Chapter 14

Event-Driven Trading Engine Implementation

This chapter provides an implementation for a fully self-contained event-driven backtest system written in Python. In particular this chapter has been written to expand on the details that are usually omitted from other algorithmic trading texts and papers. The following code will allow you to simulate high-frequency (minute to second) strategies across the forecasting, momentum and mean reversion domains in the equities, foreign exchange and futures markets.

With extensive detail comes complexity, however. The backtesting system provided here requires many components, each of which are comprehensive entities in themselves. The first step is thus to outline what event-driven software is and then describe the components of the backtester and how the entire system fits together.

14.1 Event-Driven Software

Before we delve into development of such a backtester we need to understand the concept of event-driven systems. Video games provide a natural use case for event-driven software and provide a straightforward example to explore. A video game has multiple components that interact with each other in a real-time setting at high framerates. This is handled by running the entire set of calculations within an "infinite" loop known as the event-loop or game-loop.

At each tick of the game-loop a function is called to receive the latest event, which will have been generated by some corresponding prior action within the game. Depending upon the nature of the event, which could include a key-press or a mouse click, some subsequent action is taken, which will either terminate the loop or generate some additional events. The process will then continue.

Here is some example pseudo-code:

```
while True:  # Run the loop forever
    new_event = get_new_event()   # Get the latest event

    # Based on the event type, perform an action
    if new_event.type == "LEFT_MOUSE_CLICK":
        open_menu()
    elif new_event.type == "ESCAPE_KEY_PRESS":
        quit_game()
    elif new_event.type == "UP_KEY_PRESS":
        move_player_north()
    # ... and many more events

    redraw_screen()   # Update the screen to provide animation
    tick(50)   # Wait 50 milliseconds
```

The code is continually checking for new events and then performing actions based on these events. In particular it allows the illusion of real-time response handling because the code is continually being looped and events checked for. As will become clear this is precisely what we need in order to carry out high frequency trading simulation.

14.1.1 Why An Event-Driven Backtester?

Event-driven systems provide many advantages over a vectorised approach:

- **Code Reuse** - An event-driven backtester, by design, can be used for both historical backtesting and live trading with minimal switch-out of components. This is not true of vectorised backtesters where all data must be available at once to carry out statistical analysis.
- **Lookahead Bias** - With an event-driven backtester there is no lookahead bias as market data receipt is treated as an "event" that must be acted upon. Thus it is possible to "drip feed" an event-driven backtester with market data, replicating how an order management and portfolio system would behave.
- **Realism** - Event-driven backtesters allow significant customisation over how orders are executed and transaction costs are incurred. It is straightforward to handle basic market and limit orders, as well as market-on-open (MOO) and market-on-close (MOC), since a custom exchange handler can be constructed.

Although event-driven systems come with many benefits they suffer from two major disadvantages over simpler vectorised systems. Firstly they are significantly more complex to implement and test. There are more "moving parts" leading to a greater chance of introducing bugs. To mitigate this proper software testing methodology such as test-driven development can be employed.

Secondly they are slower to execute compared to a vectorised system. Optimal vectorised operations are unable to be utilised when carrying out mathematical calculations.

14.2 Component Objects

To apply an event-driven approach to a backtesting system it is necessary to define our components (or objects) that will handle specific tasks:

- **Event** - The Event is the fundamental class unit of the event-driven system. It contains a type (such as "MARKET", "SIGNAL", "ORDER" or "FILL") that determines how it will be handled within the event-loop.
- **Event Queue** - The Event Queue is an in-memory Python Queue object that stores all of the Event sub-class objects that are generated by the rest of the software.
- **DataHandler** - The DataHandler is an abstract base class (ABC) that presents an interface for handling both historical or live market data. This provides significant flexibility as the Strategy and Portfolio modules can thus be reused between both approaches. The DataHandler generates a new MarketEvent upon every heartbeat of the system (see below).
- **Strategy** - The Strategy is also an ABC that presents an interface for taking market data and generating corresponding SignalEvents, which are ultimately utilised by the Portfolio object. A SignalEvent contains a ticker symbol, a direction (LONG or SHORT) and a timestamp.
- **Portfolio** - This is a class hierarchy which handles the order management associated with current and subsequent positions for a strategy. It also carries out risk management across the portfolio, including sector exposure and position sizing. In a more sophisticated implementation this could be delegated to a RiskManagement class. The Portfolio takes SignalEvents from the Queue and generates OrderEvents that get added to the Queue.

- **ExecutionHandler** - The ExecutionHandler simulates a connection to a brokerage. The job of the handler is to take OrderEvents from the Queue and execute them, either via a simulated approach or an actual connection to a liver brokerage. Once orders are executed the handler creates FillEvents, which describe what was actually transacted, including fees, commission and slippage (if modelled).
- **Backtest** - All of these components are wrapped in an event-loop that correctly handles all Event types, routing them to the appropriate component.

Despite the quantity of components, this is quite a basic model of a trading engine. There is significant scope for expansion, particularly in regard to how the Portfolio is used. In addition differing transaction cost models might also be abstracted into their own class hierarchy.

14.2.1 Events

The first component to be discussed is the Event class hierarchy. In this infrastructure there are four types of events which allow communication between the above components via an event queue. They are a MarketEvent, SignalEvent, OrderEvent and FillEvent.

Event

The parent class in the hierarchy is called Event. It is a base class and does not provide any functionality or specific interface. Since in many implementations the Event objects will likely develop greater complexity it is thus being "future-proofed" by creating a class hierarchy.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# event.py

from __future__ import print_function


class Event(object):
    """
    Event is base class providing an interface for all subsequent
    (inherited) events, that will trigger further events in the
    trading infrastructure.
    """
    pass
```

MarketEvent

MarketEvents are triggered when the outer while loop of the backtesting system begins a new "heartbeat". It occurs when the DataHandler object receives a new update of market data for any symbols which are currently being tracked. It is used to trigger the Strategy object generating new trading signals. The event object simply contains an identification that it is a market event, with no other structure.

```
# event.py

class MarketEvent(Event):
    """
    Handles the event of receiving a new market update with
    corresponding bars.
    """

    def __init__(self):
```

```
        """
        Initialises the MarketEvent.
        """
        self.type = 'MARKET'
```

SignalEvent

The Strategy object utilises market data to create new SignalEvents. The SignalEvent contains a strategy ID, a ticker symbol, a timestamp for when it was generated, a direction (long or short) and a "strength" indicator (this is useful for mean reversion strategies). The SignalEvents are utilised by the Portfolio object as advice for how to trade.

```
# event.py

class SignalEvent(Event):
    """
    Handles the event of sending a Signal from a Strategy object.
    This is received by a Portfolio object and acted upon.
    """

    def __init__(self, strategy_id, symbol, datetime, signal_type, strength):
        """
        Initialises the SignalEvent.

        Parameters:
        strategy_id - The unique identifier for the strategy that
            generated the signal.
        symbol - The ticker symbol, e.g. 'GOOG'.
        datetime - The timestamp at which the signal was generated.
        signal_type - 'LONG' or 'SHORT'.
        strength - An adjustment factor "suggestion" used to scale
            quantity at the portfolio level. Useful for pairs strategies.
        """

        self.type = 'SIGNAL'
        self.strategy_id = strategy_id
        self.symbol = symbol
        self.datetime = datetime
        self.signal_type = signal_type
        self.strength = strength
```

OrderEvent

When a Portfolio object receives SignalEvents it assesses them in the wider context of the portfolio, in terms of risk and position sizing. This ultimately leads to OrderEvents that will be sent to an ExecutionHandler.

The OrderEvent is slightly more complex than a SignalEvent since it contains a quantity field in addition to the aforementioned properties of SignalEvent. The quantity is determined by the Portfolio constraints. In addition the OrderEvent has a print_order() method, used to output the information to the console if necessary.

```
# event.py

class OrderEvent(Event):
    """
    Handles the event of sending an Order to an execution system.
    The order contains a symbol (e.g. GOOG), a type (market or limit),
```

```
    quantity and a direction.
    """

    def __init__(self, symbol, order_type, quantity, direction):
        """
        Initialises the order type, setting whether it is
        a Market order ('MKT') or Limit order ('LMT'), has
        a quantity (integral) and its direction ('BUY' or
        'SELL').

        Parameters:
        symbol - The instrument to trade.
        order_type - 'MKT' or 'LMT' for Market or Limit.
        quantity - Non-negative integer for quantity.
        direction - 'BUY' or 'SELL' for long or short.
        """

        self.type = 'ORDER'
        self.symbol = symbol
        self.order_type = order_type
        self.quantity = quantity
        self.direction = direction

    def print_order(self):
        """
        Outputs the values within the Order.
        """
        print(
            "Order: Symbol=%s, Type=%s, Quantity=%s, Direction=%s" %
            (self.symbol, self.order_type, self.quantity, self.direction)
        )
```

FillEvent

When an ExecutionHandler receives an OrderEvent it must transact the order. Once an order has been transacted it generates a FillEvent, which describes the cost of purchase or sale as well as the transaction costs, such as fees or slippage.

The FillEvent is the Event with the greatest complexity. It contains a timestamp for when an order was filled, the symbol of the order and the exchange it was executed on, the quantity of shares transacted, the actual price of the purchase and the commission incurred.

The commission is calculated using the Interactive Brokers commissions. For US API orders this commission is 1.30 USD minimum per order, with a flat rate of either 0.013 USD or 0.08 USD per share depending upon whether the trade size is below or above 500 units of stock.

```
# event.py

class FillEvent(Event):
    """
    Encapsulates the notion of a Filled Order, as returned
    from a brokerage. Stores the quantity of an instrument
    actually filled and at what price. In addition, stores
    the commission of the trade from the brokerage.
    """

    def __init__(self, timeindex, symbol, exchange, quantity,
                 direction, fill_cost, commission=None):
```

```
        """
        Initialises the FillEvent object. Sets the symbol, exchange,
        quantity, direction, cost of fill and an optional
        commission.

        If commission is not provided, the Fill object will
        calculate it based on the trade size and Interactive
        Brokers fees.

        Parameters:
        timeindex - The bar-resolution when the order was filled.
        symbol - The instrument which was filled.
        exchange - The exchange where the order was filled.
        quantity - The filled quantity.
        direction - The direction of fill ('BUY' or 'SELL')
        fill_cost - The holdings value in dollars.
        commission - An optional commission sent from IB.
        """

        self.type = 'FILL'
        self.timeindex = timeindex
        self.symbol = symbol
        self.exchange = exchange
        self.quantity = quantity
        self.direction = direction
        self.fill_cost = fill_cost

        # Calculate commission
        if commission is None:
            self.commission = self.calculate_ib_commission()
        else:
            self.commission = commission

    def calculate_ib_commission(self):
        """
        Calculates the fees of trading based on an Interactive
        Brokers fee structure for API, in USD.

        This does not include exchange or ECN fees.

        Based on "US API Directed Orders":
        https://www.interactivebrokers.com/en/index.php?
        f=commission&p=stocks2
        """
        full_cost = 1.3
        if self.quantity <= 500:
            full_cost = max(1.3, 0.013 * self.quantity)
        else: # Greater than 500
            full_cost = max(1.3, 0.008 * self.quantity)
        return full_cost
```

14.2.2 Data Handler

One of the goals of an event-driven trading system is to minimise duplication of code between the backtesting element and the live execution element. Ideally it would be optimal to utilise the same signal generation methodology and portfolio management components for both historical

testing and live trading. In order for this to work the Strategy object which generates the Signals, and the Portfolio object which provides Orders based on them, must utilise an identical interface to a market feed for both historic and live running.

This motivates the concept of a class hierarchy based on a DataHandler object, which gives all subclasses an interface for providing market data to the remaining components within the system. In this way any subclass data handler can be "swapped out", without affecting strategy or portfolio calculation.

Specific example subclasses could include HistoricCSVDataHandler, QuandlDataHandler, SecuritiesMasterDataHandler, InteractiveBrokersMarketFeedDataHandler etc. In this chapter we are only going to consider the creation of a historic CSV data handler, which will load intraday CSV data for equities in an Open-Low-High-Close-Volume-OpenInterest set of bars. This can then be used to "drip feed" on a bar-by-bar basis the data into the Strategy and Portfolio classes on every heartbeat of the system, thus avoiding lookahead bias.

The first task is to import the necessary libraries. Specifically it will be necessary to import pandas and the abstract base class tools. Since the DataHandler generates MarketEvents, event.py is also needed as described above.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# data.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
import os, os.path

import numpy as np
import pandas as pd

from event import MarketEvent
```

The DataHandler is an abstract base class (ABC), which means that it is impossible to instantiate an instance directly. Only subclasses may be instantiated. The rationale for this is that the ABC provides an interface that all subsequent DataHandler subclasses must adhere to thereby ensuring compatibility with other classes that communicate with them.

We make use of the __metaclass__ property to let Python know that this is an ABC. In addition we use the @abstractmethod decorator to let Python know that the method will be overridden in subclasses (this is identical to a *pure virtual method* in C++).

There are six methods listed for the class. The first two methods, get_latest_bar and get_latest_bars, are used to retrieve a recent subset of the historical trading bars from a stored list of such bars. These methods come in handy within the Strategy and Portfolio classes, due to the need to constantly be aware of current market prices and volumes.

The following method, get_latest_bar_datetime, simply returns a Python datetime object that represents the timestamp of the bar (e.g. a date for daily bars or a minute-resolution object for minutely bars).

The following two methods, get_latest_bar_value and get_latest_bar_values, are convenience methods used to retrieve individual values from a particular bar, or list of bars. For instance it is often the case that a strategy is only interested in closing prices. In this instance we can use these methods to return a list of floating point values representing the closing prices of previous bars, rather than having to obtain it from the list of bar objects. This generally increases efficiency of strategies that utilise a "lookback window", such as those involving regressions.

The final method, update_bars, provides a "drip feed" mechanism for placing bar information on a new data structure that strictly prohibits lookahead bias. This is one of the key differences between an event-driven backtesting system and one based on vectorisation. Notice that exceptions will be raised if an attempted instantiation of the class occurs:

```
# data.py

class DataHandler(object):
    """
    DataHandler is an abstract base class providing an interface for
    all subsequent (inherited) data handlers (both live and historic).

    The goal of a (derived) DataHandler object is to output a generated
    set of bars (OHLCVI) for each symbol requested.

    This will replicate how a live strategy would function as current
    market data would be sent "down the pipe". Thus a historic and live
    system will be treated identically by the rest of the backtesting suite.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def get_latest_bar(self, symbol):
        """
        Returns the last bar updated.
        """
        raise NotImplementedError("Should implement get_latest_bar()")

    @abstractmethod
    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars updated.
        """
        raise NotImplementedError("Should implement get_latest_bars()")

    @abstractmethod
    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        raise NotImplementedError("Should implement
            get_latest_bar_datetime()")

    @abstractmethod
    def get_latest_bar_value(self, symbol, val_type):
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        from the last bar.
        """
        raise NotImplementedError("Should implement
            get_latest_bar_value()")

    @abstractmethod
    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        raise NotImplementedError("Should implement
            get_latest_bars_values()")
```

```
    @abstractmethod
    def update_bars(self):
        """
        Pushes the latest bars to the bars_queue for each symbol
        in a tuple OHLCVI format: (datetime, open, high, low,
        close, volume, open interest).
        """
        raise NotImplementedError("Should implement update_bars()")
```

In order to create a backtesting system based on historical data we need to consider a mechanism for importing data via common sources. We've discussed the benefits of a Securities Master Database in previous chapters. Thus a good candidate for making a DataHandler class would be to couple it with such a database.

However, for clarity in this chapter, I want to discuss a simpler mechanism, that of importing (potentially large) comma-separated variable (CSV) files. This will allow us to focus on the mechanics of creating the DataHandler, rather than be concerned with the "boilerplate" code of connecting to a database and using SQL queries to grab data.

Thus we are going to define the HistoricCSVDataHandler subclass, which is designed to process multiple CSV files, one for each traded symbol, and convert these into a dictionary of pandas DataFrames that can be accessed by the previously mentioned bar methods.

The data handler requires a few parameters, namely an Event Queue on which to push MarketEvent information to, the absolute path of the CSV files and a list of symbols. Here is the initialisation of the class:

```
# data.py

class HistoricCSVDataHandler(DataHandler):
    """
    HistoricCSVDataHandler is designed to read CSV files for
    each requested symbol from disk and provide an interface
    to obtain the "latest" bar in a manner identical to a live
    trading interface.
    """

    def __init__(self, events, csv_dir, symbol_list):
        """
        Initialises the historic data handler by requesting
        the location of the CSV files and a list of symbols.

        It will be assumed that all files are of the form
        'symbol.csv', where symbol is a string in the list.

        Parameters:
        events - The Event Queue.
        csv_dir - Absolute directory path to the CSV files.
        symbol_list - A list of symbol strings.
        """
        self.events = events
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list

        self.symbol_data = {}
        self.latest_symbol_data = {}
        self.continue_backtest = True

        self._open_convert_csv_files()
```

The handler is will look for files in the absolute directory csv_dir and try to open them with the format of "SYMBOL.csv", where SYMBOL is the ticker symbol (such as GOOG or AAPL). The format of the files matches that provided by Yahoo Finance, but is easily modified to handle additional data formats, such as those provided by Quandl or DTN IQFeed. The opening of the files is handled by the _open_convert_csv_files method below.

One of the benefits of using pandas as a datastore internally within the HistoricCSVDataHandler is that the indexes of all symbols being tracked can be merged together. This allows missing data points to be padded forward, backward or interpolated within these gaps such that tickers can be compared on a bar-to-bar basis. This is necessary for mean-reverting strategies, for instance. Notice the use of the union and reindex methods when combining the indexes for all symbols:

```
# data.py

    def _open_convert_csv_files(self):
        """
        Opens the CSV files from the data directory, converting
        them into pandas DataFrames within a symbol dictionary.

        For this handler it will be assumed that the data is
        taken from Yahoo. Thus its format will be respected.
        """
        comb_index = None
        for s in self.symbol_list:
            # Load the CSV file with no header information, indexed on date
            self.symbol_data[s] = pd.io.parsers.read_csv(
                os.path.join(self.csv_dir, '%s.csv' % s),
                header=0, index_col=0, parse_dates=True,
                names=[
                    'datetime', 'open', 'high',
                    'low', 'close', 'volume', 'adj_close'
                ]
            ).sort()

            # Combine the index to pad forward values
            if comb_index is None:
                comb_index = self.symbol_data[s].index
            else:
                comb_index.union(self.symbol_data[s].index)

            # Set the latest symbol_data to None
            self.latest_symbol_data[s] = []

        # Reindex the dataframes
        for s in self.symbol_list:
            self.symbol_data[s] = self.symbol_data[s].\
                reindex(index=comb_index, method='pad').iterrows()
```

The _get_new_bar method creates a generator to provide a new bar. This means that subsequent calls to the method will *yield* a new bar until the end of the symbol data is reached:

```
# data.py

    def _get_new_bar(self, symbol):
        """
        Returns the latest bar from the data feed.
        """
        for b in self.symbol_data[symbol]:
```

```
            yield b
```

The first abstract methods from DataHandler to be implemented are get_latest_bar and get_latest_bars. These method simply provide either a bar or list of the last N bars from the latest_symbol_data structure:

```
# data.py

    def get_latest_bar(self, symbol):
        """
        Returns the last bar from the latest_symbol list.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-1]

    def get_latest_bars(self, symbol, N=1):
        """
        Returns the last N bars from the latest_symbol list,
        or N-k if less available.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-N:]
```

The next method, get_latest_bar_datetime, queries the latest bar for a datetime object representing the "last market price":

```
    def get_latest_bar_datetime(self, symbol):
        """
        Returns a Python datetime object for the last bar.
        """
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return bars_list[-1][0]
```

The next two methods being implemented are get_latest_bar_value and get_latest_bar_values. Both methods make use of the Python getattr function, which queries an object to see if a particular attribute exists on an object. Thus we can pass a string such as "open" or "close" to getattr and obtain the value direct from the bar, thus making the method more flexible. This stops us having to write methods of the type get_latest_bar_close, for instance:

```
    def get_latest_bar_value(self, symbol, val_type):
        """
        Returns one of the Open, High, Low, Close, Volume or OI
        values from the pandas Bar series object.
        """
```

```
        try:
            bars_list = self.latest_symbol_data[symbol]
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return getattr(bars_list[-1][1], val_type)

    def get_latest_bars_values(self, symbol, val_type, N=1):
        """
        Returns the last N bar values from the
        latest_symbol list, or N-k if less available.
        """
        try:
            bars_list = self.get_latest_bars(symbol, N)
        except KeyError:
            print("That symbol is not available in the historical data set.")
            raise
        else:
            return np.array([getattr(b[1], val_type) for b in bars_list])
```

The final method, update_bars, is the second abstract method from DataHandler. It simply generates a MarketEvent that gets added to the queue as it appends the latest bars to the latest_symbol_data dictionary:

```
# data.py

    def update_bars(self):
        """
        Pushes the latest bar to the latest_symbol_data structure
        for all symbols in the symbol list.
        """
        for s in self.symbol_list:
            try:
                bar = next(self._get_new_bar(s))
            except StopIteration:
                self.continue_backtest = False
            else:
                if bar is not None:
                    self.latest_symbol_data[s].append(bar)
        self.events.put(MarketEvent())
```

Thus we have a DataHandler-derived object, which is used by the remaining components to keep track of market data. The Strategy, Portfolio and ExecutionHandler objects all require the current market data thus it makes sense to centralise it to avoid duplication of storage between these classes.

14.2.3 Strategy

A `Strategy` object encapsulates all calculation on market data that generate *advisory* signals to a `Portfolio` object. Thus all of the "strategy logic" resides within this class. I have opted to separate out the Strategy and Portfolio objects for this backtester, since I believe this is more amenable to the situation of multiple strategies feeding "ideas" to a larger Portfolio, which then can handle its own risk (such as sector allocation, leverage). In higher frequency trading, the strategy and portfolio concepts will be tightly coupled and extremely hardware dependent. This is well beyond the scope of this chapter, however!

At this stage in the event-driven backtester development there is no concept of an *indicator* or *filter*, such as those found in technical trading. These are also good candidates for creating

a class hierarchy but are beyond the scope of this chapter. Thus such mechanisms will be used directly in derived Strategy objects.

The strategy hierarchy is relatively simple as it consists of an abstract base class with a single pure virtual method for generating `SignalEvent` objects. In order to create the `Strategy` hierarchy it is necessary to import NumPy, pandas, the `Queue` object (which has become `queue` in Python 3), abstract base class tools and the `SignalEvent`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# strategy.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd

from event import SignalEvent
```

The `Strategy` abstract base class simply defines a pure virtual `calculate_signals` method. In derived classes this is used to handle the generation of `SignalEvent` objects based on market data updates:

```
# strategy.py

class Strategy(object):
    """
    Strategy is an abstract base class providing an interface for
    all subsequent (inherited) strategy handling objects.

    The goal of a (derived) Strategy object is to generate Signal
    objects for particular symbols based on the inputs of Bars
    (OHLCV) generated by a DataHandler object.

    This is designed to work both with historic and live data as
    the Strategy object is agnostic to where the data came from,
    since it obtains the bar tuples from a queue object.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def calculate_signals(self):
        """
        Provides the mechanisms to calculate the list of signals.
        """
        raise NotImplementedError("Should implement calculate_signals()")
```

14.2.4 Portfolio

This section describes a `Portfolio` object that keeps track of the positions within a portfolio and generates orders of a fixed quantity of stock based on signals. More sophisticated portfolio objects could include risk management and position sizing tools (such as the Kelly Criterion). In fact, in the following chapters we will add such tools to some of our trading strategies to see how they compare to a more "naive" portfolio approach.

The portfolio order management system is possibly the most complex component of an event-driven backtester. Its role is to keep track of all current market positions as well as the market value of the positions (known as the "holdings"). This is simply an estimate of the liquidation value of the position and is derived in part from the data handling facility of the backtester.

In addition to the positions and holdings management the portfolio must also be aware of risk factors and position sizing techniques in order to optimise orders that are sent to a brokerage or other form of market access.

Unfortunately, Portfolio and Order Management Systems (OMS) can become rather complex! Thus I've made a decision here to keep the Portfolio object relatively straightforward, so that you can understand the key ideas and how they are implemented. The nature of an object-oriented design is that it allows, in a natural way, the extension to more complex situations later on.

Continuing in the vein of the `Event` class hierarchy a `Portfolio` object must be able to handle `SignalEvent` objects, generate `OrderEvent` objects and interpret `FillEvent` objects to update positions. Thus it is no surprise that the `Portfolio` objects are often the largest component of event-driven systems, in terms of lines of code (LOC).

We create a new file `portfolio.py` and import the necessary libraries. These are the same as most of the other class implementations, with the exception that Portfolio is NOT going to be an abstract base class. Instead it will be normal base class. This means that it can be instantiated and thus is useful as a "first go" Portfolio object when testing out new strategies. Other Portfolios can be derived from it and override sections to add more complexity.

For completeness, here is the performance.py file:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# performance.py

from __future__ import print_function

import numpy as np
import pandas as pd


def create_sharpe_ratio(returns, periods=252):
    """
    Create the Sharpe ratio for the strategy, based on a
    benchmark of zero (i.e. no risk-free rate information).

    Parameters:
    returns - A pandas Series representing period percentage returns.
    periods - Daily (252), Hourly (252*6.5), Minutely(252*6.5*60) etc.
    """
    return np.sqrt(periods) * (np.mean(returns)) / np.std(returns)


def create_drawdowns(pnl):
    """
    Calculate the largest peak-to-trough drawdown of the PnL curve
    as well as the duration of the drawdown. Requires that the
    pnl_returns is a pandas Series.
```

```
    Parameters:
    pnl - A pandas Series representing period percentage returns.

    Returns:
    drawdown, duration - Highest peak-to-trough drawdown and duration.
    """

    # Calculate the cumulative returns curve
    # and set up the High Water Mark
    hwm = [0]

    # Create the drawdown and duration series
    idx = pnl.index
    drawdown = pd.Series(index = idx)
    duration = pd.Series(index = idx)

    # Loop over the index range
    for t in range(1, len(idx)):
        hwm.append(max(hwm[t-1], pnl[t]))
        drawdown[t]= (hwm[t]-pnl[t])
        duration[t]= (0 if drawdown[t] == 0 else duration[t-1]+1)
    return drawdown, drawdown.max(), duration.max()
```

Here is the import listing for the Portfolio.py file. We need to import the `floor` function from the `math` library in order to generate integer-valued order sizes. We also need the `FillEvent` and `OrderEvent` objects since the `Portfolio` handles both. Notice also that we are adding two additional functions, create_sharpe_ratio and create_drawdowns, both from the performance.py file described above.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# portfolio.py

from __future__ import print_function

import datetime
from math import floor
try:
    import Queue as queue
except ImportError:
    import queue

import numpy as np
import pandas as pd

from event import FillEvent, OrderEvent
from performance import create_sharpe_ratio, create_drawdowns
```

The initialisation of the Portfolio object requires access to the bars DataHandler, the events Event Queue, a start datetime stamp and an initial capital value (defaulting to 100,000 USD).

The Portfolio is designed to handle position sizing and current holdings, but will carry out trading orders in a "dumb" manner by simply sending them directly to the brokerage with a predetermined fixed quantity size, irrespective of cash held. These are all unrealistic assumptions, but they help to outline how a portfolio order management system (OMS) functions in an event-driven fashion.

The portfolio contains the `all_positions` and `current_positions` members. The former stores a list of all previous *positions* recorded at the timestamp of a market data event. A position is simply the quantity of the asset held. Negative positions mean the asset has been shorted. The latter current_positions dictionary stores contains the current positions for the last market bar update, for each symbol.

In addition to the positions data the portfolio stores *holdings*, which describe the current market *value* of the positions held. "Current market value" in this instance means the closing price obtained from the current market bar, which is clearly an approximation, but is reasonable enough for the time being. `all_holdings` stores the historical list of all symbol holdings, while `current_holdings` stores the most up to date dictionary of all symbol holdings values:

```
# portfolio.py

class Portfolio(object):
    """
    The Portfolio class handles the positions and market
    value of all instruments at a resolution of a "bar",
    i.e. secondly, minutely, 5-min, 30-min, 60 min or EOD.

    The positions DataFrame stores a time-index of the
    quantity of positions held.

    The holdings DataFrame stores the cash and total market
    holdings value of each symbol for a particular
    time-index, as well as the percentage change in
    portfolio total across bars.
    """

    def __init__(self, bars, events, start_date, initial_capital=100000.0):
        """
        Initialises the portfolio with bars and an event queue.
        Also includes a starting datetime index and initial capital
        (USD unless otherwise stated).

        Parameters:
        bars - The DataHandler object with current market data.
        events - The Event Queue object.
        start_date - The start date (bar) of the portfolio.
        initial_capital - The starting capital in USD.
        """
        self.bars = bars
        self.events = events
        self.symbol_list = self.bars.symbol_list
        self.start_date = start_date
        self.initial_capital = initial_capital

        self.all_positions = self.construct_all_positions()
        self.current_positions = dict( (k,v) for k, v in \
            [(s, 0) for s in self.symbol_list] )

        self.all_holdings = self.construct_all_holdings()
        self.current_holdings = self.construct_current_holdings()
```

The following method, `construct_all_positions`, simply creates a dictionary for each symbol, sets the value to zero for each and then adds a datetime key, finally adding it to a list. It uses a dictionary comprehension, which is similar in spirit to a list comprehension:

```
# portfolio.py
```

```
    def construct_all_positions(self):
        """
        Constructs the positions list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        return [d]
```

The `construct_all_holdings` method is similar to the above but adds extra keys for cash, commission and total, which respectively represent the spare cash in the account after any purchases, the cumulative commission accrued and the total account equity including cash and any open positions. Short positions are treated as negative. The starting cash and total account equity are both set to the initial capital.

In this manner there are separate "accounts" for each symbol, the "cash on hand", the "commission" paid (Interactive Broker fees) and a "total" portfolio value. Clearly this does not take into account margin requirements or shorting constraints, but is sufficient to give you a flavour of how such an OMS is created:

```
# portfolio.py

    def construct_all_holdings(self):
        """
        Constructs the holdings list using the start_date
        to determine when the time index will begin.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['datetime'] = self.start_date
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return [d]
```

The following method, `construct_current_holdings` is almost identical to the method above except that it doesn't wrap the dictionary in a list, because it is only creating a single entry:

```
# portfolio.py

    def construct_current_holdings(self):
        """
        This constructs the dictionary which will hold the instantaneous
        value of the portfolio across all symbols.
        """
        d = dict( (k,v) for k, v in [(s, 0.0) for s in self.symbol_list] )
        d['cash'] = self.initial_capital
        d['commission'] = 0.0
        d['total'] = self.initial_capital
        return d
```

On every *heartbeat*, that is every time new market data is requested from the `DataHandler` object, the portfolio must update the current market value of all the positions held. In a live trading scenario this information can be downloaded and parsed directly from the brokerage, but for a backtesting implementation it is necessary to calculate these values manually from the bars DataHandler.

Unfortunately there is no such as thing as the "current market value" due to bid/ask spreads and liquidity issues. Thus it is necessary to estimate it by multiplying the quantity of the asset held by a particular approximate "price". The approach I have taken here is to use the closing

price of the last bar received. For an intraday strategy this is relatively realistic. For a daily strategy this is less realistic as the opening price can differ substantially from the closing price.

The method `update_timeindex` handles the new holdings tracking. It firstly obtains the latest prices from the market data handler and creates a new dictionary of symbols to represent the current positions, by setting the "new" positions equal to the "current" positions.

The current positions are only modified when a `FillEvent` is obtained, which is handled later on in the portfolio code. The method then appends this set of current positions to the `all_positions` list.

The holdings are then updated in a similar manner, with the exception that the market value is recalculated by multiplying the current positions count with the closing price of the latest bar. Finally the new holdings are appended to `all_holdings`:

```
# portfolio.py

    def update_timeindex(self, event):
        """
        Adds a new record to the positions matrix for the current
        market data bar. This reflects the PREVIOUS bar, i.e. all
        current market data at this stage is known (OHLCV).

        Makes use of a MarketEvent from the events queue.
        """
        latest_datetime = self.bars.get_latest_bar_datetime(
            self.symbol_list[0]
        )

        # Update positions
        # ================
        dp = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dp['datetime'] = latest_datetime

        for s in self.symbol_list:
            dp[s] = self.current_positions[s]

        # Append the current positions
        self.all_positions.append(dp)

        # Update holdings
        # ===============
        dh = dict( (k,v) for k, v in [(s, 0) for s in self.symbol_list] )
        dh['datetime'] = latest_datetime
        dh['cash'] = self.current_holdings['cash']
        dh['commission'] = self.current_holdings['commission']
        dh['total'] = self.current_holdings['cash']

        for s in self.symbol_list:
            # Approximation to the real value
            market_value = self.current_positions[s] * \
                self.bars.get_latest_bar_value(s, "adj_close")
            dh[s] = market_value
            dh['total'] += market_value

        # Append the current holdings
        self.all_holdings.append(dh)
```

The method `update_positions_from_fill` determines whether a `FillEvent` is a Buy or a Sell and then updates the `current_positions` dictionary accordingly by adding/subtracting

the correct quantity of shares:

```
# portfolio.py

    def update_positions_from_fill(self, fill):
        """
        Takes a Fill object and updates the position matrix to
        reflect the new position.

        Parameters:
        fill - The Fill object to update the positions with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update positions list with new quantities
        self.current_positions[fill.symbol] += fill_dir*fill.quantity
```

The corresponding `update_holdings_from_fill` is similar to the above method but updates the *holdings* values instead. In order to simulate the cost of a fill, the following method does not use the cost associated from the `FillEvent`. Why is this? Simply put, in a backtesting environment the fill cost is actually unknown (the *market impact* and the *depth of book* are unknown) and thus is must be estimated.

Thus the fill cost is set to the the "current market price", which is the closing price of the last bar. The holdings for a particular symbol are then set to be equal to the fill cost multiplied by the transacted quantity. For most lower frequency trading strategies in liquid markets this is a reasonable approximation, but at high frequency these issues will need to be considered in a production backtest and live trading engine.

Once the fill cost is known the current holdings, cash and total values can all be updated. The cumulative commission is also updated:

```
# portfolio.py

    def update_holdings_from_fill(self, fill):
        """
        Takes a Fill object and updates the holdings matrix to
        reflect the holdings value.

        Parameters:
        fill - The Fill object to update the holdings with.
        """
        # Check whether the fill is a buy or sell
        fill_dir = 0
        if fill.direction == 'BUY':
            fill_dir = 1
        if fill.direction == 'SELL':
            fill_dir = -1

        # Update holdings list with new quantities
        fill_cost = self.bars.get_latest_bar_value(fill.symbol, "adj_close")
        cost = fill_dir * fill_cost * fill.quantity
        self.current_holdings[fill.symbol] += cost
        self.current_holdings['commission'] += fill.commission
        self.current_holdings['cash'] -= (cost + fill.commission)
```

```
        self.current_holdings['total'] -= (cost + fill.commission)
```

The pure virtual `update_fill` method from the `Portfolio` class is implemented here. It simply executes the two preceding methods, `update_positions_from_fill` and `update_holdings_from_fill`, upon receipt of a fill event:

```
# portfolio.py

    def update_fill(self, event):
        """
        Updates the portfolio current positions and holdings
        from a FillEvent.
        """
        if event.type == 'FILL':
            self.update_positions_from_fill(event)
            self.update_holdings_from_fill(event)
```

While the `Portfolio` object must handle `FillEvent`s, it must also take care of generating `OrderEvent`s upon the receipt of one or more `SignalEvent`s.

The `generate_naive_order` method simply takes a signal to go long or short an asset, sending an order to do so for 100 shares of such an asset. Clearly 100 is an arbitrary value, and will clearly depend upon the portfolio total equity in a production simulation.

In a realistic implementation this value will be determined by a risk management or position sizing overlay. However, this is a simplistic `Portfolio` and so it "naively" sends all orders directly from the signals, without a risk system.

The method handles longing, shorting and exiting of a position, based on the current quantity and particular symbol. Corresponding `OrderEvent` objects are then generated:

```
# portfolio.py

    def generate_naive_order(self, signal):
        """
        Simply files an Order object as a constant quantity
        sizing of the signal object, without risk management or
        position sizing considerations.

        Parameters:
        signal - The tuple containing Signal information.
        """
        order = None

        symbol = signal.symbol
        direction = signal.signal_type
        strength = signal.strength

        mkt_quantity = 100
        cur_quantity = self.current_positions[symbol]
        order_type = 'MKT'

        if direction == 'LONG' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'BUY')
        if direction == 'SHORT' and cur_quantity == 0:
            order = OrderEvent(symbol, order_type, mkt_quantity, 'SELL')

        if direction == 'EXIT' and cur_quantity > 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'SELL')
        if direction == 'EXIT' and cur_quantity < 0:
            order = OrderEvent(symbol, order_type, abs(cur_quantity), 'BUY')
```

```
        return order
```

The `update_signal` method simply calls the above method and adds the generated order to the events queue:

```
# portfolio.py

    def update_signal(self, event):
        """
        Acts on a SignalEvent to generate new orders
        based on the portfolio logic.
        """
        if event.type == 'SIGNAL':
            order_event = self.generate_naive_order(event)
            self.events.put(order_event)
```

The penultimate method in the `Portfolio` is the generation of an equity curve. This simply creates a returns stream, useful for performance calculations, and then normalises the equity curve to be percentage based. Thus the account initial size is equal to 1.0, as opposed to the absolute dollar amount:

```
# portfolio.py

    def create_equity_curve_dataframe(self):
        """
        Creates a pandas DataFrame from the all_holdings
        list of dictionaries.
        """
        curve = pd.DataFrame(self.all_holdings)
        curve.set_index('datetime', inplace=True)
        curve['returns'] = curve['total'].pct_change()
        curve['equity_curve'] = (1.0+curve['returns']).cumprod()
        self.equity_curve = curve
```

The final method in the `Portfolio` is the output of the equity curve and various performance statistics related to the strategy. The final line outputs a file, equity.csv, to the same directory as the code, which can loaded into a Matplotlib Python script (or a spreadsheet such as MS Excel or LibreOffice Calc) for subsequent analysis.

Note that the Drawdown Duration is given in terms of the absolute number of "bars" that the drawdown carried on for, as opposed to a particular timeframe.

```
    def output_summary_stats(self):
        """
        Creates a list of summary statistics for the portfolio.
        """
        total_return = self.equity_curve['equity_curve'][-1]
        returns = self.equity_curve['returns']
        pnl = self.equity_curve['equity_curve']

        sharpe_ratio = create_sharpe_ratio(returns, periods=252*60*6.5)
        drawdown, max_dd, dd_duration = create_drawdowns(pnl)
        self.equity_curve['drawdown'] = drawdown

        stats = [("Total Return", "%0.2f%%" % \
                    ((total_return - 1.0) * 100.0)),
                 ("Sharpe Ratio", "%0.2f" % sharpe_ratio),
                 ("Max Drawdown", "%0.2f%%" % (max_dd * 100.0)),
                 ("Drawdown Duration", "%d" % dd_duration)]
```

```
        self.equity_curve.to_csv('equity.csv')
        return stats
```

The `Portfolio` object is the most complex aspect of the entire event-driven backtest system. The implementation here, while intricate, is relatively elementary in its handling of positions.

14.2.5 Execution Handler

In this section we will study the execution of trade orders by creating a class hierarchy that will represent a simulated order handling mechanism and ultimately tie into a brokerage or other means of market connectivity.

The `ExecutionHandler` described here is exceedingly simple, since it fills all orders at the current market price. This is highly unrealistic, but serves as a good baseline for improvement.

As with the previous abstract base class hierarchies, we must import the necessary properties and decorators from the `abc` library. In addition we need to import the `FillEvent` and `OrderEvent`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# execution.py

from __future__ import print_function

from abc import ABCMeta, abstractmethod
import datetime
try:
    import Queue as queue
except ImportError:
    import queue

from event import FillEvent, OrderEvent
```

The `ExecutionHandler` is similar to previous abstract base classes and simply has one pure virtual method, `execute_order`:

```
# execution.py

class ExecutionHandler(object):
    """
    The ExecutionHandler abstract class handles the interaction
    between a set of order objects generated by a Portfolio and
    the ultimate set of Fill objects that actually occur in the
    market.

    The handlers can be used to subclass simulated brokerages
    or live brokerages, with identical interfaces. This allows
    strategies to be backtested in a very similar manner to the
    live trading engine.
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def execute_order(self, event):
        """
        Takes an Order event and executes it, producing
        a Fill event that gets placed onto the Events queue.
```

```
        Parameters:
        event - Contains an Event object with order information.
        """
        raise NotImplementedError("Should implement execute_order()")
```

In order to backtest strategies we need to simulate how a trade will be transacted. The simplest possible implementation is to assume all orders are filled at the current market price for all quantities. This is clearly extremely unrealistic and a big part of improving backtest realism will come from designing more sophisticated models of slippage and market impact.

Note that the `FillEvent` is given a value of `None` for the `fill_cost` (see the penultimate line in `execute_order`) as we have already taken care of the cost of fill in the `Portfolio` object described above. In a more realistic implementation we would make use of the "current" market data value to obtain a realistic fill cost.

I have simply utilised ARCA as the exchange although for backtesting purposes this is purely a string placeholder. In a live execution environment this venue dependence would be far more important:

```
# execution.py

class SimulatedExecutionHandler(ExecutionHandler):
    """
    The simulated execution handler simply converts all order
    objects into their equivalent fill objects automatically
    without latency, slippage or fill-ratio issues.

    This allows a straightforward "first go" test of any strategy,
    before implementation with a more sophisticated execution
    handler.
    """

    def __init__(self, events):
        """
        Initialises the handler, setting the event queues
        up internally.

        Parameters:
        events - The Queue of Event objects.
        """
        self.events = events

    def execute_order(self, event):
        """
        Simply converts Order objects into Fill objects naively,
        i.e. without any latency, slippage or fill ratio problems.

        Parameters:
        event - Contains an Event object with order information.
        """
        if event.type == 'ORDER':
            fill_event = FillEvent(
                datetime.datetime.utcnow(), event.symbol,
                'ARCA', event.quantity, event.direction, None
            )
            self.events.put(fill_event)
```

14.2.6 Backtest

We are now in a position to create the Backtest class hierarchy. The Backtest object encapsulates the event-handling logic and essentially ties together all of the other classes that we have discussed above.

The Backtest object is designed to carry out a nested while-loop event-driven system in order to handle the events placed on the Event Queue object. The outer while-loop is known as the "heartbeat loop" and decides the temporal resolution of the backtesting system. In a live environment this value will be a positive number, such as 600 seconds (every ten minutes). Thus the market data and positions will only be updated on this timeframe.

For the backtester described here the "heartbeat" can be set to zero, irrespective of the strategy frequency, since the data is already available by virtue of the fact it is historical!

We can run the backtest at whatever speed we like, since the event-driven system is agnostic to *when* the data became available, so long as it has an associated timestamp. Hence I've only included it to demonstrate how a live trading engine would function. The outer loop thus ends once the DataHandler lets the Backtest object know, by using a boolean continue_backtest attribute.

The inner while-loop actually processes the signals and sends them to the correct component depending upon the event type. Thus the Event Queue is continually being populated and depopulated with events. This is what it means for a system to be *event-driven*.

The first task is to import the necessary libraries. We import pprint ("pretty-print"), because we want to display the stats in an output-friendly manner:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# backtest.py

from __future__ import print_function

import datetime
import pprint
try:
    import Queue as queue
except ImportError:
    import queue
import time
```

The initialisation of the Backtest object requires the CSV directory, the full symbol list of traded symbols, the initial capital, the heartbeat time in milliseconds, the start datetime stamp of the backtest as well as the DataHandler, ExecutionHandler, Portfolio and Strategy objects. A Queue is used to hold the events. The signals, orders and fills are counted:

```
# backtest.py

class Backtest(object):
    """
    Enscapsulates the settings and components for carrying out
    an event-driven backtest.
    """

    def __init__(
        self, csv_dir, symbol_list, initial_capital,
        heartbeat, start_date, data_handler,
        execution_handler, portfolio, strategy
    ):
        """
        Initialises the backtest.
```

```
        Parameters:
        csv_dir - The hard root to the CSV data directory.
        symbol_list - The list of symbol strings.
        intial_capital - The starting capital for the portfolio.
        heartbeat - Backtest "heartbeat" in seconds
        start_date - The start datetime of the strategy.
        data_handler - (Class) Handles the market data feed.
        execution_handler - (Class) Handles the orders/fills for trades.
        portfolio - (Class) Keeps track of portfolio current
            and prior positions.
        strategy - (Class) Generates signals based on market data.
        """
        self.csv_dir = csv_dir
        self.symbol_list = symbol_list
        self.initial_capital = initial_capital
        self.heartbeat = heartbeat
        self.start_date = start_date

        self.data_handler_cls = data_handler
        self.execution_handler_cls = execution_handler
        self.portfolio_cls = portfolio
        self.strategy_cls = strategy

        self.events = queue.Queue()

        self.signals = 0
        self.orders = 0
        self.fills = 0
        self.num_strats = 1

        self._generate_trading_instances()
```

The first method, _generate_trading_instances, attaches all of the trading objects (DataHandler, Strategy, Portfolio and ExecutionHandler) to various internal members:

```
# backtest.py

    def _generate_trading_instances(self):
        """
        Generates the trading instance objects from
        their class types.
        """
        print(
            "Creating DataHandler, Strategy, Portfolio and ExecutionHandler"
        )
        self.data_handler = self.data_handler_cls(self.events, self.csv_dir,
            self.symbol_list)
        self.strategy = self.strategy_cls(self.data_handler, self.events)
        self.portfolio = self.portfolio_cls(self.data_handler, self.events,
                                            self.start_date,
                                            self.initial_capital)
        self.execution_handler = self.execution_handler_cls(self.events)
```

The _run_backtest method is where the signal handling of the Backtest engine is carried out. As described above there are two while loops, one nested within another. The outer keeps track of the heartbeat of the system, while the inner checks if there is an event in the Queue object, and acts on it by calling the appropriate method on the necessary object.

For a MarketEvent, the Strategy object is told to recalculate new signals, while the Portfolio object is told to reindex the time. If a SignalEvent object is received the Portfolio is told to handle the new signal and convert it into a set of OrderEvents, if appropriate. If an OrderEvent is received the ExecutionHandler is sent the order to be transmitted to the broker (if in a real trading setting). Finally, if a FillEvent is received, the Portfolio will update itself to be aware of the new positions:

```
# backtest.py

    def _run_backtest(self):
        """
        Executes the backtest.
        """
        i = 0
        while True:
            i += 1
            print i
            # Update the market bars
            if self.data_handler.continue_backtest == True:
                self.data_handler.update_bars()
            else:
                break

            # Handle the events
            while True:
                try:
                    event = self.events.get(False)
                except queue.Empty:
                    break
                else:
                    if event is not None:
                        if event.type == 'MARKET':
                            self.strategy.calculate_signals(event)
                            self.portfolio.update_timeindex(event)

                        elif event.type == 'SIGNAL':
                            self.signals += 1
                            self.portfolio.update_signal(event)

                        elif event.type == 'ORDER':
                            self.orders += 1
                            self.execution_handler.execute_order(event)

                        elif event.type == 'FILL':
                            self.fills += 1
                            self.portfolio.update_fill(event)

            time.sleep(self.heartbeat)
```

Once the backtest simulation is complete the performance of the strategy can be displayed to the terminal/console. The equity curve pandas DataFrame is created and the summary statistics are displayed, as well as the count of Signals, Orders and Fills:

```
# backtest.py

    def _output_performance(self):
        """
        Outputs the strategy performance from the backtest.
```

```
        """
        self.portfolio.create_equity_curve_dataframe()

        print("Creating summary stats...")
        stats = self.portfolio.output_summary_stats()

        print("Creating equity curve...")
        print(self.portfolio.equity_curve.tail(10))
        pprint.pprint(stats)

        print("Signals: %s" % self.signals)
        print("Orders: %s" % self.orders)
        print("Fills: %s" % self.fills)
```

The last method to be implemented is `simulate_trading`. It simply calls the two previously described methods, in order:

```
# backtest.py

    def simulate_trading(self):
        """
        Simulates the backtest and outputs portfolio performance.
        """
        self._run_backtest()
        self._output_performance()
```

This concludes the event-driven backtester operational objects.

14.3 Event-Driven Execution

Above we described a basic `ExecutionHandler` class that simply created a corresponding `FillEvent` instance for every `OrderEvent`. This is precisely what we need for a "first pass" backtest, but when we wish to actually hook up the system to a brokerage, we need more sophisticated handling. In this section we define the `IBExecutionHandler`, a class that allows us to talk to the popular Interactive Brokers API and thus automate our execution.

The essential idea of the `IBExecutionHandler` class is to receive `OrderEvent` instances from the events queue and then to execute them directly against the Interactive Brokers order API using the open source IbPy library. The class will also handle the "Server Response" messages sent back via the API. At this stage, the only action taken will be to create corresponding `FillEvent` instances that will then be sent back to the events queue.

The class itself could feasibly become rather complex, with execution optimisation logic as well as sophisticated error handling. However, I have opted to keep it relatively simple here so that you can see the main ideas and extend it in the direction that suits your particular trading style.

As always, the first task is to create the Python file and import the necessary libraries. The file is called `ib_execution.py` and lives in the same directory as the other event-driven files.

We import the necessary date/time handling libraries, the IbPy objects and the specific Event objects that are handled by `IBExecutionHandler`:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# ib_execution.py

from __future__ import print_function

import datetime
import time
```

```
from ib.ext.Contract import Contract
from ib.ext.Order import Order
from ib.opt import ibConnection, message

from event import FillEvent, OrderEvent
from execution import ExecutionHandler
```

We now define the `IBExecutionHandler` class. The `__init__` constructor firstly requires knowledge of the `events` queue. It also requires specification of `order_routing`, which I've defaulted to "SMART". If you have specific exchange requirements, you can specify them here. The default `currency` has also been set to US Dollars.

Within the method we create a `fill_dict` dictionary, needed later for usage in generating `FillEvent` instances. We also create a `tws_conn` connection object to store our connection information to the Interactive Brokers API. We also have to create an initial default `order_id`, which keeps track of all subsequent orders to avoid duplicates. Finally we register the message handlers (which we'll define in more detail below):

```
# ib_execution.py

class IBExecutionHandler(ExecutionHandler):
    """
    Handles order execution via the Interactive Brokers
    API, for use against accounts when trading live
    directly.
    """

    def __init__(
        self, events, order_routing="SMART", currency="USD"
    ):
        """
        Initialises the IBExecutionHandler instance.
        """
        self.events = events
        self.order_routing = order_routing
        self.currency = currency
        self.fill_dict = {}

        self.tws_conn = self.create_tws_connection()
        self.order_id = self.create_initial_order_id()
        self.register_handlers()
```

The IB API utilises a message-based event system that allows our class to respond in particular ways to certain messages, in a similar manner to the event-driven backtester itself. I've not included any real error handling (for the purposes of brevity), beyond output to the terminal, via the `_error_handler` method.

The `_reply_handler` method, on the other hand, is used to determine if a `FillEvent` instance needs to be created. The method asks if an "openOrder" message has been received and checks whether an entry in our `fill_dict` for this particular orderId has already been set. If not then one is created.

If it sees an "orderStatus" message and that particular message states than an order has been filled, then it calls `create_fill` to create a `FillEvent`. It also outputs the message to the terminal for logging/debug purposes:

```
# ib_execution.py

    def _error_handler(self, msg):
        """
```

```
        Handles the capturing of error messages
        """
        # Currently no error handling.
        print("Server Error: %s" % msg)

    def _reply_handler(self, msg):
        """
        Handles of server replies
        """
        # Handle open order orderId processing
        if msg.typeName == "openOrder" and \
            msg.orderId == self.order_id and \
            not self.fill_dict.has_key(msg.orderId):
            self.create_fill_dict_entry(msg)
        # Handle Fills
        if msg.typeName == "orderStatus" and \
            msg.status == "Filled" and \
            self.fill_dict[msg.orderId]["filled"] == False:
            self.create_fill(msg)
        print("Server Response: %s, %s\n" % (msg.typeName, msg))
```

The following method, `create_tws_connection`, creates a connection to the IB API using the IbPy `ibConnection` object. It uses a default port of 7496 and a default clientId of 10. Once the object is created, the `connect` method is called to perform the connection:

```
# ib_execution.py

    def create_tws_connection(self):
        """
        Connect to the Trader Workstation (TWS) running on the
        usual port of 7496, with a clientId of 10.
        The clientId is chosen by us and we will need
        separate IDs for both the execution connection and
        market data connection, if the latter is used elsewhere.
        """
        tws_conn = ibConnection()
        tws_conn.connect()
        return tws_conn
```

To keep track of separate orders (for the purposes of tracking fills) the following method `create_initial_order_id` is used. I've defaulted it to "1", but a more sophisticated approach would be th query IB for the latest available ID and use that. You can always reset the current API order ID via the Trader Workstation > Global Configuration > API Settings panel:

```
# ib_execution.py

    def create_initial_order_id(self):
        """
        Creates the initial order ID used for Interactive
        Brokers to keep track of submitted orders.
        """
        # There is scope for more logic here, but we
        # will use "1" as the default for now.
        return 1
```

The following method, `register_handlers`, simply registers the error and reply handler methods defined above with the TWS connection:

```
# ib_execution.py
```

```
    def register_handlers(self):
        """
        Register the error and server reply
        message handling functions.
        """
        # Assign the error handling function defined above
        # to the TWS connection
        self.tws_conn.register(self._error_handler, 'Error')

        # Assign all of the server reply messages to the
        # reply_handler function defined above
        self.tws_conn.registerAll(self._reply_handler)
```

In order to actually transact a trade it is necessary to create an IbPy `Contract` instance and then pair it with an IbPy `Order` instance, which will be sent to the IB API. The following method, `create_contract`, generates the first component of this pair. It expects a ticker symbol, a security type (e.g. stock or future), an exchange/primary exchange and a currency. It returns the `Contract` instance:

```
# ib_execution.py

    def create_contract(self, symbol, sec_type, exch, prim_exch, curr):
        """
        Create a Contract object defining what will
        be purchased, at which exchange and in which currency.

        symbol - The ticker symbol for the contract
        sec_type - The security type for the contract ('STK' is 'stock')
        exch - The exchange to carry out the contract on
        prim_exch - The primary exchange to carry out the contract on
        curr - The currency in which to purchase the contract
        """
        contract = Contract()
        contract.m_symbol = symbol
        contract.m_secType = sec_type
        contract.m_exchange = exch
        contract.m_primaryExch = prim_exch
        contract.m_currency = curr
        return contract
```

The following method, `create_order`, generates the second component of the pair, namely the `Order` instance. It expects an order type (e.g. market or limit), a quantity of the asset to trade and an "action" (buy or sell). It returns the `Order` instance:

```
# ib_execution.py

    def create_order(self, order_type, quantity, action):
        """
        Create an Order object (Market/Limit) to go long/short.

        order_type - 'MKT', 'LMT' for Market or Limit orders
        quantity - Integral number of assets to order
        action - 'BUY' or 'SELL'
        """
        order = Order()
        order.m_orderType = order_type
        order.m_totalQuantity = quantity
```

```
        order.m_action = action
        return order
```

In order to avoid duplicating `FillEvent` instances for a particular order ID, we utilise a dictionary called the `fill_dict` to store keys that match particular order IDs. When a fill has been generated the "filled" key of an entry for a particular order ID is set to `True`. If a subsequent "Server Response" message is received from IB stating that an order has been filled (and is a duplicate message) it will not lead to a new fill. The following method `create_fill_dict_entry` carries this out:

```
# ib_execution.py

    def create_fill_dict_entry(self, msg):
        """
        Creates an entry in the Fill Dictionary that lists
        orderIds and provides security information. This is
        needed for the event-driven behaviour of the IB
        server message behaviour.
        """
        self.fill_dict[msg.orderId] = {
            "symbol": msg.contract.m_symbol,
            "exchange": msg.contract.m_exchange,
            "direction": msg.order.m_action,
            "filled": False
        }
```

The following method, `create_fill`, actually creates the `FillEvent` instance and places it onto the events queue:

```
# ib_execution.py

    def create_fill(self, msg):
        """
        Handles the creation of the FillEvent that will be
        placed onto the events queue subsequent to an order
        being filled.
        """
        fd = self.fill_dict[msg.orderId]

        # Prepare the fill data
        symbol = fd["symbol"]
        exchange = fd["exchange"]
        filled = msg.filled
        direction = fd["direction"]
        fill_cost = msg.avgFillPrice

        # Create a fill event object
        fill = FillEvent(
            datetime.datetime.utcnow(), symbol,
            exchange, filled, direction, fill_cost
        )

        # Make sure that multiple messages don't create
        # additional fills.
        self.fill_dict[msg.orderId]["filled"] = True

        # Place the fill event onto the event queue
        self.events.put(fill_event)
```

Now that all of the preceeding methods having been implemented it remains to override the `execute_order` method from the `ExecutionHandler` abstract base class. This method actually carries out the order placement with the IB API.

We first check that the event being received to this method is actually an `OrderEvent` and then prepare the `Contract` and `Order` objects with their respective parameters. Once both are created the IbPy method `placeOrder` of the connection object is called with an associated `order_id`.

It is *extremely important* to call the `time.sleep(1)` method to ensure the order actually goes through to IB. Removal of this line leads to inconsistent behaviour of the API, at least on my system!

Finally, we increment the order ID to ensure we don't duplicate orders:

```
# ib_execution.py

    def execute_order(self, event):
        """
        Creates the necessary InteractiveBrokers order object
        and submits it to IB via their API.

        The results are then queried in order to generate a
        corresponding Fill object, which is placed back on
        the event queue.

        Parameters:
        event - Contains an Event object with order information.
        """
        if event.type == 'ORDER':
            # Prepare the parameters for the asset order
            asset = event.symbol
            asset_type = "STK"
            order_type = event.order_type
            quantity = event.quantity
            direction = event.direction

            # Create the Interactive Brokers contract via the
            # passed Order event
            ib_contract = self.create_contract(
                asset, asset_type, self.order_routing,
                self.order_routing, self.currency
            )

            # Create the Interactive Brokers order via the
            # passed Order event
            ib_order = self.create_order(
                order_type, quantity, direction
            )

            # Use the connection to the send the order to IB
            self.tws_conn.placeOrder(
                self.order_id, ib_contract, ib_order
            )

            # NOTE: This following line is crucial.
            # It ensures the order goes through!
            time.sleep(1)

            # Increment the order ID for this session
```

```
            self.order_id += 1
```

This class forms the basis of an Interactive Brokers execution handler and can be used in place of the simulated execution handler, which is only suitable for backtesting. Before the IB handler can be utilised, however, it is necessary to create a live market feed handler to replace the historical data feed handler of the backtester system.

In this way we are reusing as much as possible from the backtest and live systems to ensure that code "swap out" is minimised and thus behaviour across both is similar, if not identical.