and shim handle the problem of parse errors so differently, as safety dictates that the tws reject uncertain requests, to avoid malformed or incomplete orders, and that the shim seek to extract every last ounce of information from the event stream, in order to keep the downstream informed.

As a rule, parse errors do not occur with the upstream protocol languages except temporarily, after version updates. Although table driven design has been used to limit the effort required, the shim must sometimes be modified when the IB tws api version changes. In addition, on one occasion a (now obsolete and unavailable) release of tws was observed to garble messages, although a replacement release was available the same day we observed the problem.

Once stable operation has been achieved, however, I never see parse errors for supported request and message types. It is nevertheless the responsibility of the downstream client to watch the message stream and note the various error messages that might indicate that a parse error has occurred. Note that § **??** and § **??** following, in addition to describing the api request and message types, also indicate **??** that subset currently supported by the shim.

6.1.3 The Client – IB tws Handshake

The IB tws supports multiple versions of the api protocol, and it decides which one to speak based on the initial handshake from the client.

The handshake has two steps, one message each from the downstream client and the IB tws server. The client sends the api version it intends to speak, and the server sends back the current and maximum protocol version it supports, followed by, after version 20 and more recent, the connection time string.

6.2 The Shim Output Channels

6.2.1 File Writes

6.2.2 Database Posts

Chapter 7

The Database Architecture

Chapter 8

Patterns, Tables, and Classes

8.1 Significant Design Patterns Used in the Shim

[32]

8.1.1 Binding Patterns

8.1.2 The Singletree

8.1.3 Type Symbols

8.1.4 Factories, Accumulators, and Wrappers

See Figure 8.1

These objects may be allocated on the stack at will, and are meant for parameter passing only. They should never occur as a data member except as part of another temporary. The suffix `_a` to the reference short name, here meaning auto, is meant to remind the programmer of this.

In most cases, to provide transparency, the ctor arguments are one-to-one with the data members, and to allow convenient allocation in expressions, there are no non-const member functions beyond the constructor. In short, as the file and class name indicates, they're just wrappers.

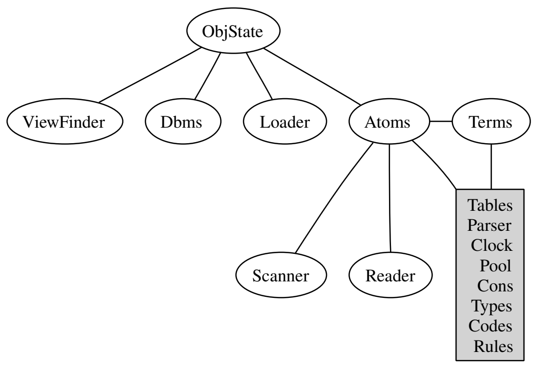


Figure 8.1: Factory wrappers

8.2 The Shim Database

See Figure 8.2 for the foreign key dependency graph of the tables in the shim database.

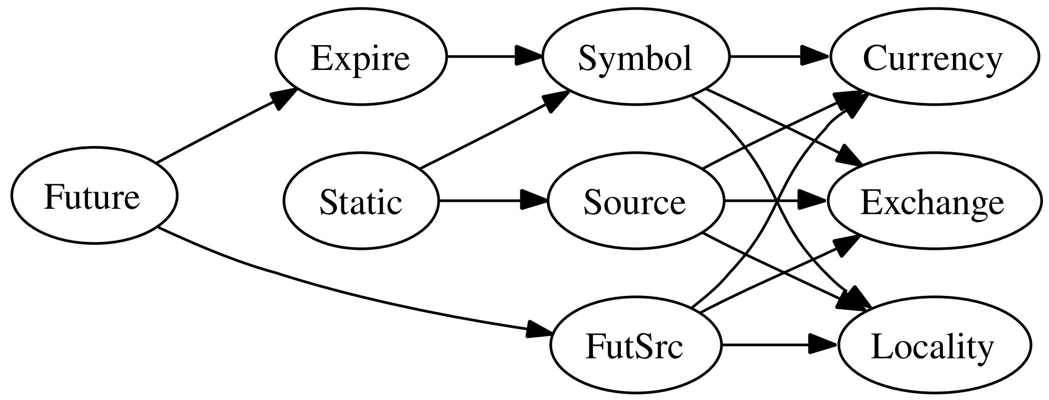
8.2.1 Database and Other Scripts

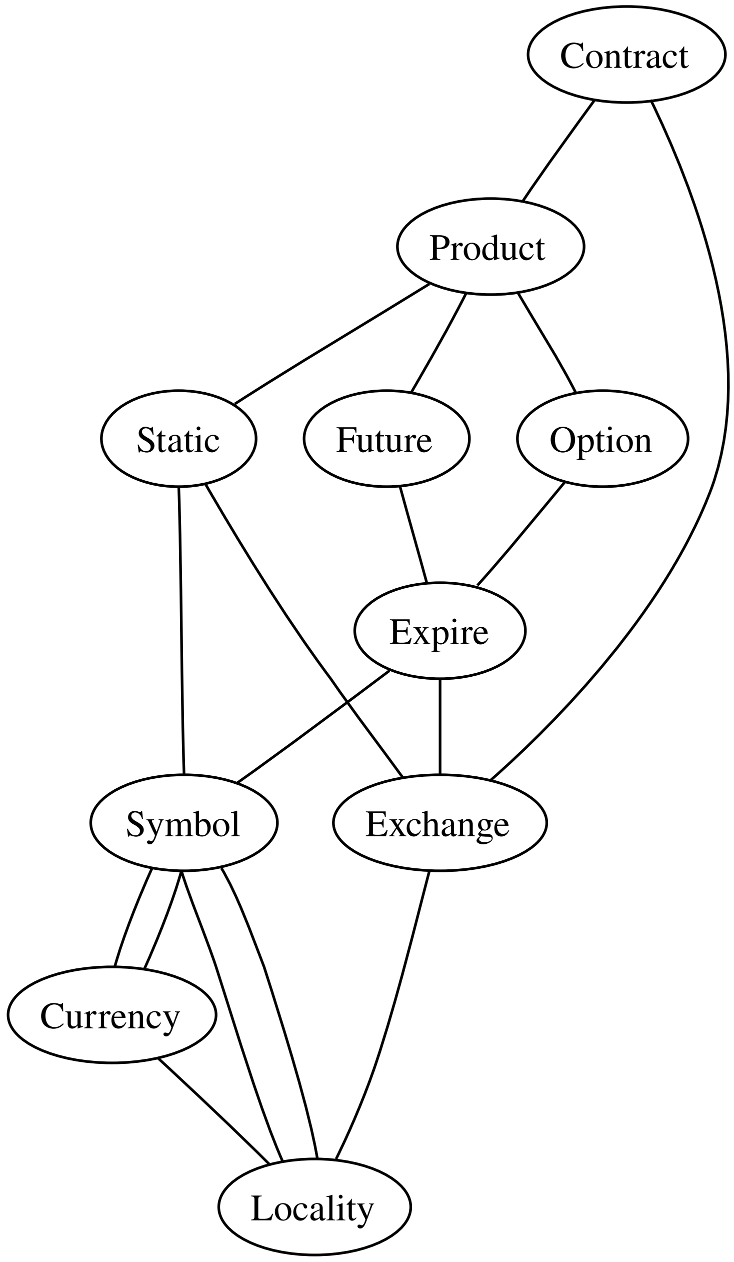
Once given a database (**??**), the shim database tables are created and loaded with initial data values by running the `create.sql` script, described in § 8.2.1.

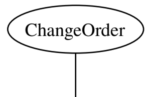
Database and Table Creation

Tables naturally group into the three categories of security, subscription, and order transactions, with tables from the latter two categories having foreign key constraints depending on security-related tables.

In order to create or recreate the dbms, tables are first dropped in top-down order by foreign key dependency, the reverse order of creation. Then create table statements for securities, subscriptions, and orders are source'd in creation order, that is bottom-up by key dependency. (The GraphViz dot program and the dbms-related dot files in ../dot may be used to generate foreign key dependency graphs). Table loads are finally performed via a mixture of literal and source'd insert statements, the load files for the latter case named according to the table name, e.g., for a given table TabName, the load file if it exists is named TabName.sql, and found in one of the directories req or mod (table load files in opt are not used here).


Expire
Symbol
Currency
Future
Static
Source
Exchange
FutSrc
Locality


Contract
Product
Static
Future
Option
Expire
Symbol
Exchange
Currency
Locality


ChangeOrder

To populate an empty dbms and load initial data, run this script once from the mysql interpreter to create and load tables, and then rerun a second time to ensure that a fix-point has been reached, that is to check that all tables are conditionally dropped before creation.

Since recreating the dbms tables also loads tables with symbols, contracts, and orders, this script is one means by which to add new tuples for dbms relations that provide input to the shim; edit the ¡table-name¿.sql insert statement in the appropriate load file, halt all instances of the shim that might read the dbms, and rerun this script. Realize that all history and state information stored in the dbms will be lost, unless it is first extracted via mysqldump, added to load files, and source'd as necessary.

All persistent tables have the engine type InnoDb, to provide foreign key dependency checking. This engine type *must not* be changed to some other type, e.g., MyISAM, else the shim die due to subscript-out-of-range errors.

For the Contract and SubRequest tables, the SecType and SubType attributes, respectively, tag an otherwise untyped numerical index, the actual type of which is a union of foreign key types, even though not explicitly declareable as such. Since the derived type data members for some objects in the shim are defined via these tagged variants, the underlying tables must be loaded with the related values, else there be an index-out-of-range error, so that for the Contract and SubRequest relations, in addition to the explicitly declared foreign key dependencies, there are also these implicit dependency constraints on the underlying Detail or Filter tables.

```
source drop.sql;
source enum.sql;
source base.sql;
source secs.sql;
source subs.sql;
source xact.sql;
source load.sql;
source risk.sql;
```

Figure 8.3: Commands from the `create.sql` script

Table Updates

8.3 The Type System

The type symbols are used to label application domain objects: all events, database table records, and database table attributes have related predicate, function, and constant type symbols, respectively, with which the term instances are labelled. The type symbols provide operations to parse input text, construct events and tuples, lookup constants, and specialize — downcast — terms to the constituent atom or finite domain symbol when such exists. The type hierarchy is enormous, and so I'll introduce it in stages, starting from the root, and considering alternatives.

8.3.1 The Fundamental Three-Way Partition for Types

There are a number of ways to classify the type symbols. Most basically, it should be possible to partition the Type class according to the predicate, function, and constant object categories mentioned above, as shown by the top portion of a possible type class derivation hierarchy in Figure 8.4.

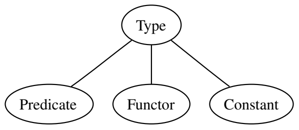


Figure 8.4: A hypothetical type derivation hierarchy

8.3.2 Multiple Inheritance in the Type System

In addition, the three basic kinds of symbol may be usefully grouped in two different ways, as we focus on where the related terms occur, and how they are constructed; the first categorization asks whether term trees are roots or not, and the second, whether they are leaves or not. Of these partitions, the first distinguishes top level terms from data, and the second, compound terms from constants.

The use of multiple inheritance, see Figure 8.5, allows the inheritance hierarchy to express both cases: a type is a label for either a predicate, top-level term, or else for an argument of another term; and in addition a type denotes either a compound term, or a constant.

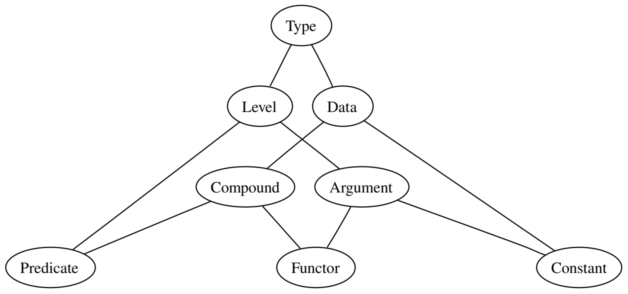


Figure 8.5: The top of the type symbol hierarchy

The derived classes of Level, Predicate and Argument, together partition the types in accord with the fact that a top level term is labelled with a Predicate symbol, while Function and Constant symbols may be arguments to a term; and similarly, the children of Data, Compound and Constant, again partition the types as a term has internal structure or not.

8.3.3 Input Matching

Type symbols are used to implement the parse algorithm. The Type base class provides a virtual function `match()`, and rule definitions are found for the start symbols, currently the only disjunctive stage in parsing; compound terms, where the rule text loops over the argument types of the schema, matching recursively for each; and for the base case, the terminal or `Constant` types.

8.3.4 The Full Hierarchy

In Figure 8.6, the full derivation hierarchy of the type classes, the shaded rectangular nodes represent class templates, and the numbers in parentheses following the template name, the number of instances defined via typedefs.

8.3.5 Application Finite Domain Dual Types

Given a type with dual index domains, in particular sequential numbers and character strings, and where we wish to lookup using either domain, so that there are two useful values for id(), the sequence and hash numbers, the Dual class template provides a cyclical wrapping that:

- conserves type information, so the original type can be recovered; and
- redefines the operator¡() relation to the identifier hash code, so that the wrapper sorts in hash order, and supports indexed lookup via the label text.

Note that the template type T must have a data member named ’name’ itself having member function id(). The type T is typically an identifier or string.

Although the finite domains are the common case of dual types, the Dual template is also used for the Relation class.

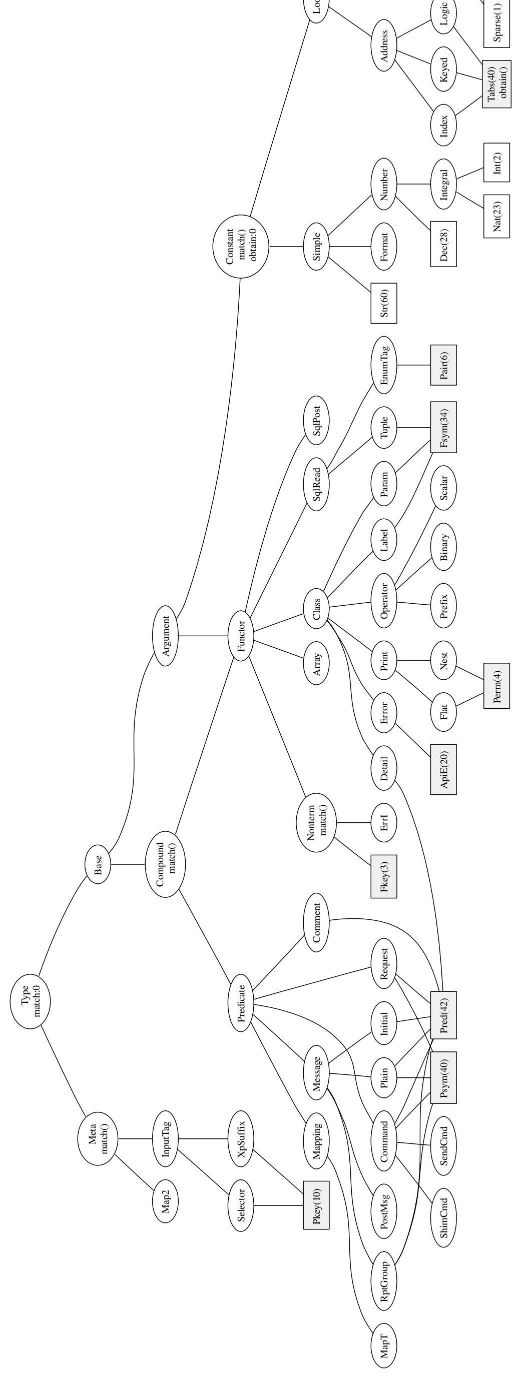


Figure 8.6: The type hierarchy

8.4 Atoms: the Routeable Objects

See Figures 8.7 and 8.8 for the derivation hierarchies for the object and term hierarchies.

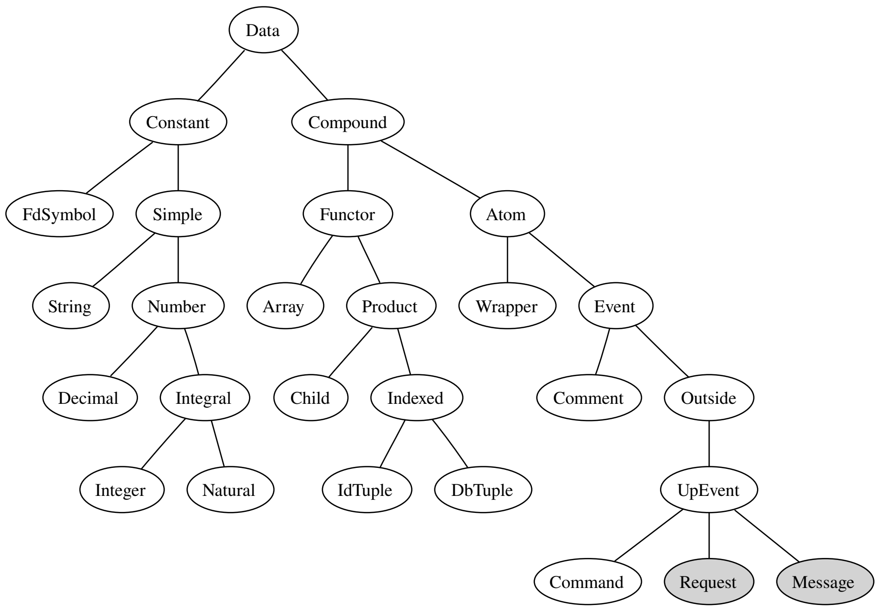


Figure 8.7: Objects

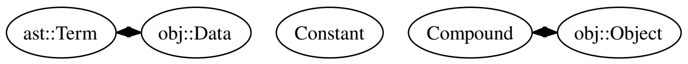


Figure 8.8: Terms

8.4.1 Database Tuples

See Figure 8.9

See Figure 8.10

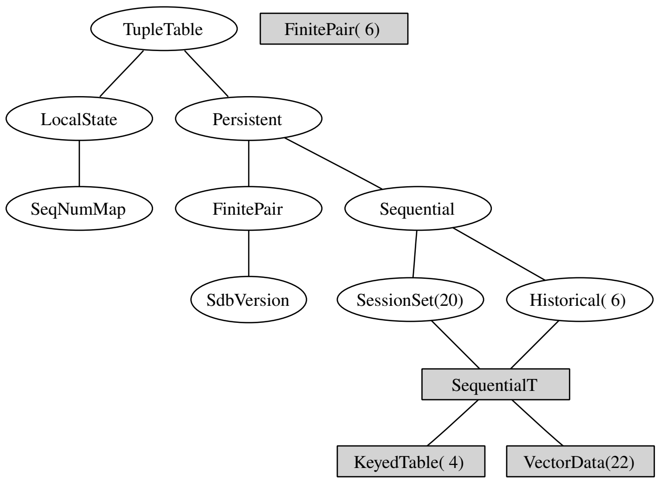


Figure 8.9: Relational tables

8.4.2 The Event Hierarchy

Commands

See Figure 8.11 for the derivation hierarchy of the Command events.

Requests

See Figure 8.12 for the derivation hierarchy of the Request events.

Messages

See Figure 8.13 for the derivation hierarchy of the Message events.

See Figure 8.14 for the derivation hierarchy for error message type codes.

Comments

Other Atoms

See Figure 8.15

See Figure 8.16

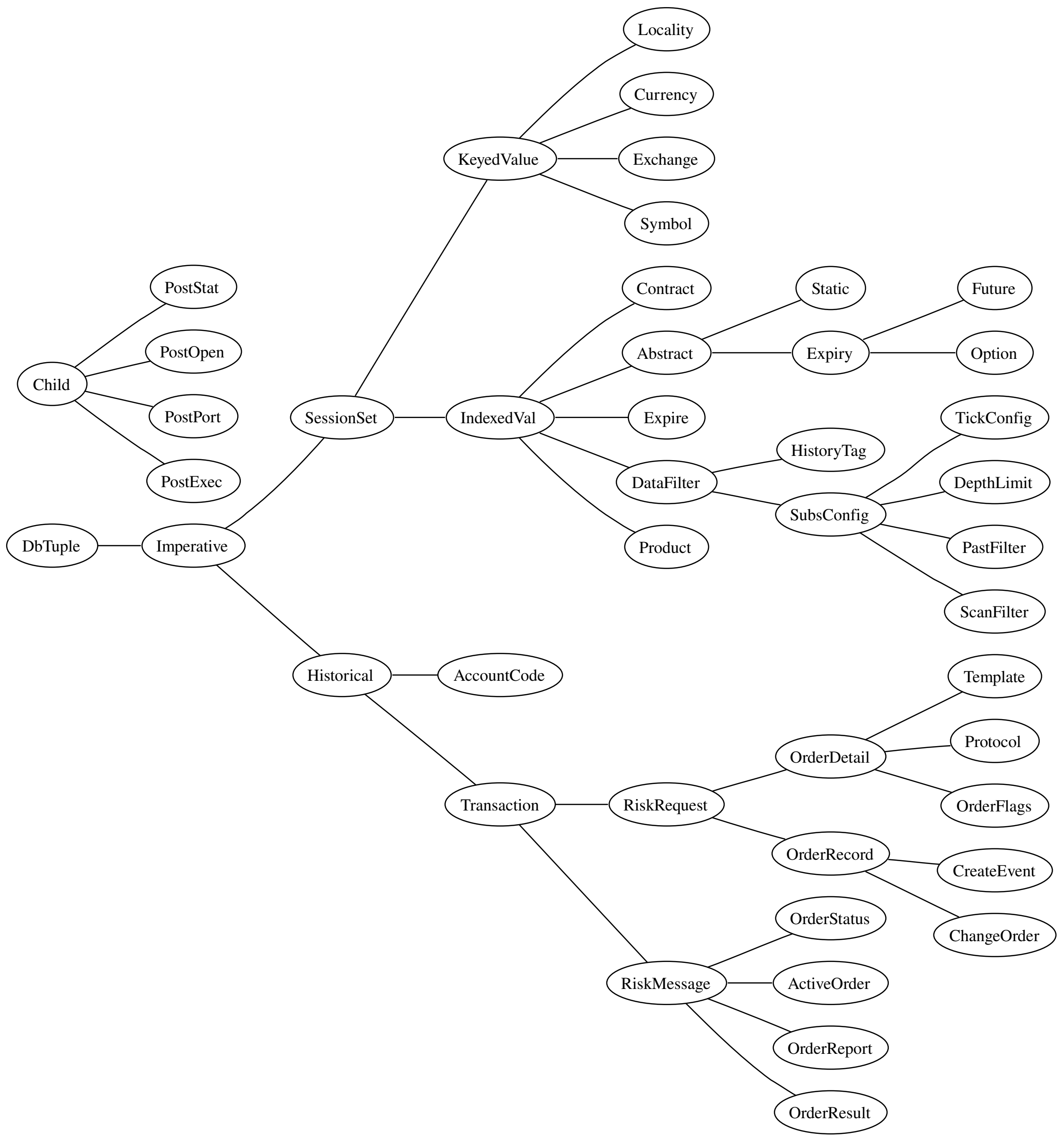


Figure 8.10: Records

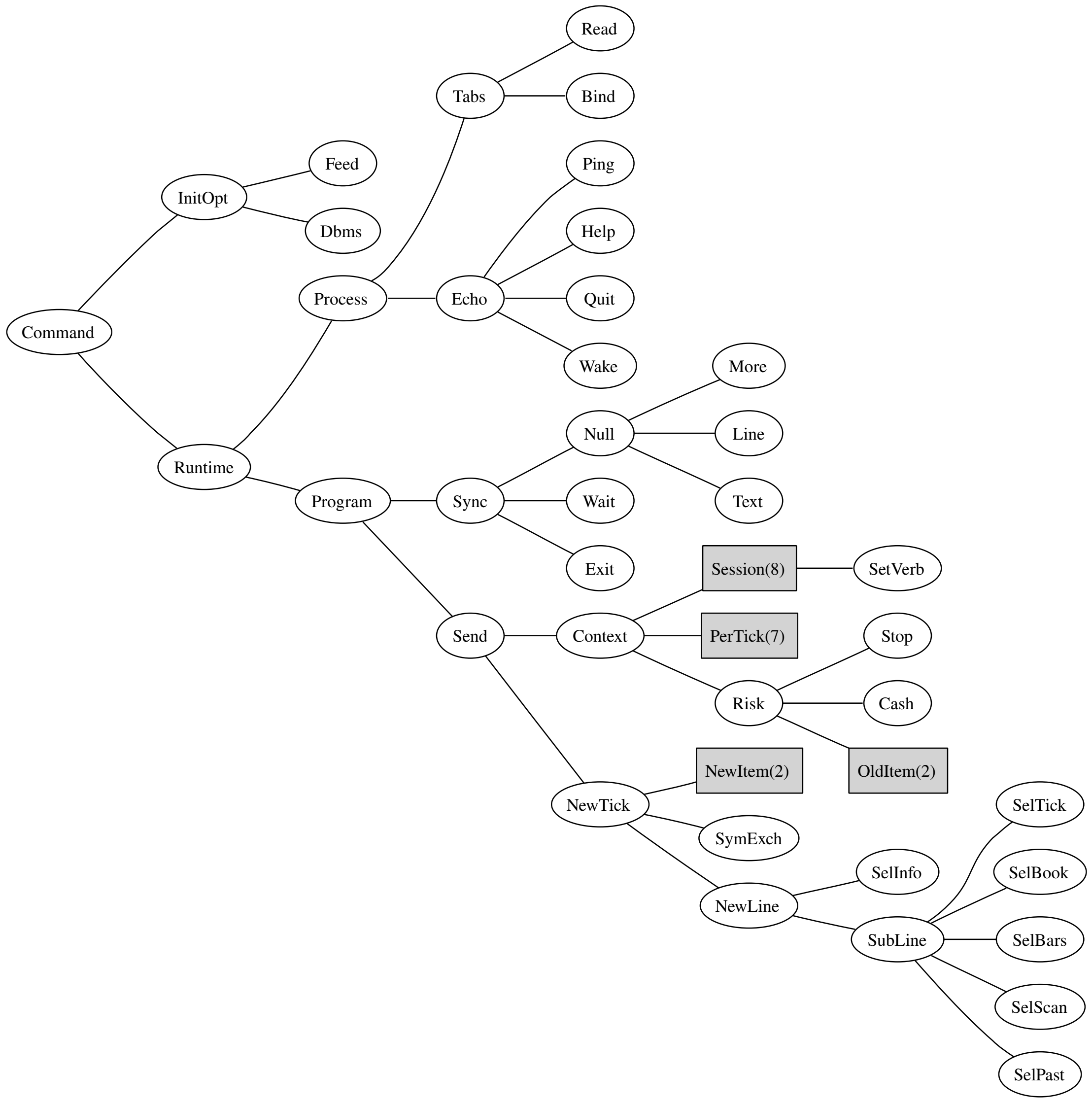


Figure 8.11: Commands

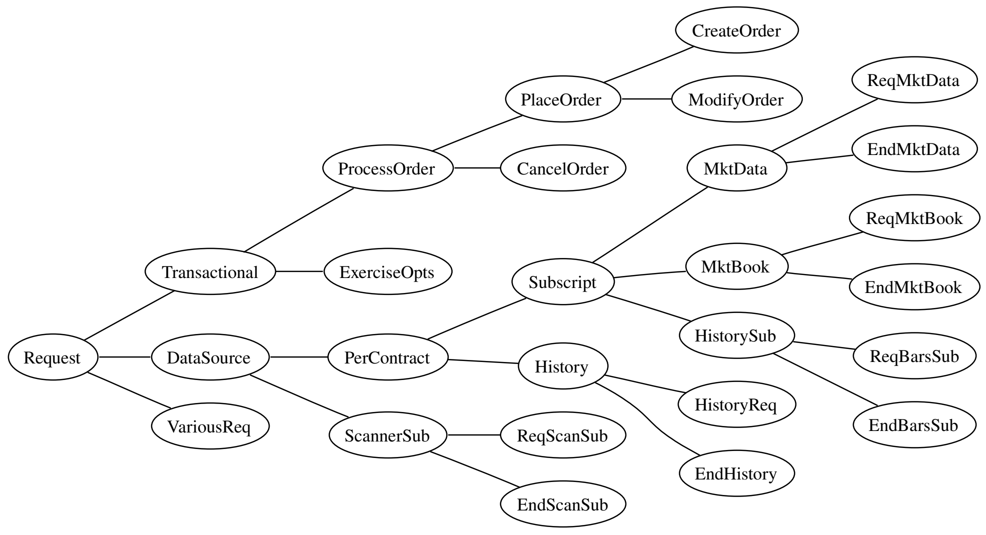


Figure 8.12: Requests

See Figure 8.17

8.5 IO Stream Objects

See Figure 8.18 for the derivation hierarchy of the Stream classes.

See Figure 8.19 for the derivation hierarchy for IO channel abstraction.

See Figure 8.20

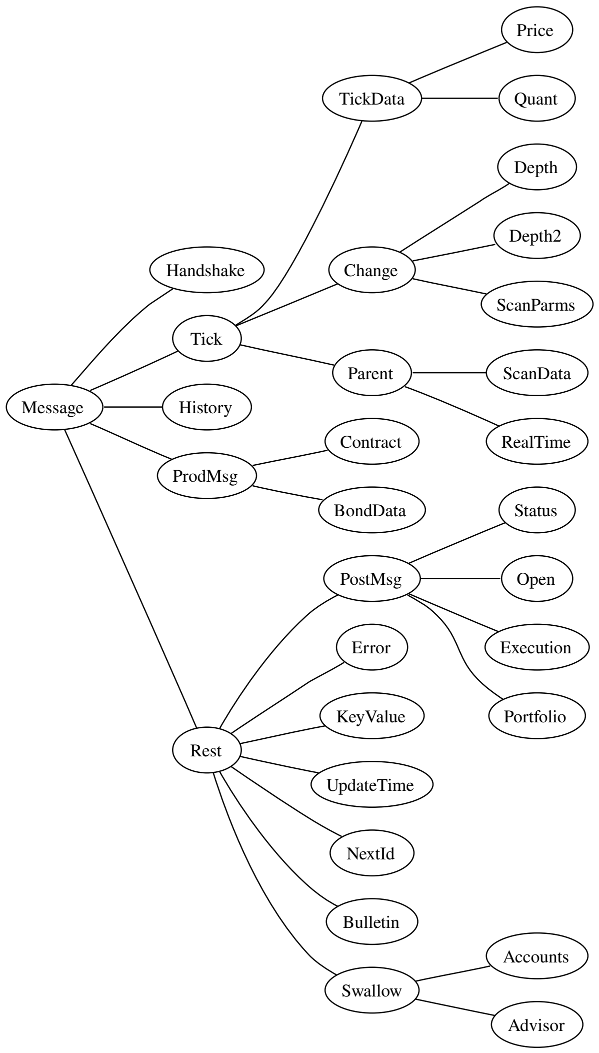


Figure 8.13: Messages

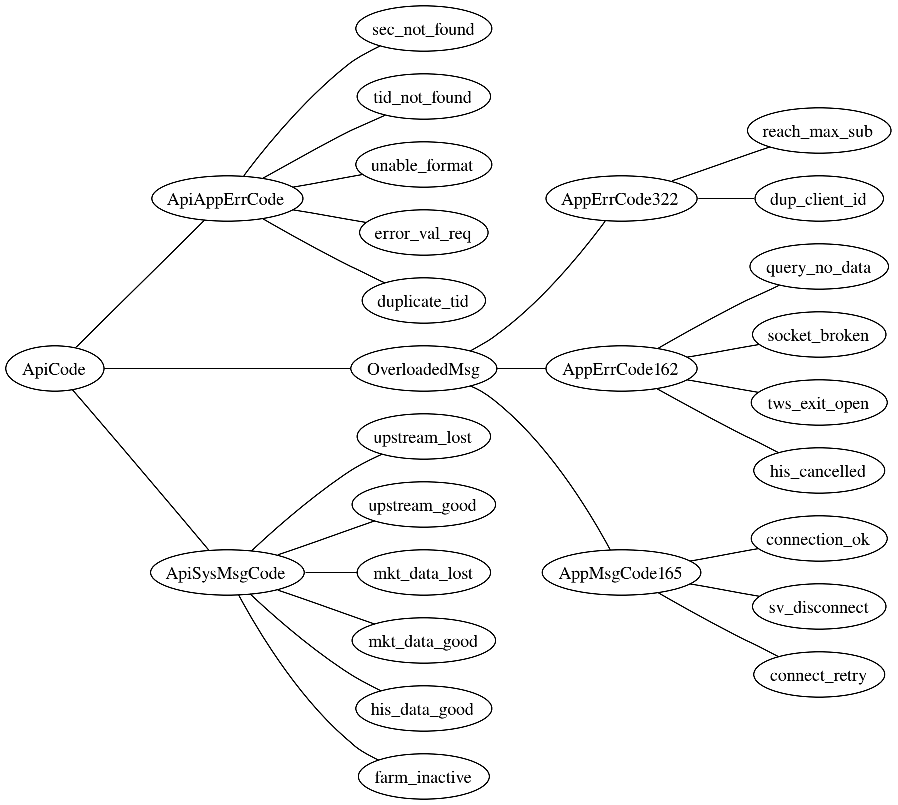


Figure 8.14: Error messages

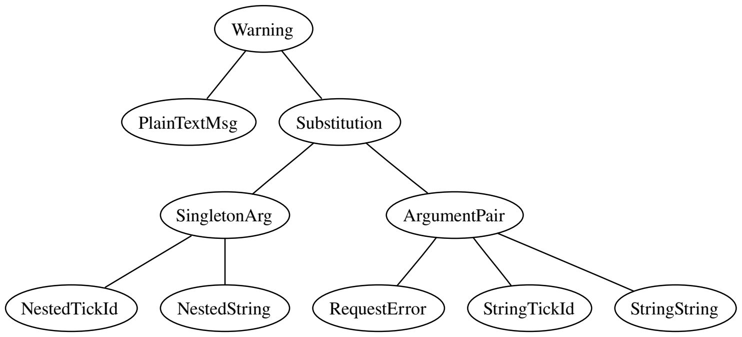


Figure 8.15: Warnings

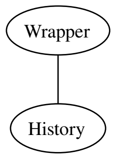


Figure 8.16: Augmented atoms

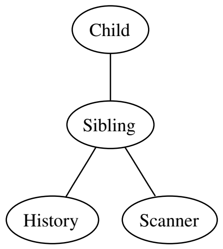


Figure 8.17: Sibling atoms

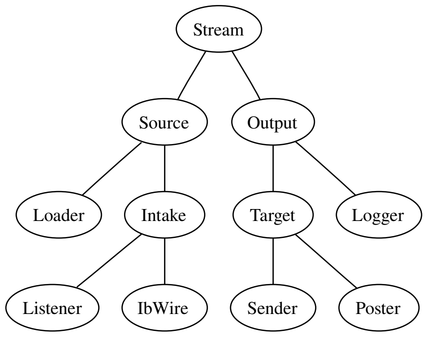


Figure 8.18: Stream variants

Figure 8.19: IO channel abstraction

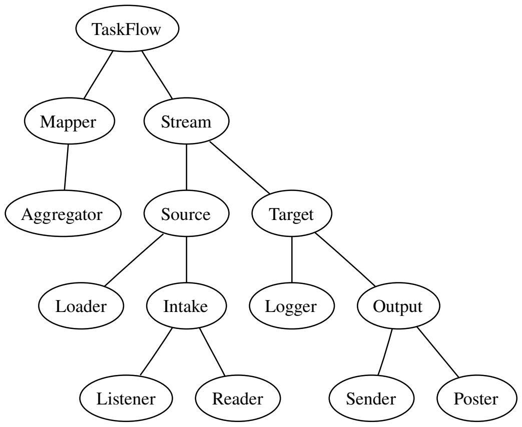


Figure 8.20: Mapper hierarchy

8.6 Time-related State

- the multiple time scales
  - clock ticks – seconds
  - request sends, and IO timeouts – 20 milleseconds
  - log data time stamps – microseconds
  - the processor clock, presumably nanoseconds or better
- the processor time stamp counter

8.6.1 The TimeStamp

8.6.2 The Clock

The Clock interface provides access by its parent and friend, the Timer, and via two public methods, `clock_time()`, and `clock_usec`, see Figure 8.21.

The private `check_time()` method uses the `localtime()` system call to determine the externally maintained time, in order to see if another second has elapsed since the last call, and, if so, updates the `time` data member, returning true if an update occurred, and false otherwise.

friend/method	explanation
Timer	singleton parent checks for clock ticks due to periodic IO timeouts
clock_time()	return saved seconds since midnight, update is triggered via Timer
clock_usec()	microseconds since the last clock tick, from time stamp counter
check_time()	synchronize saved time with local time, and return true if lagging
now()	determine the local time via an operating system call

Figure 8.21: Interface and private methods of the Clock

The Timer must check the Clock sufficiently often to obtain each clock tick, since the `check_time()` method indicates only the presence or absence of a clock tick, not the number that have occurred.

The TimeStamp constructor is the only client to use the `clock_usec` method. Clients using the `clock_time` method include TimeStamp, Timer, Tasks, and various agents such as the Historian.

8.6.3 The Timer

8.6.4 The Scheduler

8.6.5 The Task Set

Chapter 9

Stages of Computation

Program process stages are suggested by the root of the program call graph, Figure 9.1, with computation beginning with the construction of singletons in `main()` via static member functions and followed by a member function call of the `Shim` singletree; for the common case, a run loop that waits for input and processes events; and either, given normal termination, close of the upstream connections prior to successful program exit, or otherwise, exit with abnormal return code.

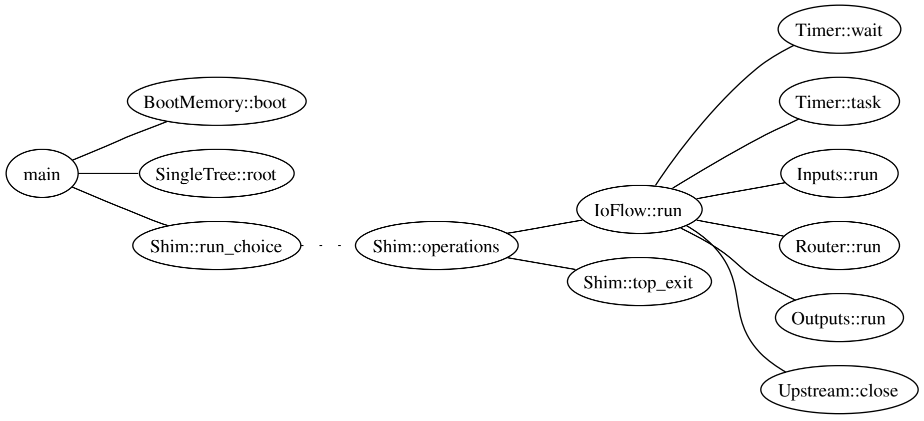


Figure 9.1: Tip of shim call graph

9.1 Initialization Via Construction of the Singletree

There are many advantages to including all singletons within a single root object, the *singletree*, not least of which is that the constructor call graph controls the order of initialization. Under the more widely used approach of static variables, singletons are bound via multiple definitions scattered over multiple files, and there is no straight forward approach to controlling, or even knowing, the sequence of singleton construction. Applying the singletree pattern eliminates the global, class, and local static variables that infest most programs, and allows the designer to restrict access to global state.

For a system such as the shim where the sources follow this singletree pattern, there is a singleton inclusion hierarchy, Figure 9.2, and related constructor call graph that correspond closely to the stages of program initialization. For the shim, the overall construction of the `Shim` object breaks down into five steps, first the four constructor calls of: the components singleton, § C.1.1; program constants, § 9.1.1; `IoFlow` object, § 9.1.2; and `Router` object, § 9.1.3; followed by initialization procedures, `IoFlow::init()` and `Router::init()`, § 9.1.4, called from the `Shim` constructor itself.

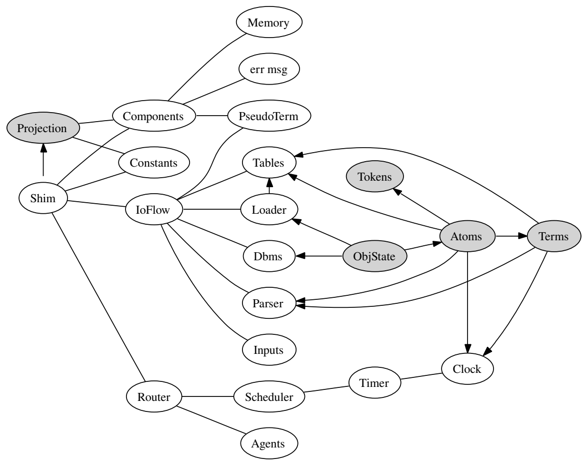


Figure 9.2: The upper nodes of the graph of singletons (unshaded), and related wrapper and factory classes (shaded). Edges flow generally from left to right. The `IoFlow` and `Router` objects are the primary application domain data members of the `Shim` object.

9.1.1 `one::Constants`

9.1.2 `iof::IoFlow`

9.1.3 `one::Router`

Readers send input events to the router, which distributes them, as needed, to its agents before sending them to the various output channels. Table 9.1 gives the mapping from each agent to the request and message class types for which it bears responsbility, though in many cases no analysis is currently performed, with the router acting only as a multiplexer to relay events to the downstream receivers. The two agents `Runner` and `Bookkeeper` are closely coupled in function, as suggested by the lack and multiplicity of message types for the first and second, respectively, and the implementation of these two agents is in particular likely to change significantly in the future. The agent classes are arranged according to the derivation hierarchy of Figure **??**.

9.1.4 Delayed initialization

9.1.5 Modes, Options, and Commands

Modes

See Figure 9.3 for the derivation hierarchy of the Mode classes.

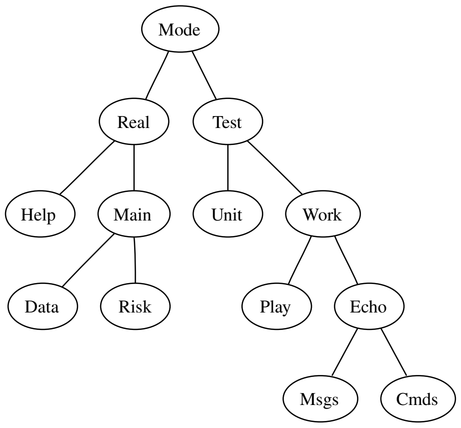


Figure 9.3: Modes

Agent	Requests	Messages
Contracts	ReqConData	Contract
		BondData
TickerTape	ReqMktData	Price
	EndMktData	Quant
Specialist	ReqMktBook	Depth
	EndMktBook	Depth2
Historian	ReqHistory	History
	EndHistory	
Reporter	ReqBulletin	Bulletin
	EndBulletin	
	ReqScanParms	ScanParms
	ReqScanSub	ScanData
	EndScanSub	
Runner	PlaceOrder	
	CancelOrder	
	ExerciseOpts	
Bookkeeper	OpenOrders	Status
	AccountData	KeyValue
	Executions	Execution
	AllOpens	Open
		Portfolio
		UpdateTime
FinAdvisor	AutoOpens	
	ManagedAccts	Accounts
	FinAdvisor	Advisor
	ReplaceFa	

Table 9.1: Agents, requests, and messages

Options

Commands

```
Command function -- a brief guide:

    help     Display the text you are reading now

    ping     Check connectivity between downstream and the shim
    next       "     "            "     the shim and the tws
    look     Add newly inserted entries of the database to the shim
    load     Load the SubRequest table, adding and deleting subscriptions
    list     List current subscriptions

    wait     Add an n-second bubble to the shim's request queue
    wake     Clear the bubble counter, restarting   "       "
    quit     Terminate the shim (after waiting on bubble, if any)

    eval     Run a child program, reading commands and returning messages
    sign     Broadcast a signal and its args within a stream group
    text       "       free-form message text  "    "   "      "
    done     Close the IO channels between the shim and the sending child

    verb     Set the tws log level
    news     Subscribe and unsubscribe to news bulletins
    open     Query for open orders
    acct       "    "  the (label, value, currency, account) tuple list
    data       "    "  contract data

    tick     Subscribe and unsubscribe to market data
    book       "       to market depth (cancel currently broken)
    past     Query for historical data (cancel not yet implemented)

    wire     Create, modify, submit or cancel an order
    cash     Exercise an option

The following command verbs are synonyms:

    load and bulk
    acct  "  account
    past  "  history
    wire  "  order
    cash  "  exercise
```

```
Note that the bulk and individual subscription commands overlap in
functionality, and that it is safest for now to use one or the other type,
but not both, within a session.  That is, if you use load, consider avoidin
tick, book, and past, and vice versa.
```

```
Command notation -- a brief guide:

Commands begin with the command verb, followed by the parameters, if any,
and terminated -- always -- by a semicolon.

The simplest command verbs have no parameters:

    help    next    list    wake
    open    look    load    bulk
    done

The following command verbs allow whitespace and arbitrary comment text
to follow up to the terminating semicolon -- don't forget it!

    ping    quit

The parameter syntax for most of the remaining commands is minimal:

    wait N;
    verb Level;
    data Cid;
    tick Req Cid I;
    book Req Cid I;
    past Req Cid I;
    cash Act Cid Q Force;
    eval Mode [Group] Path;

    N    : the number of seconds
    Q    :  "  quantity
    Cid  :  "  contract id, a database uid attribute value of Contract
    I    :  "  configuration id, a database table uid, one of, by verb:
           tick: TickConfig
           book: DepthLimit
           past: PastFilter
    Level: one of (System, Error, Warning, Info, Detail)
    Mode :  "  "  (data, risk)
    Req  :  "  "  (add, del)
    Act  :  "  "  (exercise, lapse)
    Force:  "  "  (yes, no)
    Group: a stream group id, or zero for the group parent
    Path : " pathname, the program of which should write commands to stdout
           and accept log-style text on the stdin

The order command, due to the number of api parameters, is more complicated.
Much of the complexity is hidden in the LineItem record, but modifiable
parameters must be provided on the command line.

    wire(Oid,Type,Op,Q,P,Aux,T);

    Oid : the line item id, a database uid attribute value of LineItem
    Type: an order type, e.g., MKT, LMT, STP, or TRAIL
    Op  : one of (Create, Submit, Modify, Cancel)
    Q   : the quantity
    P   :  \"  limit price
    Aux :  \"  auxiliary price
    T   :  \"  timeout (just a dummy for now, not yet used)

For examples, and in any case if this crib sheet is not sufficient,
see the regression tests, in particular the program text of bin/includes.

Note that the execution report, market scanner, option modelling, and all
financial advisor related requests and messages are not yet supported.
Please subscribe to the mailing list and let us know if you need these
features, see:

    http://www.trading-shim.org/mailman/listinfo
```

9.1.6 Building the Internal Database Dependency Graph

The vector dependency tree representation below is induced by (manually) eliding edges from the foreign key dependency graph for the dbms relations, according to four cases:

- Edges that lead to multi-node cycles are removed by eliminating the offending foreign key type specification in the schema table returned by TupK:schema(). For now, this is a problem only with ComboSet and ComboLeg. It follows that these tables are not yet effectively supported, and that they will need to be redesigned before becoming useful.

- Self edges are removed; it is up to the downstream user to ensure that local keys in tuples are strictly less than the tuple id.

- Edges to functional tables are removed; these tables are checked exactly once, prior to any vector input, and so are well defined as long as downstream users do not (incorrectly) alter such tables while the shim is running.
- Redundant edges are non-deterministically removed from the resulting acyclic graph until the result is a free tree.

The third step is not only a significant optimization, as it reduces dbms load, but also necessary for correctness, since the table provided for vector lookup during the post-order traversal does not include the functional tables, so that there would be an index-out-of-range error if edges for those tables were included.

The last step, though at most a minor optimization, is actually more a matter of convenience, as fewer edges need to be added to the adjacency lists below.

Note that the initial – and resulting – dependency graph *includes* the union-join edges for the variant parts of Contract (e.g., FutDetail) and SubRequest (e.g., PastFilter), even though these foreign key dependencies are not explicitly declared via the sql create table statements.

See the foreign.* files in ../dot for a graphical representation of the dbms foreign key dependency graph. Union-join (tagged variant key) edges are indicated there by dashed-line edges.

9.2 IO Selection and Event Scheduling

9.2.1 Calculating the Processor Clock Frequency

`Clock::cpu_mips()` calculates the processor clock frequency in Mhz for use in calibrating the processor time stamp counter, which gives much better resolution than API access to the operating system clock. Such counters are standard on modern processors, access to them may be obtained via gcc inline assembler, and such code is wrapped here in the `set_time_stamp()` call provided in the library, which leaves the problem of calibrating the counter with respect to wall clock time.

The best approach is to compare two processor time stamp counter values after a reasonably predictable delay, here a call to sleep for 1 second, which has the virtues of simplicity, accuracy, precision, and portability, though with an admittedly large time cost.

That cost is acceptable given that it is paid only once, during Clock construction, and that worse problems arise with the alternatives. Shorter time delays, e.g. via select(), have proven much less accurate, and though reading /proc/info is fast and simple, it is not portable.

Since this frequency calibration value affects Scheduler operation, and is used as well for every event time stamp, the correct choice here is to get it right to start with. Although averaging might be used, the likelihood of interrupts increases, and computation must still deal with counter wrap around.

In the procedure `Clock::cpu_mips()`, a loop is used to eliminate such rollover events, with termination requiring that the processor clock frequency be lower than $2^6 4$; the risk of the alternative is considered acceptable for the foreseeable future.

In order to avoid the one second startup cost, and reduce the running time of the regression test suite, access to the `Clock::cpu_info()` function is provided via the ”fast” command line option, for test modes only.

9.2.2 Finite State Automata Definitions

See Figure 9.4

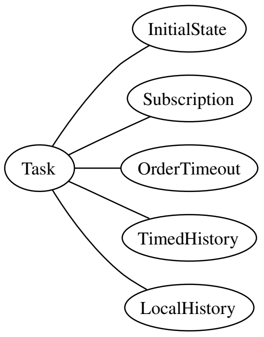


Figure 9.4: Tasks

See Figure 9.5

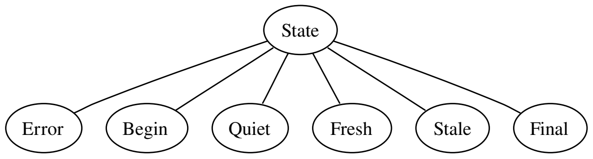


Figure 9.5: States

See Figure 9.6

Figure 9.6: States

The Parse Cursor State Transition Function

See Figure 9.7

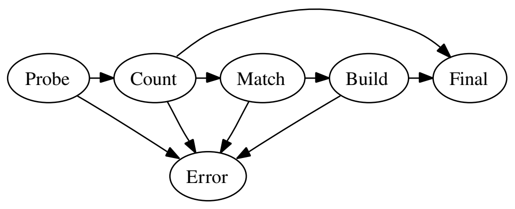


Figure 9.7: Cursor states during event parsing

The Timer Class State Transition Function

See Figure 9.8

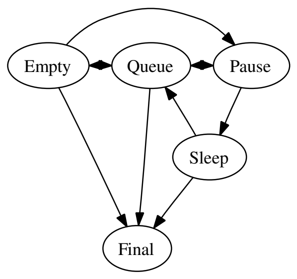


Figure 9.8: Shim process timer states

Empty is the start state, and the error state is never reached, with empty cells corresponding to nops. Feed events occur due to request enqueues, and wait, wake and quit events, respectively, due to their eponymous commands.

In queue state, 20 msec intervals generate slot, last, or null events as the queue has more than one, one, or no requests, leading to a request send in the case of a non-empty queue, and changing to the empty state if there was at most one request to begin with.

In pause and sleep states, 1 second intervals generate tick or time events, as the pause timer is greater than zero or not, and in the case of tick events, also decrement the timer by one. The wait command increments the pause timer by its parameter, and is cummulative.

Sleep mode is reached when a quit command is made in pause mode, and wake clears the pause timer at any time that it occurs, so that (a), quit commands are delayed by the pause timer; (b), that delay may be increased via additional wait commands; and (c), the pending quit command is retractable while the pause timer is counting down.

The Subscription Tracking Transition Function

See Figure 9.9

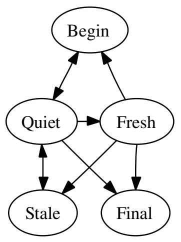


Figure 9.9: Subscription retry states

Market data subscription tracking starts in the Begin state, with a request enqueued, whether due to a downstream command, or a timeout-induced retry.

The request send causes a change in state to Quiet, reflecting the lack of data returned for the request up to this point. Tick data price and size events shift state to Fresh, and quiescent intervals without additional data, whether from Quiet or Fresh, cause a retry action and change state back to Begin.

Market data farm connection loss, or down events, shift state to Stale, where, unlike Quiet and Fresh, there is no timer running. When the link comes back up, the state moves to Quiet.

Finally, if the retry limit, if any, is ever reached, the subscription enters the Final state, and is cancelled without an attempted renewal.

There are log messages generated to reflect changes in subscription state when the first data for a given subscription request is received; a retry is performed; and in the case that the subscription is cancelled. Market data farm state is also tracked outside the fsm for this automata, and the related messages echoed as internal log messages, so that downstream processes can track subscription state via log comments, without having to follow tws error messages.

9.2.3 Query Data Bar Intervals

The edge list of an interval T_0 is the set of larger bar intervals for which T_0 is the largest even divisor.

Or, equivalently, given the graph with bar intervals as nodes and the evenly divisible relation as edges, prune away all redundant edges that leave the graph fully connected.

Let a roll event for a bar interval be a time, measured in seconds since the epoch, that is evenly divisible by the bar interval width.

Given a 1 second tick event, the associated number of seconds since the epoch, and a vector of last roll event times for each interval, then this graph allows us to check a minimum number of bar intervals for roll events, for the common case just two.

Note that at each second the 1 second bar necessarily rolls, and that it is trivially a common divisor of any larger interval measured in seconds. Using depth-first traversal starting at that smallest bar, and following each path until a node fails to roll or a leaf node is reached, correctness follows from transitivity for divisibility, and minimum work from the definition of the edge relation and structural induction on the graph, noting that branches are mutually disjunctive.

In practice, for the bar intervals as currently defined, the only bar of interest is m01, with the two successor nodes of m02 and m05. If the m02 interval is eliminated, the resulting graph is a list, and array traversal can be used in place of depth-first search.

9.3 Input Analysis

9.3.1 The Internal Database Update Algorithm

Given an explicit representation of the foreign key dependency graph, check the sizes of all monadic tables prior to non-trivial queries, mark the dependency nodes of enlarged tables as changed, and, for the dependency graph branch rooted at that table indicated via the read request, use post-order traversal of that graph to order the queries of marked nodes to ensure that foreign key dependencies are satisfied.

9.3.2 The Event Input Algorithm

for each reader R of interest, and given its start symbol S for each compound term T matched starting with S against some prefix of R remove the prefix text from R, and route T

9.3.3 Tokenization and Type Checking

Note: According to the logic in the sample client, the tws is free to use the null string to represent zero. It follows then that the scanner must also read the null string as 0.

9.4 Object Routing and Processing

9.4.1 The Subscription Watchlist Update Algorithm

The results of dbms watchlist queries are sorted in contract id order, and merge traversal is used to match for adds and deletes:

Once the current watchlist is read from the dbms, copy subscriptions from the watchlist vector to a temporary, clear the watchlist, and then traverse the old and new watch lists, queueing subscription cancellations and requests to the router.

9.5 Output Processing

9.5.1 Request Sending

See Figure 9.10

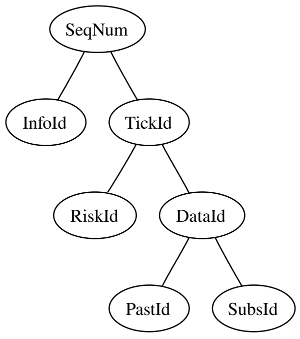


Figure 9.10: Tick Ids

9.5.2 Dbms Post

9.5.3 Event Logging

9.6 The `Position`, `OrderContext` Command

9.7 The `Wire` Command

9.7.1 Order Wire Format Specification

For more information about order semantics, see the ib docs, that is, at http://www.interactivebrokers.com/en/main.php, pull down SOFTWARE, and drill down to the C++ or Java socket client properties frame via:

FIX/API → API User's Guide → Use the APIs → Java → Java Socket Properties

Note that the PlaceOrder wire format is the high turnover datatype, with a version id of 15 for the current protocol record format, and that the documentation above describes approximately 20 additional fields not found in the version 15 format. The newest version differentiates ignoreRth and rthOnly, and in addition adds auction strategy, ocaType, rule80A, settlingFirm, allOrNone, minQty, percentOffset, eTradeOnly, firmQuoteOnly, nbboPriceCap, and six BOX parameters.

To verify that the data element sequence ordering corresponds to that of the order portion of the wire format, see the Java client sample code. For an overall view of the wire format data element ordering, see the comments in the PlaceOrder class.

9.8 Dealing with Broken Compiles Caused by Preprocessor Macros

Chapter 10

Exploring the Sources Directly

Chapter 11

Roadmap

Part III

Appendices

```
shim-070803$ time make
make -j4  shim
make[1]: Entering directory '/home/pippin/src/tws/src/shim-070803'
g++ -Wall -g -I/usr/include/mysql  -c -o mode.o src/mode.c
g++ -Wall -g -I/usr/include/mysql  -c -o once.o src/once.c
g++ -Wall -g -I/usr/include/mysql  -c -o bind.o src/bind.c
g++ -Wall -g -I/usr/include/mysql  -c -o data.o src/data.c
g++ -Wall -g -I/usr/include/mysql  -c -o type.o src/type.c
g++ -Wall -g -I/usr/include/mysql  -c -o tabs.o src/tabs.c
g++ -Wall -g -I/usr/include/mysql  -c -o rule.o src/rule.c
g++ -Wall -g -I/usr/include/mysql  -c -o dfsa.o src/dfsa.c
g++ -Wall -g -I/usr/include/mysql  -c -o syms.o src/syms.c
g++ -Wall -g -I/usr/include/mysql  -c -o help.o src/help.c
g++ -Wall -g -I/usr/include/mysql  -c -o talk.o src/talk.c
g++ -Wall -g -I/usr/include/mysql  -c -o init.o src/init.c
g++ -Wall -g -I/usr/include/mysql  -c -o link.o src/link.c
g++ -Wall -g -I/usr/include/mysql  -c -o load.o src/load.c
g++ -Wall -g -I/usr/include/mysql  -c -o bulk.o src/bulk.c
g++ -Wall -g -I/usr/include/mysql  -c -o shim.o src/shim.c
g++ -Wall -g -I/usr/include/mysql  -c -o wait.o src/wait.c
g++ -Wall -g -I/usr/include/mysql  -c -o time.o src/time.c
g++ -Wall -g -I/usr/include/mysql  -c -o read.o src/read.c
g++ -Wall -g -I/usr/include/mysql  -c -o flow.o src/flow.c
g++ -Wall -g -I/usr/include/mysql  -c -o exec.o src/exec.c
g++ -Wall -g -I/usr/include/mysql  -c -o feed.o src/feed.c
g++ -Wall -g -I/usr/include/mysql  -c -o send.o src/send.c
g++ -Wall -g -I/usr/include/mysql  -c -o echo.o src/echo.c
g++ -Wall -g -I/usr/include/mysql  -c -o post.o src/post.c
g++ -Wall -g -I/usr/include/mysql  -c -o calc.o src/calc.c
g++ -Wall -g -I/usr/include/mysql  -c -o leaf.o src/leaf.c
g++ -Wall -g -I/usr/include/mysql  -c -o name.o src/name.c
g++ -Wall -g -I/usr/include/mysql  -c -o atom.o src/atom.c
g++ -Wall -g -I/usr/include/mysql  -c -o term.o src/term.c
g++ -Wall -g -I/usr/include/mysql  -c -o envs.o src/envs.c
g++ -Wall -g -I/usr/include/mysql  -c -o pool.o lib/pool.c
g++ -Wall -g -I/usr/include/mysql  -c -o hash.o lib/hash.c
g++ -Wall -g -I/usr/include/mysql  -c -o text.o lib/text.c
g++ -Wall -g -I/usr/include/mysql  -c -o wrap.o lib/wrap.c
g++ -Wall -g -I/usr/include/mysql  -c -o inet.o lib/inet.c
g++ -Wall -g -I/usr/include/mysql  -c -o fork.o lib/fork.c
g++ -Wall -g -I/usr/include/mysql  -c -o boot.o lib/boot.c
g++ -Wall -g -I/usr/include/mysql  -c -o else.o src/else.c
g++ -Wall -g -I/usr/include/mysql  -c -o unit.o src/unit.c
date; g++ -g -o shim once.o bind.o mode.o data.o type.o tabs.o rule.o dfsa.o s
yms.o help.o talk.o init.o link.o load.o bulk.o shim.o wait.o time.o read.o flow
.o exec.o feed.o send.o post.o echo.o calc.o name.o leaf.o atom.o term.o tilt.o
pool.o hash.o text.o wrap.o inet.o fork.o boot.o else.o unit.o -L/usr/lib/mysql
-lmysqlclient -lm
Fri Aug  3 21:18:49 EDT 2007
make[1]: Leaving directory '/home/pippin/src/tws/src/shim-070803'
```

Figure A.1: Output from the make process

Appendix A

Related Command Scripts

A.1 The Makefile

A.2 Database and Table Setup

A.2.1 Database Setup

`sql/setup.sql`

`sql/table.sql`

`sql/names.sql`

`sql/perms.sql`

A.2.2 Table Creation and Recreation

`sql/create.sql`

`sql/drop.sql`

A.2.3 Common Table Creation Scripts

`sql/enum.sql`

`sql/syms.sql`

`sql/secs.sql`

`sql/subs.sql`

`sql/xact.sql`

`sql/load.sql`

Appendix B

Error Messages and Exceptions

Appendix C

Library Components and Usage

C.1 Singletons and Constants

C.1.1 The Components Singleton

An object also an instance of the single tree pattern has three data members of types Init, Data, and Root, each of which are const values, and only the last of which, root, is available to the parent class via its static member function root().

The data members are defined as values in order to allow cyclical references within the single tree instance constructor.

The second data member, Data, should include all the pure constant singleton objects of the application.

An object is pure const if it is itself const, and all its data members are either values or pure const references. An object is const if it only occurs in const contexts once constructed, that is accessible via a const reference, or as a value within a const context.

Perhaps more simply, a pure const object has no imperative references, and no hidden imperative aliases to itself created during construction. Such an object is also purely functional, that is completely defined during construction.

The first, Init, must set up references to all containers and other stateful objects referred to during data construction. E.g., if Data includes a constant cache to publicly named symbols also stored anonymously in a symbol table, then the empty symbol table must be created by Init, so that it is available when the lookup/insertion is done as the cache copy is looked up. Or, if a stateful derived class mode variable is used to choose among constant values within Data, again that mode object must be set up within Init.

The final data member value, of type Root, is meant to be the root reference to all the singleton objects in the application. It should include references to all the members of Init and Data.

An instance of the singletons class template has three states, as Init, Data, and Root are constructed one by one. It constrains singleton construction to occur in this order, provides a layer of indirection via its parent class to avoid header file coupling, and serves to encourage stratification of singletons by binding pattern, with Init providing the minimal state needed for application domain specific bootstrapping, Data all the pure functional objects that can be defined without runtime input beyond the command line arguments and configuration files, and Root a mixture of new, stateful object references and the simpler data members borrowed from Init and Data.

C.1.2 The Memory Allocator

See Figure C.1

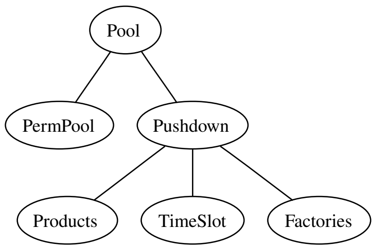


Figure C.1: Pools

C.1.3 The Token Type Hierarchy

All types in the lattice are either valid, or else the empty type None. Base types, the immediate parents of the empty type None, partition the type space. Derived types, all the valid non-base types, are type unions of other valid types.

Conversion from a specific to more general type is allowed, so that the edges in the type lattice represent the allowed type conversions, of types to their ancestors.

The stream lexical analyzer prefers most specific valid types, so that all well-formed data fields observed as input are given as type one of the base types. The required types used in record format specifications, on the other hand, may be named by more general derived types, in order to accomodate input variety.

Since base types have type codes a unique power of two, and derived types are bitwise disjunctive sums of their children, it follows that observed types are consistent with required types if and only if their bitwise sums are non-null. */

C.2 Strings, Ids, Hash Codes, and Text Buffers

The `m`^{3} String class, § C.2.1, provides constant character sequences, as opposed to the C++ standard library component of the same name, which allows a wide variety of imperative operations. The Buffer class, § C.2.3, is the much simpler, and so more limited `m`^{3} alternative to imperative standard strings. As elsewhere in the `m`^{3} library, the primary reason for avoiding the standard library components is the need to provide pool-based memory allocation via parameterized rather than static state. In this case, the freedom to expose to clients the results of hash code computation, § C.2.2, is an additional advantage.

C.2.1 The String Component

An instance of the string class is a triple *(octs, data, code)*, the number of octets, pointer to data, and hash code, respectively. The String class is unusual in providing two constructors. The first constructs the string from an ordinary null-terminated c-string, and using a object stack allocator, so that the result may be short-lived. The second takes . . .

C.2.2 Hash Code Computation

Hash code computation of 32 bit codes for word-aligned strings, null padded to be integral multiples of 12 bytes.

The C source code this version is based on, along with an explanatory article, was found at http://burtleburtle.net/bob/hash/evahash.html .

It came with the following notice:

> By Bob Jenkins, 1996. bob_jenkins@burtleburtle.net. You may use this code any way you wish, private, educational, or commercial. It's free.
>
> Use for hash table lookup, or anything where one collision in 2^{32} is acceptable. Do NOT use for cryptographic purposes.

I've adaptated it to C++ and specialized it for 32-bit x86. In particular, the hash code arithmetic has been rewritten to first copy the key to a word-aligned, null-padded,

class	functionality	plural	usage
TmpBuf	string temporary	false	internally, for temporaries
CatBuf	string catenation	false	internally, for temporaries
SqlBuf	null-term strings	true	read via the tws socket api
NulBuf	null-term strings	true	send ” ” ” ” ”
BarBuf	log format output	true	by the Logger, for requests

Table C.1: Buffer derived types and usage

multiple-of-12-byte buffer, which input works out to be much more tractable than raw, unaligned, arbitrary length C-style character strings.

Judging by the article, Jenkins is clearly aware that the word aligned computation is easier to read and write; he evidently chose the unaligned approach in order to satisfy the widest possible range of clients. Note, by the way, that he gives source for 64-bit hash code computation as well.

Word alignment allows each group of four character operations to be folded into a single unsigned integer expression, and the null padding allows the switch logic for string remainders to be folded into the main loop.

Although the loop traversal code is considerably different in style from that of Bob Jenkins' original, each iteration of the loop still works with 12-byte character sequences, and in Hash::State::operator+=(nat_0 key), the first three assignments correspond to Jenkins' combine() step, and the following loop, to the mix(), or permutation, step. Note that in order to fold the post-loop remainder computation into the main loop body, I chose to: (1) regularize the string remainder suffix combining step to conform with that used in the loop body; the original varied slightly between loop body and post-loop; and (2) use the string length as the salt, rather than attempt to combine it with the state vector part way through the computation.

As a result, the hash codes that are computed for strings of length greater than 8 differ from Jenkins' original code. Since I still use all the original input information, and the permutation step is unchanged, the likelihood of collisions should also be unchanged.

C.2.3 Buffer Types and Their Uses

See Figure C.2 for the derivation hierarchy of the buffer classes.

See Table C.1.

A buffer is plural when multiple distinct tokens can be appended to, and distinguished within, the buffer all at one time. E.g., null-terminated strings in a NulBuf, or vertical bar terminated character sequences within a BarBuf. Alternatively, a buffer is singular when each character sequence appended to the buffer is catenated directly

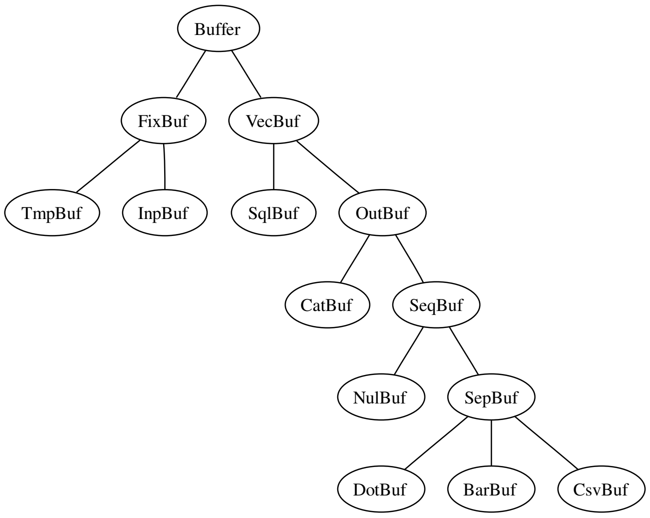


Figure C.2: Buffer variants

adjacent to the previous, the one example being the CatBuf.

Source streams have one msg buffers, stream, providing fixed size blocks as the targets of system input routines,

Cat buffers occur in Tables, the Dbms, the Logger, and the Historian; Nul buffers, in Tables, twice;

There is currently only one instance of a TmpBuf, in Source, where it is shared with a tokenizer for the duration of a parse, and from there used by Cursor::probe, Parser::scan, and Tokens::scan itself.

Reading Characters

Whether input is via reads from a socket, or file, and in the latter case, by using the fread or read api, still there is the potential for incomplete records due to partial reads. In addition to the possibility that the api calls might return a partially filled buffer, that buffer is in any case of fixed size, and the size is used to limit the number of characters read, so that even when complete records are ready, they might be split anyway.

Due to this possibility of partial reads, callers to the Buffer read procedures are expected to copy characters out of the read buffer, e.g. to a queue, and allow them to accumulate until a complete record is available.

The Hash Map Component

Let a domain type $\mathcal{D}$ provide equality and a hash function `id()`, and a range type $\mathcal{R}$ provide `operator*()`, an inverse operation returning an exemplar of type $\mathcal{D}$. Then for the class template *Function*, an instance of type `Function<D,R>` maintains a function mapping between pairs of the domain and range according to the calls made to the operator `R::invert()`. More precisely, for a map F; a triple (D, R, S) where D is in the domain of F, both R and S are in the range, and D is the exemplar of R, so that $*R = D$; and given a call `F.invert(R)` returning S, so that $(D, S) \in F$, then $(D, R) \in F \Leftrightarrow R = S$.

The operational interpretation is that the pair (D, R) is in the hash F when the inverse $*R$ is D; there has already been at least one call `F.invert(R)`; and for the first occurrence of such a call, the key D did not already exist in the hash prior to that call. Or, less formally, given an attempted insertion of R into F, and as long as R "got there first" as an argument to `F::invert()`, that is before any other member of $\mathcal{R}$ having D as exemplar, then $F(D) = R \Leftrightarrow *R = D$.

The class template `Function` is monotonic, so that pairs once added are not removed. The uncertainty about whether a call `F.invert(R)` achieves its intended result of inserting R into F follows directly from the possibility that multiple instances of R may share a common exemplar. For the common case where the operation `R::operator*()` is one-to-one, then its inverse is by definition a function, the map F implements that function, and calls to invert necessarily succeed, so that for $*R = D$, the call `F.invert(R)` returns R, and we know $(D, R) \in F$.

The class template Function provides as operators to derived types the previously mentioned bind operation `invert()`; the lookup operation `select()`, which returns a pointer of type R; and the traditional `size()` and `capacity()` operations. The open buckets are kept in sort order, and lookup in collision sets uses binary search.

The Function Type and the Domain, Subset, and Folder Derived Types

For a domain D and range R, the base class Function provides the membership assertion constraint invert() for function pairs (D, R), the membership lookup operator select(D), and also the traditional size() and capacity() queries. Since names in Function are protected, that template is not directly useful to clients.

There are three derived class templates:

- Domain provides a history of all set members seen up to the present.
- Subset exposes the membership pair constraint, and allows the client to test for membership.
- Folder provides as well the operator clear() to rewrite history.

Since Domain provides only the membership constraint, it is fully declarative, while Subset, with the membership pair binding operator, is monadic, and Folder, with the potential to destroy information via clear(), is fully imperative.

In return for providing the convenience of the declarative map operator[], Domain requires a third template parameter U, the type of the factory for the universe of the range type R; that the factory provide an operator create(D) returning a new instance of R; and that a reference to the factory be provided during construction. While Domain is used by the identifier factory Identifiers, most client mappings are implemented using the more flexible Subset class template.

Initialization of the Components Singleton

The Components singleton includes as its first data member a reference to an object of class Memory, the memory allocator singleton. Memory is a factory for allocator pools, which themselves contain memory blocks chopped from (much larger) memory mappings. The memory allocator, in addition to creating pools, keeps track of all pools, blocks, and mappings.

Clients typically work with the pools; blocks are hidden, and client access to the memory allocator is also limited to restrict pool construction. The pools themselves include a reference back to the memory allocator so that new blocks can be requested as needed.

The static BootMemory boot() method constructs Memory bottom-up. An initial mapping is first requested from the system, and a BootMemory object created from it, during which: a block is constructed; a permanent pool, the seed, is constructed using the block; and the singleton memory allocator Memory is constructed from the seed.

Data members in the BootMemory object are values, declared as such both to simplify memory allocation accounting, and to allow forward references, in particular of Memory as an argument to the seed pool constructor. Fortunately, permanent pools also take an initial block as one of the arguments to the ctor, and do not actually call on the Memory reference data member until that first block is exhausted.

The boot() method goes on to construct and return the Components object, the root singleton for all library components, including Memory.

Runtime Sized, Fixed Dimension Array Constants

The `Store`, `Table`, `Array`, `Block`, and `Index` class templates provide runtime sized, fixed dimension arrays, with indexing and bounds checking varying as described in Table C.2; the sizeof values were taken using `gcc 3.4.4`.

Store is essentially a wrapper for a pointer to a logic variable, and all the other array

	indexed by (given index i and base j)	sizeof
Store	i unchecked	4
Table	i-j, bounds checked	12
Array	i-0, ditto	12
Block	i-1, ditto	12
Index	via binary search	12
Value	as for Table, though cells are values	12

Table C.2: Indexing for table template classes

types other than Value inherit from Store, so that for all the reference-semantic tables, the cells are logic variables, and values once bound to a cell are immutable. All the array types other than Store maintain their index range as a half-open interval, which is why the sizeof values jump from 4 to 12.

The Binary Search Object

Given that the template type T provides an id() operator returning a natural number; in the ctor, for the block pointer parameter, that the referred-to block has size equal to the ctor parameter n, itself less than the size of the hardware address space; and that the block be filled with valid object pointers, the referred-to objects of which are in ascending order according to operator id(); then an instance of `Search<T>`returns, upon dereference, a pointer to an object of T having id() equal to the ctor key parameter, if such an object exists in the block, and zero otherwise.

The Timestamp Function

For a processor with 32bit words, and where the long long unsigned (Long) type is 64bits, read the processor time stamp counter into the two halves of a Long reference parameter.

File Descriptor Setup for Pseudo Terminals

PseudoTerm is a factory for ForkResult objects describing pty connections, or else the system call failure that prevented their setup. The algorithm used here is loosely based on that of [8]; see pp. 692–693 there for an example in C.

For system V platforms, and in particular Linux and Solaris, the slave becomes the controlling terminal on open(). The ioctl call with TIOCSCTTY is used on BSD (and OS X) to acquire a controlling terminal; linux also allows this call, although it is, as stated above, not necessary.

For now, the child_image() procedure calls acquire_ct(), and so performs the ioctl operation unconditionally. As an alternative, the PseudoTerm factory could have architecture-specific control, and although the resulting mode data would be undesirable, it may be necessary to add such in the future.

PseudoTerm is a *non-reentrant* factory for pty setup in support of coprocesses; the plumbing() method may be used as a replacement for fork(), with the pid parameter indicating to the caller whether the return is in the parent or child.

The object bifurcation that allows this to work occurs because the singleton heap allocation is of MAP_PRIVATE memory, and changes to such memory are private (in practice, by COW) to each process, and the parent and child each get, in effect, individual copies of the PseudoTerm object.

Making the plumbing method reentrant, and so thread-safe, would require that: the three scalar reference data members be replaced with a stack of tuples; a lock be used to control access to the stack; calls to the ptsname() system call, which is not reentrant, be locked; and that the return value for the slave pty name be copied out into distinct strings, those also stored on the stack. Note additionally that the strerror() function used in error reporting is also not required to be reentrant.

C.3 Equivalence Classes and Logic Variables

C.3.1 Representing Equality Theories Via Disjoint Set Union

Although many application domain classes can use simple pointer comparison to determine equality, sometimes multiple objects may be grouped into equivalence classes, so that a more coarse-grained equality theory is needed.

The disjoint set template class `EqClass` provides, in addition to the constructor, the public operations `operator+=()` and `operator==()`, the former to take the union of two equivalence classes, and the latter to determine whether two representatives are in the same equivalence class. The `operator+=()` method implements set union by rank with path compression, and is nearly constant time; see Ch 22 of [4] for details.

What is the Logical Reading?

The set union operator creates a binding between objects, after which they are in the same equivalence class, which raises the question of the logical meaning of the union operator; it appears to provide some kind of equality constraint, but what kind?

For the common case where the EqClass template is instantiated for some finite domain type, the union operator can be usefully compared to equality constraints over that domain.

If the notion of finite domain types seems unfamiliar, consider the finite set: $\{red, green, blue\}$, which provides symbols that may be used in logical statements for the related domain. Given e.g., two pixel variables P and Q with values constrained to be in $\{red, green\}$ and $\{green, blue\}$ respectively, then the additional knowledge that the pixels are located in a region filled with a common color gives as well the equation P = Q, and the variables are constrained by set intersection to the value $green$.

So what, if anything, does disjoint set union over a finite domain have to do with equality for that domain? Each non-trivial binding by the union operator gives a less precise equality theory, so that information is discarded. This is in opposition to domain constraints as above, which monotonically add information to the system. For a given finite domain, equations and disjoint set union are in some sense duals, as domain sets are constrained by intersection or generalized by union. The set union operator is actually second order, mapping from one equality theory to another; it's no help whatsoever in testing equality constraints for a particular theory.

Common Case Usage

That's why for the common case disjoint set objects are used as part of a two-stage computation, with bindings created in the first step, and equality tests made only in the second, once the equality theory is known. E.g., given a set of type constants for both object and object index types, and where an instance of the index type can be used to lookup the related object, rules relating object and index types are added at initialization, and comparisons made after that stage.

Note that it is up to the client to use privacy to prevent unwanted late stage union operations; friendship can't be used in the template to limit access to the union operation, since the template type T is possibly a typedef, not necessarily a class, and so not qualified to be a class key. Const annotation is of no help either; the optimizations to the union-find algorithm that change the time cost from linear to effectively constant time make the core `find_set()` operation thoroughly imperative in nature.

C.3.2 Single Assignment Pointers as Logic Variables

C.4 Block-Doubling Containers

C.4.1 Block-Doubling Via Handles

Handle

A `Handle<T,C>`is essentially a wrapper to an imperative T** data member intended for use by a container of type C. The handle knows the size of the pointer block, and has the facilities to free and reallocate a larger such block as needed. The related container of type C, by the same token, must provide a replication operation copy() which accepts two handles source and target and copies all the T* values from the first to the second.

More literally, a handle is an imperative quintuple:

Handle(Copy, Cont, Data, Exp2, Used)

where Copy is the Copier singleton, for allocation and reallocation; Cont, the container of type C responsible for the copy update after reallocation; Data, a T** pointer to a region of size 2^{Exp2}; and Used, the largest offset from the base of Data yet having been bound to an application-defined value.

The `Handle<>`interface provides read access via T* operator[](nat), and write access using void bind(nat, T* e). Access is checked, and, letting N stand for Maps::Size, Table C.3 gives the four cases for an index i to bind() for a handle with size n.

	description	action

1	$0 <= i < n$	Data[Used..i] := 0
		Data[i] := e
		Used := max(Used, i)

2	$n <= i < 2n, 2n <= N$	enlarge and copy Data, and bind as in 1
3	$n <= i < 2n, 2n > N$	throw an exception
4	$2n <= i$	throw an exception

Table C.3: Case analysis for `Handle::bind()`

Although test code may allow initialization in order to simplify debugging, for the finished component, blocks will not be initialized when first allocated, in order to minimize the time cost of dynamic allocation for large containers. Conversely, the handle must ensure that the unassigned region [0..Used] is initialized, which is the reason for the incremental initialization in the bind() operator (and operator[], as well).

As a result, though access time is still constant time for sequentially filled vectors, it is only amortized constant time for hash tables. In practice this isn't very important, for a number of reasons, and for the application programmer who disagrees, it is easy enough to choose a small initial block size.

Handle creation is performed by the friend Copier on behalf of related containers. The size parameter Exp2 is interpreted as a request for a block of size 2^{Exp2}, so that Exp2 should be the log base 2 of the desired block size. The size parameter is of type unsigned character, or Char, to remind the application programmer that the size parameter is *not* interpreted as the absolute block size. The value of the size parameter is restricted to fall in 0, 31, else an exception is thrown, and the actual size of the allocated block is furthur clamped between 2^6 and some implementation defined limit, typically the memory allocator map size, at the time of this writing `0x1000000`, or $2^2 4$, around 16 meg.

frequency	object size	word size	floor ratio	factor
132	4	4	1	1
48	1	4	0	1
1	12	4	3	4

Table C.4: Frequency and other statistics for instances of `Handle <>`

The Area Component

The pair (Exp2, Data) is in the relation Area when Data points to an memory area of 2^{Exp2} words

C.4.2 Vectors and Queues

Reference and Value Vectors, Stacks, Queues, and Scrolls

See Figure C.3

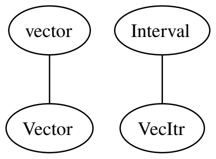


Figure C.3: Vector implementation

See Figure C.4

See Figure C.5

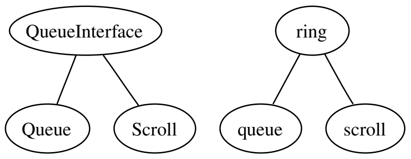


Figure C.4: Queue implementation

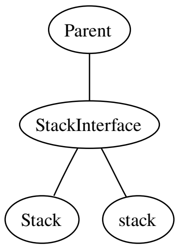


Figure C.5: Stack implementation

The Vector and vector templates provide reference and by-value semantics, with append storing pointers and values, and the index operator returning references and values, respectively.

The Stack and stack templates provide reference and by-value semantics, respectively, while sharing an otherwise common interface. See the parent classes `Vector<T>`and `vector<T>`for details.

C.4.3 Splay Trees

The component code for the `Tree<>`class template implements splay trees, as defined and analyzed in [5], and is based on sources by Daniel Sleator, who released them to the public domain. The original, `top-down-splay.c`, can also be obtained drilling down from Daniel Sleator's home page. He explains in comments from that source file as follows:

> "Splay trees", or "self-adjusting search trees" are a simple and efficient data structure for storing an ordered set. The data structure consists of a binary tree, without parent pointers, and no additional fields. It allows searching, insertion, deletion, deletemin, deletemax, splitting, joining, and many other operations, all with amortized logarithmic performance. Since the trees adapt to the sequence of requests, their performance on real access patterns is typically even better.

The splay algorithm in the code by Sleator is adapted from simple top-down splay, as given at the bottom of p. 669 in [5]. Sleator goes on to explain his adaptation:

> The chief modification here is that the splay operation works even if the item being splayed is not in the tree, and even if the tree root of the tree is nil. So the line:
>
> $$\texttt{t} = \texttt{splay}(\texttt{i}, \texttt{t});$$
>
> causes it to search for item with key i in the tree rooted at t. If it's there, it is splayed to the root. If it isn't there, then the node put at the root is the last one before nil that would have been reached in a normal binary search for i. (It's a neighbor of i in the tree.) This allows many other operations to be easily implemented ...

See Figures C.6 and C.7.

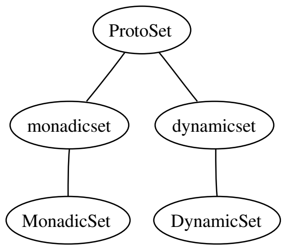


Figure C.6: Set maps

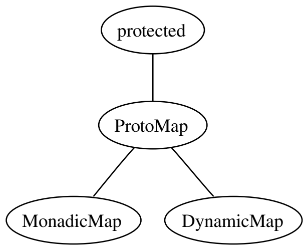


Figure C.7: Map inheritance

C.4.4 Hashed Functions

See Figure C.8

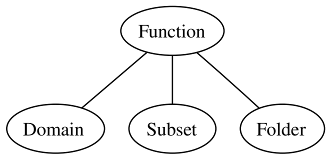


Figure C.8: Domain maps

C.5 Mostly Declarative Tables and Sequences

C.5.1 Logical Tables

The Table `Build <>` Factory

`Build<T,S>`is a short-lived factory template used to create constant tables of user-defined objects or primitive values. The input tables for these templates are arrays of values, e.g., object pointers or natural numbers, and must have sizeof at least S = N * sizeof(V), where N is the table dimension, and V the value type, T* or T as T is an object or value.

The Build template has five factory methods, value(), array(), block(), index(), and table(), each producing an indexable result, the similarly named array type from table.h. The value() method is the only one of these to produce a table of primitive values, such as natural numbers, with the others expecting an array of object pointers as input.

The first three of the aforementioned methods simply construct and fill a Value, Array, or Block table, respectively, while given their preconditions, the last two enforce ordering constraints as well. Briefly, given an input vector, index() sorts the possibly sparse values and returns an Index, while table() requires that the inputs be a sequence permutation, putting that permutation in sequence and returning a Table.

More precisely, index() and table() have the following preconditions and results: both require that objects of type T be totally ordered via `operator<()`, which is used in constructing an intermediate heap; and both copy those inputs in ascending order from the temporary heap to the result array. The method table() in addition requires that T have a member function id() returning an integral type ordered consistently with `operator<()`, which it uses for sequence checking.

The Index array produced by index() uses binary search to provide operator[], so that sparsely indexed inputs are accepted, at the cost of lgN access time. The Table array produced by table() provides based array indexing, so that for an object T in a result Tab, T == Tab[T.id()] To achieve this invariant, table() requires that the inputs once sorted be a sequence according to id(), that is be in ascending order without gaps or duplicates in the index values, and throws an exception otherwise.

For the common case where table() is applied to already ordered input vectors, and since heapifying the input vector takes linear time, does not disturb the already sorted input array, and heap decrement is constant time for an input vector that is already in sort order, then the sort operation adds only a constant factor to the time cost of table construction. E.g., for a Type T, Pool_pda p, and an array of object pointers tab, where the objects are in sequence by T::id(), the expressions below construct, in linear time, tables indexed by T::id() with unit and log index times, respectively:

```
    Build<T, sizeof(tab)>(p, tab).table();
    Build<T, sizeof(tab)>(p, tab).index();
```

C.5.2 Sequences

Lists

A proper list is bound, with tail other than self, while unbound and self-cyclical lists are false, with nil in particular constructed to be self-cyclical. The bool test compares logic var values – pointers – rather than the variables themselves, which, for a previously unbound tail, would create a cycle between the newly bound tail and its preceeding cons cell.

Lifo

The List copy constructor is meant to provide limited client access, (e.g., to construct a list iterator), without actually letting a reference escape; the value return is by design.

Cells

`Cell<T>` occupies an intermediate point in type safety between `Pair<T>`, and raw `void` pointers. It provides more flexibility than pairs, since in the binary constructor, cells allow an untyped, imperative body, while pairs accept at most a list tail. Cells wrap a head reference to the template type with a `void*` body, useful as the argument to placement new during list and pair construction. As wrappers, they provide an explicit type, so obviating the need to pass naked `void*` parameters around.

Cells have been used in the append operator for `Tape<T>`, and the constructors for `List<T>`and `Lifo<T>`, where they are passed in by value, allowing those types also to be constructed by value, without needing pool allocation. E.g., the Pool class has a `Tape<Block>`pda, for allocation blocks, and one of its derived classes, PushDown, has a `Lifo<Mark>`pda for memory marks; callers to both the append operation and the template ctors take advantage of this ability to pass pairs by value.

The Unit Type

The hard truth is that neither C nor C++ have a true unit type, making do instead with 0. If we want a real unit type for recursively defined generic lists, we need to define one, and use casts within the template list classes to convert it to whatever the instantiated type might be.

In the ideal case, the unit type has no internal structure; it is simply a unique address arbitrarily designated to terminate lists. Whatever happens to its internal value, that address remains constant, so that for properly written code, it continues to serve its purpose even if the internal value is corrupted.

Still, since the unit type is shared across all lists in a program, we must consider the possibility that one list client will fail to test for the empty list, and attempt to treat the unit object as an ordinary cons cell; and that other, later code, will then go on to make the same mistake again, in which case we wish to contain the damage and simplify debugging.

This dictates, first, that the unit type be the size of an ordinary cons cell, to shield following memory; second, that the ”head” be zero, so that loops attempting to dereference an empty list and interpret the head as a reference to a data value crash immediately; and third, that the ”tail” be non-zero, and so treated as bound, so that it can not be bound to real cons cells.

None of this is an issue for lists and links, where the unit type is used as a private value of the class. Even for lisp, or lifo, lists, it is meant as a private value for use by the default constructor, and again, there is no problem. The nature of a unit type, however, is to be used widely, and in ways which the original designer did not predict, and so perhaps these precautions are sensible. Certainly they are effectively free, since the unit type is the prototypical singleton; and certainly they are reasonable, since after all we are using casts and so are not fully protected by the type system.

The bottom line is that clients that fail to test for the empty list before dereferencing for head values will have problems, just as if the list was not even terminated in the first place. There is a sense in which we can’t get past something that acts more or less like zero as a unit type, at least until algebraic type inference is added to C++. The real value of the unit type is not for the application clients – if they use lists properly, they’ll never know it exists – it’s for the template classes, which can avoid nasty bugs by properly terminating lists, so that an unbound list tail is not accidently bound to some comparison value via equality. In particular, list differences are terminated when read, so that list usage naturally has two modes, first declarative creation with logic variables, followed next by functional traversal with fully bound references.

C.6 System Call Wrappers

Bibliography

[1] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. Stroustrup has a page for TC++PL: http://www.research.att.com/~bs/3rd.html.

[2] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995. See http://web.engr.oregonstate.edu/~budd/Books/leda/.

[3] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. See http://users.rcn.com/jcoplien/.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990. See http://theory.lcs.mit.edu/~clr/.

[5] Sleator and Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985. The file `top-down-splay.c` is available from http://www.link.cs.cmu.edu/splay/.

[6] Bob Jenkins. Hash functions for hash table lookup. On his site at http://burtleburtle.net/bob/hash/evahash.html, written over 1995 – 1997.

[7] Bob Jenkins. Hash functions. *Dr. Dobbs Journal*, September 1997. http://www.burtleburtle.net/bob/hash/doobs.html.

[8] W. Richard Stevens and Stephen A. Rago. *Advanced Programming in a Unix Environment*. Addison-Wesley, second edition, 2005. The publisher's page is currently http://www.aw-bc.com/catalog/academic/product/0,1144,0201433079,00.html.

[9] Douglas E. Comer, David L. Stevens, and Michael Evengelista. *Internetworking with TCP/IP*, volume III: Client-Server Programming and Applications. Prentice Hall, 2000. See http://www.cs.purdue.edu/homes/dec/netbooks.html.

[10] Paul DuBois. *MySQL*. Sams, third edition, 2005. The doorstop has its own home page at http://www.kitebird.com/mysql-book/.

[11] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. Amazon.

[12] Robert Mecklenburg. *Managing Projects with GNU Make*. O'Reilly, third edition, 2005. Currently available online at http://www.oreilly.com/catalog/make3/book/index.csp.

[13] Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, 2000. The Goat book is available online at http://sourceware.org/autobook/.

[14] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1984. Knuth's page for the Computers & Typesetting books is http://www-cs-faculty.stanford.edu/~knuth/abcde.html.

[15] Leslie Lamport. *LaTeX: User's Guide and Reference Manual*, 1986. Lamport's publications.

[16] Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing Graphs with Dot*, February 2002.

[17] Nick Drakos. *The LaTeX2html Translator*, 2007. www.latex2html.org.

[18] Burton G. Malkiel. *A Random Walk Down Wall Street*. Norton, first edition, 1973. First published in 1973, and now in its 9th edition; see http://www.princeton.edu/~bmalkiel/.

[19] Benoit B. Mandelbrot. *The (Mis) Behavior of Markets : a Fractal View of Risk, Ruin, and Reward*. Basic Books, 2004. Amazon.

[20] Emanuel Derman. *My Life as a Quant: Reflections on Physics and Finance*. Wiley, 2004. Derman's *Writings on Quantitative Finance* page: http://www.ederman.com/new/index.html.

[21] Benjamin Graham. *The Intelligent Investor*. Harper & Row, fourth edition, 1973. Amazon.

[22] Peter Lynch with John Rothchild. *One Up On Wall Street*. Simon & Schuster, 1989. Amazon.

[23] Peter Lynch with John Rothchild. *Beating the Street*. Simon & Schuster, 1993. Amazon.

[24] Jack D. Schwager. *Market Wizards*. HarperBusiness, 1989. Amazon.

[25] Jack D. Schwager. *The New Market Wizards*. HarperBusiness, 1992. Amazon.

[26] Jack D. Schwager. *Stock Market Wizards*. HarperBusiness, 2001. Amazon.

[27] Edwin Lefévre. *Reminiscences of a Stock Operator*. Wiley, 2006. First published in 1923, this is effectively a fictionalized biography of Jesse Livermore; there are reviews at Amazon.

[28] Chester W. Keltner. *How to Make Money in Commodities*. The Keltner Statistical Service, 1004 Baltimore Ave., Kansas City 5, MO, 1960.

[29] Ralph M. Ainsworth. *Profitable Grain Trading*. Traders Press, Greenville, SC, 1980. First published in 1933, and reprinted by Traders' Press.

[30] J. Welles Wilder, Jr. *New Concepts in Technical Trading Systems*. Trend Research, P.O. Box 450, Greensboro, NC 27402, 1978. Amazon.

[31] Bill Pippin. *Optimizing Threads of Computation in Constraint Logic Programs*. PhD thesis, The Ohio State University, January 2003. Online copy.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995. For more about the GoF book, see: http://hillside.net/patterns/DPBook/DPBook.html.