19.9 *Case Study and Lab:* Stock Exchange

In this section we consider a larger, more realistic case study: a miniature stock exchange. A stock exchange is an organization for trading shares in publicly owned companies. In the OTC (Over The Counter) system, stocks are traded electronically through a vast network of securities dealers connected to a computer network. There is no physical "stock exchange" location. In the past few years, thousands of investors have started trading stocks directly from their home computers through Internet-based online brokerage firms.

In this project we will program our own stock exchange and electronic brokerage, which we call *SafeTrade*. What do we mean by "safe"? Not too long ago, some unscrupulous online brokerage firms started encouraging a practice called "day trading," in which traders hold a stock for a few hours or even minutes rather than months or years. As a result, quite a few people lost all their savings and got into debt. Actually, this case study would be more appropriately placed in Chapter 11: in the U.S. code of bankruptcy laws,[uscode] "Chapter 11" deals with reorganization due to bankruptcy. With our *SafeTrade* program, you stay safely offline and out of trouble and don't pay commissions to a broker.

> **We picked this project because it illustrates appropriate uses of many of the interfaces and classes in the Java collections framework. This project is large enough to warrant a meaningful team development effort.**

❖ ❖ ❖

The stock exchange system keeps track of buy and sell orders placed by traders and automatically executes orders when the highest "bid" price to buy stock meets the lowest asking price for that stock. Orders to buy stock at a certain price (or lower) or to sell stock at a certain price (or higher) are called "limit" orders. There are also "market" orders: to buy at the currently offered lowest asking price or to sell at the currently offered highest bid price.

Each stock is identified by its trading symbol. For example, Sun Microsystems is "SUNW" and Microsoft is "MSFT." In the real world, very small stock prices may include fractions of cents, but in *SafeTrade* we only go to whole cents.

SafeTrade's brokerage maintains a list of registered traders and allows them to log in and trade stocks. The program keeps track of all active buy and sell orders for each

stock. A trader can place buy and sell orders, specifying the price for a "limit" order or choosing a "market" order. Each order deals with only one stock. The order for a given stock holds six pieces of information: a reference to the trader who placed it, the stock symbol, the buy or sell indicator, the desired number of shares to be bought or sold, the market or limit indicator, and the price for a limit order. *SafeTrade* acknowledges a placed order by sending a message back to the trader.

When a new order comes in, *SafeTrade* checks if it can be executed and, if so, executes the trade and reports it to both parties by sending messages to both traders. In *SafeTrade*, all orders are "partial" orders. This means that if an order cannot be executed for the total number of shares requested in it, the maximum possible number of shares changes hands and an order for the remaining shares remains active.

A trader can also request a quote for a stock. The quote includes the last sale price, the price and number of shares offered in the current highest bid and lowest "ask" (sell order), the day's high and low price for the stock, and the volume, which is the total number of shares traded during the "day." (In our model, a "day" is one run of the program.)

> **The details of how orders are executed and at what price and the format for a stock quote are described in the *Javadoc* documentation for the `Stock` class.**

SafeTrade does not keep track of the availability of money or shares on the trader's account. If you want, you can add this functionality. For example, you can keep all transactions for a given trader in a list and have a separate field to hold his or her available "cash."

At a first glance, this appears to be a pretty large project. However, it turns out that with careful planning and an understanding of the requirements, the amount of code to be written is actually relatively small. The code is simple and consists of a number of small pieces, which can be handled either by one programmer or by a team of several programmers. We have contributed the main class and a couple of GUI classes.

One of the challenges of a project like this is testing. One of the team members should specialize in QA (Quality Assurance). While other team members are writing code, the QA person should develop a comprehensive test plan. He or she then tests the program thoroughly and works with programmers on fixing bugs.

To experiment with the executable program, set up a project in your IDE with the `SafeTrade.java` and `SafeTrade.jar` files from `JM\Ch19\SafeTrade`. (Actually, `SafeTrade.jar` is a runnable jar file, so you can run *SafeTrade* by just double-clicking on `SafeTrade.jar`.)

`SafeTrade`'s `main` method creates a `StockExchange` and a `Brokerage` and opens a `LoginWindow`. To make program testing easier, `main` also lists several stocks on the `StockExchange` —

```
StockExchange exchange = new StockExchange();
server.listStock("GWP", "GridWorldProductions.com", 12.33);
server.listStock("NSTL", "Nasty Loops Inc.", 0.25);
server.listStock("GGGL", "Giggle.com", 28.00);
server.listStock("MATI", "M and A Travel Inc.", 28.20);
server.listStock("DDLC", "Dulce De Leche Corp.", 57.50);
server.listStock("SAFT", "SafeTrade.com Inc.", 322.45);
```

— and registers and logs in a couple of traders at the `Brokerage`:

```
Brokerage safeTrade = new Brokerage(exchange);
safeTrade.addUser("stockman", "sesame");
safeTrade.login("stockman", "sesame");
safeTrade.addUser("mstrade", "bigsecret");
safeTrade.login("mstrade", "bigsecret");
```

❖ ❖ ❖

Our design process for *SafeTrade* consists of four parts. The first part is structural design, which determines which data structures will be used in the program. The second part is object-oriented design, which determines the types of objects to be defined and the classes and interfaces to be written. The third part is detailed design, which determines the fields, constructors, and methods in all the classes. The fourth part is developing a test plan. We are going to discuss the structural design first, then the classes involved, and after that the detailed design and testing.

1. Structural design

Our structural design decisions are summarized in Table 19-3. We are lucky to have a chance to use many of the collections classes discussed in this chapter.

Data	*interface* => class
Registered traders	`Map => TreeMap<String, Trader>`
Logged-in traders	`Set => TreeSet<Trader>`
Mailbox for each trader	`Queue => LinkedList<String>`
Listed stocks	`Map => HashMap<String, Stock>`
"Sell" orders for each stock	`Queue => PriorityQueue<TradeOrder>` (with ascending price comparator)
"Buy" orders for each stock	`Queue => PriorityQueue<TradeOrder>` (with descending price comparator)

Table 19-3. Structural design decisions for *SafeTrade*

We have chosen to hold all registered traders in a `TreeMap`, keyed by the trader's login name. A `HashMap` could potentially work faster, but the response time for a new registration is not very important, as long as it takes seconds, not hours. A `HashMap` might waste some space, and we hope that thousands of traders will register, so we can't afford to waste any space in our database. For similar reasons, we have chosen a `TreeSet` over a `HashSet` to hold all currently logged-in traders.

A trader may log in, place a few orders, and log out. Meanwhile, *SafeTrade* may execute some of the trader's orders and send messages to the trader. But the trader may already not be there to read them. So the messages must be stored in the trader's "mailbox" until the trader logs in again and reads them. This is a perfect example of a queue. In *SafeTrade*, a mailbox for each trader is a `Queue<String>` (implemented as a `LinkedList<String>`).

SafeTrade also needs to maintain data for each listed stock. A stock is identified by its trading symbol, so it is convenient to use a map where the stock symbol serves as the key for a stock and the whole `Stock` object serves as the value. The number of all listed stocks is limited to two or three thousand, and the list does not change very often. *SafeTrade* must be able to find a stock immediately for real-time quotes and especially to execute orders. Traders will get upset if they lose money because their order was delayed. Therefore, a good choice for maintaining listed stocks is a hash table, a `HashMap`.

Finally, *SafeTrade* must store all the buy and sell orders placed for each stock in such a way that it has quick access to the highest bid and the lowest ask. Both adding and

executing orders must be fast. This is a clear case for using priority queues. We need two of them for each stock: one for sell orders and one for buy orders. For sell orders the order with the lowest asking price has the highest priority, while for buy orders the order with the highest bid price has the highest priority. Therefore, we need to write a comparator class and provide two differently configured comparator objects — one for the buy priority queue and one for the sell priority queue.

2. Object-oriented design

Figure 19-20 shows a class diagram for *SafeTrade*. The project involves nine classes and one interface. We have provided three classes and the interface: the application launcher class `SafeTrade`, the GUI classes `LoginWindow` and `TraderWindow`, and the `Login` interface. Your team's task is to write the remaining six classes. The following briefly describes the responsibilities of different types of objects.

Shams

Leo

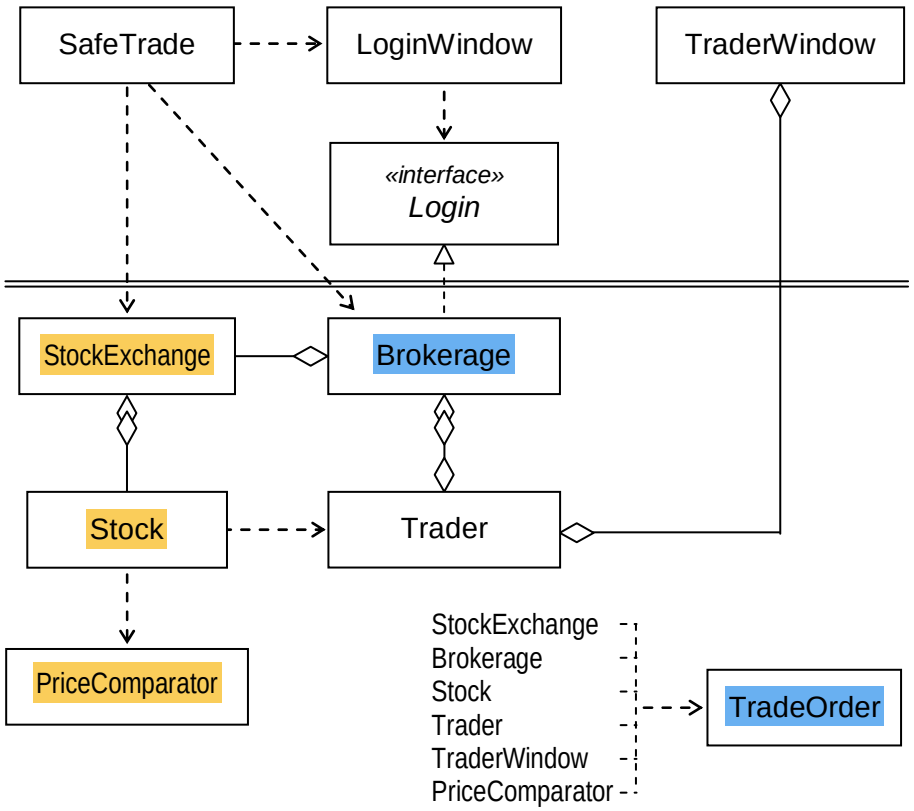

Figure 19-20. Class diagram for the *SafeTrade* program

Our classes:

We have already mentioned `SafeTrade`, the class that has `main` and starts the program.

The `LoginWindow` accepts a user name and a password from a user and can register a new trader.

> **The `Login` interface isolates `LoginWindow` from `Brokerage`, because logging in is a common function: we want to keep the `LoginWindow` class general and reusable in other projects.**

The `TraderWindow` object is a GUI front end for a `Trader` (Figure 19-21). Each `Trader` creates one for itself. The `TraderWindow` collects the data about a quote request or a trade order and passes that data to the `Trader` by calling `Trader`'s methods.

Figure 19-21. A trader window

Your classes:

The `StockExchange` keeps a `HashMap` of listed stocks, keyed by their symbols, and relays quote requests and trade orders to them.

The `Brokerage` keeps a `TreeMap` of registered traders and a `TreeSet` of logged-in traders. It receives quote requests and trade orders from traders and relays them to the `StockExchange`.

A `Stock` object holds the stock symbol, the company name, the lowest and highest sell prices, and the volume for the "day." It also has a priority queue for sell orders and another priority queue for buy orders for that stock.

> **The `Stock` class is more difficult than the other classes, because it includes an algorithm for executing orders.**

A `PriceComparator` compares two `TradeOrder` objects based on their prices; an ascending or a descending comparator is passed as a parameter to the respective priority queue of orders when the priority queue is created.

A `Trader` represents a trader; it can request quotes and place orders with the brokerage. It can also receive messages and store them in its mailbox (a `Queue<String>`) and tell its `TraderWindow` to display them.

A `TradeOrder` is a "data carrier" object used by other objects to pass the data about a trade order to each other. Since all the other classes depend on `TradeOrder`, it makes sense to write it first. This is a simple class with fields and accessor methods that correspond to the data entry fields in a `TraderWindow` (Figure 19-21).

3. Detailed design

The detailed specs for the *SafeTrade* classes have been generated from the Javadoc comments in the source files and are provided in the `SafeTradeDocs.zip` file in `JM\Ch19\SafeTrade`. Open `index.html` to see the documentation.

4. Testing

Proper testing for an application of this size is in many ways more challenging than writing the code. Entering and executing a couple of orders won't do. The QA specialist has to make a list of possible scenarios and to develop a strategy for testing them methodically. In addition, the tester must make a list of features to be tested. The most obvious are, for example: Do the "login" and "add user" screens work? Does the "Get quote" button work? When a trader logs out and then logs in again, are the messages preserved? And so on.

The `Stock` / `PriceComparator` subsystem can be tested separately. Write a simple console application for this. You will also need a *stub* class (a greatly simplified version of a class) for `Trader`, with a simple constructor, which sets the trader's name, and one method `receiveMessage(String msg)`, which prints out the trader's name and `msg`. Test the `StockExchange` class separately, using a stub class for Stock. Test the Brokerage / Trader subsystem separately using a stub class for StockExchange.