MotiveWave™

Software Development Kit (SDK)

Programming Guide

Version: 1.3

Preface

This document explains how to use the MotiveWave™ Software Development Kit (SDK) to implement custom studies and strategies. The primary audience of this material is individual traders, or consultants (both with a programming background, aka developers) looking to implement (and possibly distribute) custom studies and/or strategies.

The development kit is based on the Java™ programming language. While advanced knowledge of this language is not required, it is recommended that the person implementing the study or strategy have a basic background in the language before reading this document.

Developers are free to use any development environment, including the command line tools in the Java™ Development Kit. Examples provided will be with the Eclipse IDE (Integrated Development Environment) available from: http://www.eclipse.org.

This document is intended to be a guide on how to use the SDK and is not a complete programming reference. API (Application Programming Interface) documentation is available (generated using Oracle's Javadoc tool) that explains all of the classes, interfaces and enumerations provided by the SDK.

Change History

Several enhancements have been added in version 1.1 of the SDK (these are compatible with version 2.2 and higher of MotiveWave™). These enhancements include the following:

1. **Path Color** – The color of a path can be changed dynamically (DataSeries::setPathColor(…) see API documentation).
2. **Multiple Instruments** – Studies/Strategies may incorporate data for one or more instruments. Trades may also be placed on more than one instrument.
3. **Composite Studies** – A study may be composed of multiple study plots and overlays.
4. **Access Control** – Distribution and usage of your studies/strategies can be controlled and managed using a web interface.
5. **Trading Sessions** – These may be used to constrain the trading hours for a strategy (intraday data only).
6. **Help Link** – This new attribute on the StudyHeader allows you provide a link to a webpage with more information on the study/strategy.

The following changes have been added in version 1.2 of the SDK (these are compatible with version 5.0 and higher of MotiveWave™). These enhancements include the following:

1. **Tick Data –** Support for live and historical data. See section on Tick data.

The following changes have been added in version 1.3 of the SDK (these are compatible with version 5.3 and higher of MotiveWave™). These enhancements include the following:

1. **Resize Points –** Support mouse interaction using resize points.
2. **Context Menu –** Support for custom items added to the context menu (right click on study)

Table of Contents

1 Introduction

Welcome to the MotiveWave™ Software Development Kit (SDK)! If you are reading this document then you are interested in developing a custom study and/or strategy for use within MotiveWave™.

Knowledge of the Java™ programming language is necessary for you to implement your studies/strategies. If you are unfamiliar with this language, it is recommended that you consult a book or take a basic course on Java programming.

All of the studies and strategies that are built into MotiveWave™ were programmed using the SDK. The source code for these are freely available and may be used as examples or starting points.

Before you begin, it is important to understand studies and strategies and the difference between them.

1.1 What is a Study?

A *study* uses historical price and/or volume data to display new information to the user to assist them in making buying or selling decisions. There are two types of studies:

1. Overlays
2. Study Plots

It is also possible to create studies that contain multiple plots and overlays.

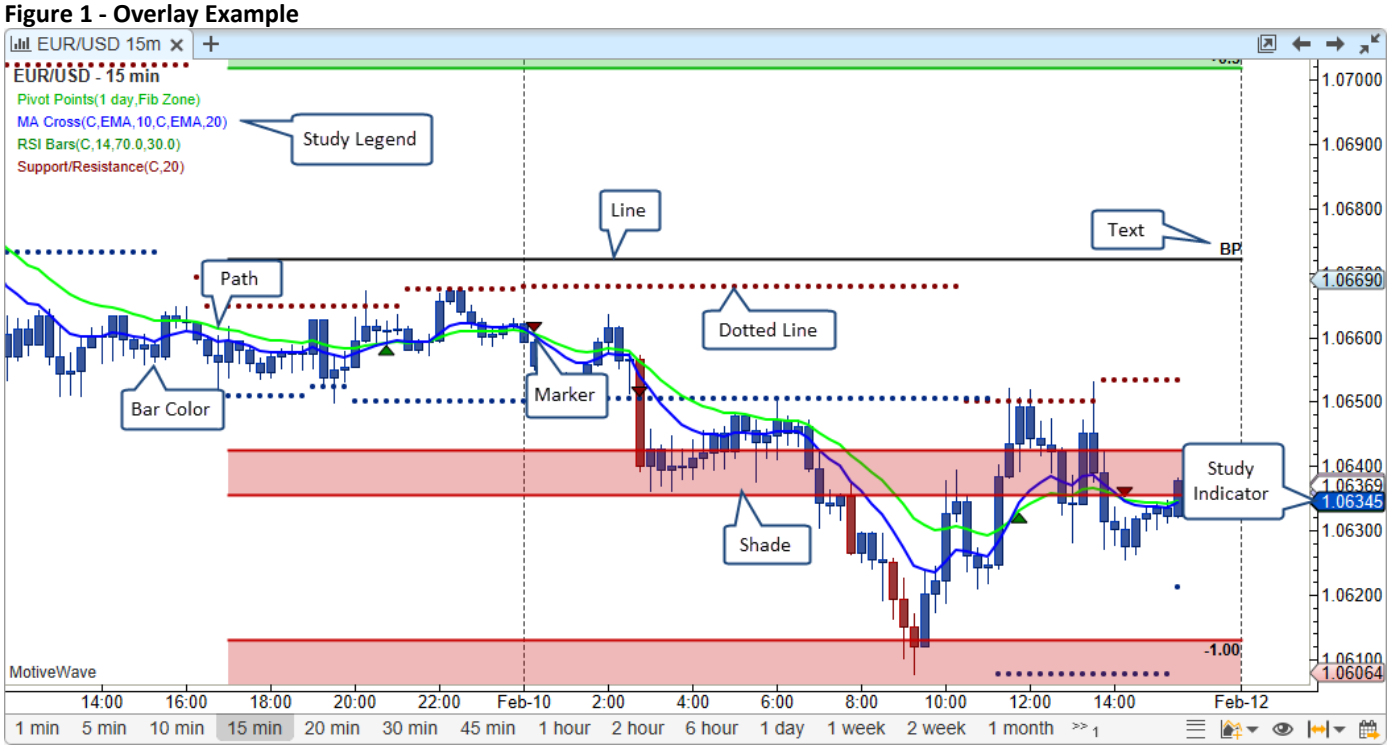
1.1.1 Overlays

Overlays display information that is drawn on top of an existing plot (most typically the price plot). What is actually displayed depends on the study itself. Some examples of what a study may display include:

- **Paths** – A path is a series of lines that connects data points. Examples of this include a moving average or price bands.
- **Markers** – Markers may be used to indicate points of interest (such as buy, sell or stop loss locations). Markers come in many forms: arrows, circles, triangles, letters, numbers etc
- **Shades** – Area of a plot may be shaded to indicate zones of interest
- **Lines** – May include trend lines, support or resistance areas
- **Paint Bars** – Price or volume bars may be displayed using specific colors
- **Text** – Descriptive text may be used to explain elements of the study
- **Figures** – any type of figure or drawing may be drawn on a plot as part of the overlay.
- **Indicators** – Indicators may be added to the vertical axis to show the current value of a study.

The following screen shot illustrates an example of some of the elements that may be part of an overlay:

Figure 1 - Overlay Example



1.1.2 Study Plots

Study plots display information drawn in a plot that is separate from the price plot. The typical reason why this is displayed in a separate plot is because the values generated are independent (or outside) of the price range.

Overlays may be added to a study plot to display additional information (such as a moving average).

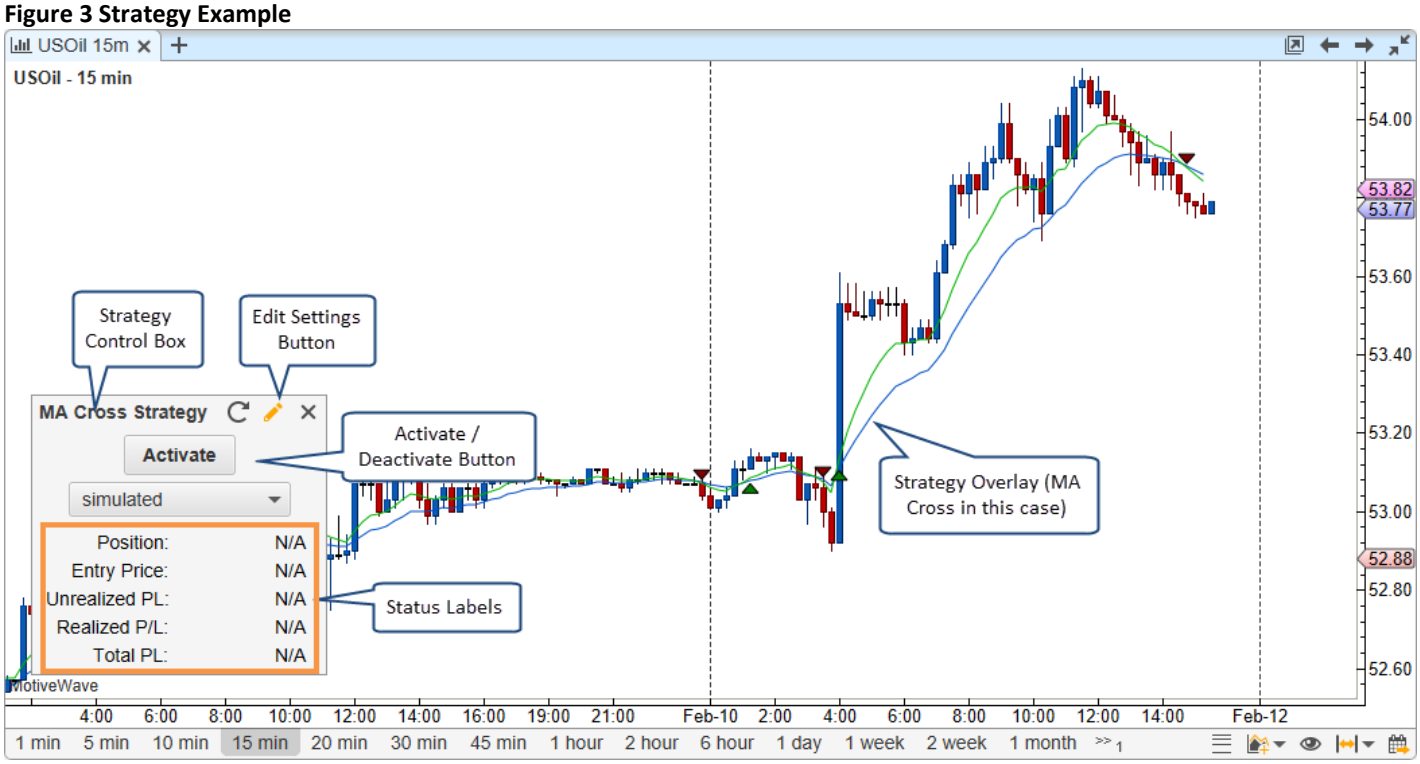
The following screen shot shows some examples of study plots:

Figure 2 - Study Plot Example

1.2 What is a Strategy?

A *strategy* is a special type of study that may be used to automate or partially automate trading. In addition to displaying the study information, a **Control Box** is made available that allows the user to activate/deactivate a strategy and view important runtime information. The following screen shot shows an example of the Moving Average Cross Strategy:

Figure 3 Strategy Example



MotiveWave™ supports two modes for strategies:

1. **Automatic** – Once the user activates the strategy, it will automatically buy and sell based on the internal logic.
2. **Manual** – In this mode, the user tells the strategy when it is OK to enter.

1.3 Distribution

Studies (and strategies) may be distributed to users by packaging them together in Jar (Java™ Archive) files. If you feel the need to protect the contents of these packages you may use obfuscators (such as ProGuard) to prevent reverse engineering of the binary code.

1.3.1 Access Control

You can control the access to a set of studies/strategies by using the 'secured' attribute in the Study Header. Setting this attribute to 'true' will ensure that only users that you have given access will be allowed to load and execute studies and strategies in the given namespace.

Access control requires an account to be setup with MotiveWave™. If you would like to utilize this feature, send an email requesting that an account be created to: support@motivewave.com.

2 Fundamental Classes

This section describes the fundamental classes that you will need to interact with when building your custom study/strategy. For a complete view of all of the classes/interfaces in the SDK, please consult the API documentation.

2.1 Packages

The SDK consists of the following 6 packages:

1. **com.motivewave.platform.sdk.common** – Contains common classes and interfaces. These include 'info' classes, enumerations, utility functions and 'context' classes that expose functionally and data from MotiveWave™
2. **com.motivewave.platform.sdk.common.desc** – Contains 'Descriptor' classes. These are used to describe settings and values to the MotiveWave™ runtime environment.
3. **com.motivewave.platform.sdk.common.menu** – Contains classes for implementing custom context menus.
4. **com.motivewave.platform.sdk.draw** – The classes in this package are used to draw figures on the price and study plots.
5. **com.motivewave.platform.sdk.study** – Contains the base classes for creating and interacting with studies and strategies.
6. **com.motivewave.platform.sdk.order_mgmt** – Contains classes/interfaces for managing orders. These are used in conjunction with strategies.

2.2 Study Class

The *Study* class is the base class for all studies and strategies. When implementing any study/strategy you will first start by deriving directly or indirectly from this class.

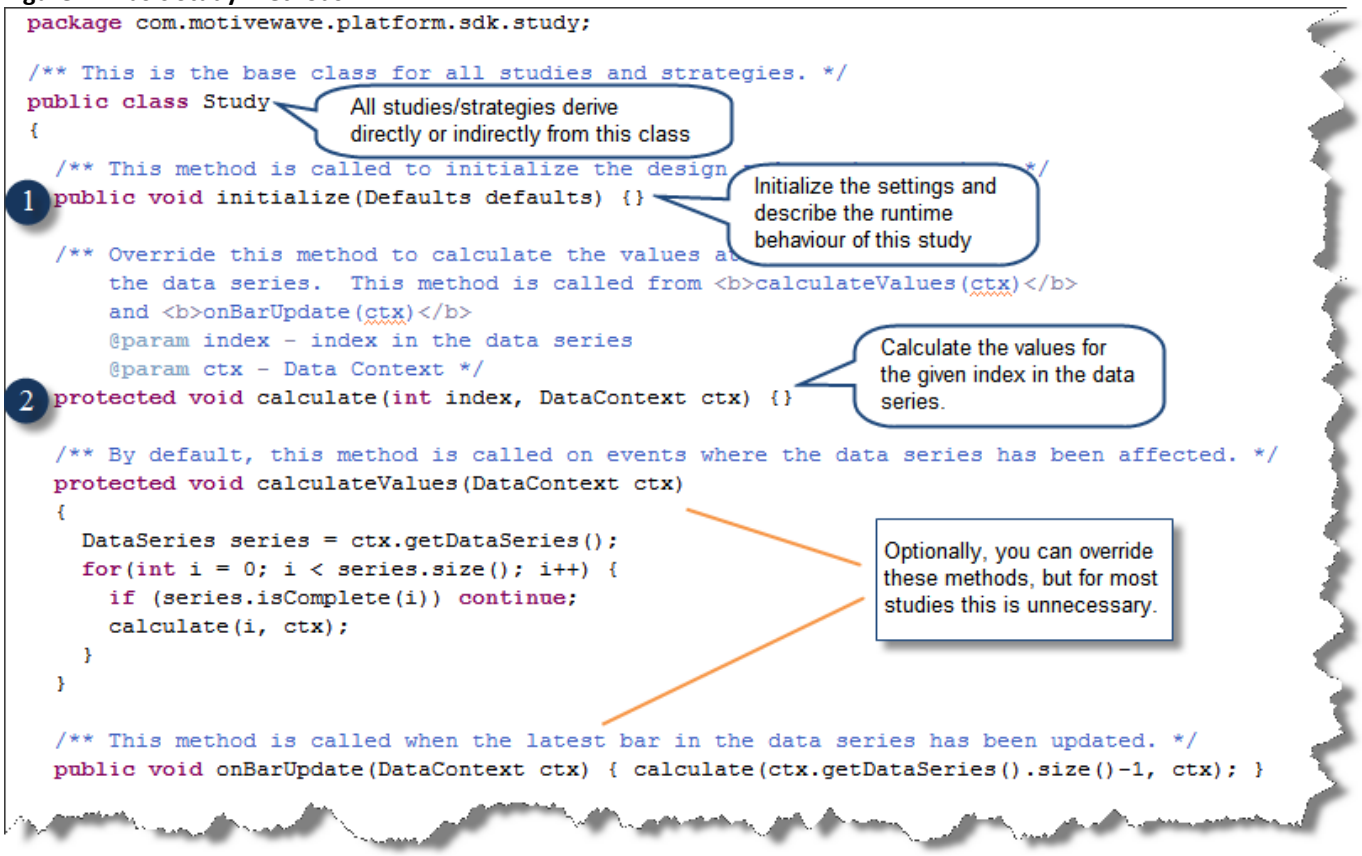
Why is there no Strategy Class?
Strategies are a specialized version of a study, in fact most strategies are based (at least in part) on an existing study. If there was a separate *Strategy* class it would be difficult (if not impossible) to implement a strategy by deriving from an existing study. It is for this reason that the methods and properties that are specific to strategies are included in the *Study* class.

For most studies there are two methods that you will override:

1. ***initialize*** – The purpose of this method is to describe the user configurable settings for the study and describe the runtime behavior.
2. ***calculate*** – This method calculates the values for the study at the given historical bar.

The following diagram illustrates the basic elements that you need to be concerned with in the Study class. For a complete list of methods and properties, see the API documentation.

Figure 4 - Basic Study Methods

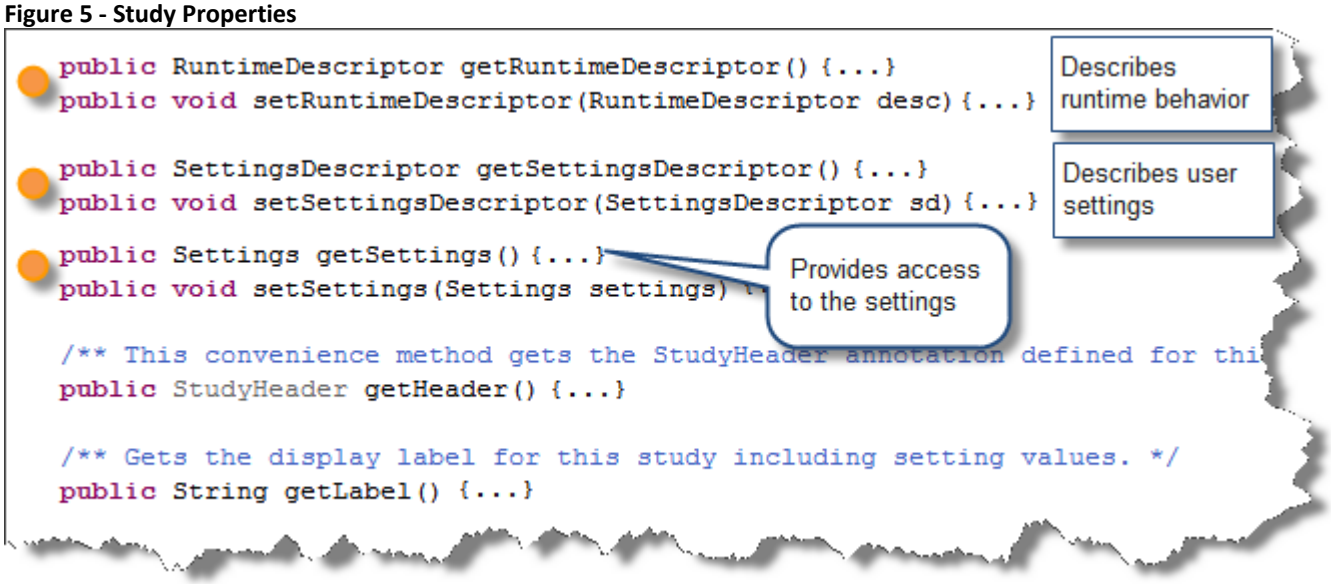



```
package com.motivewave.platform.sdk.study;

/** This is the base class for all studies and strategies. */
public class Study
{
  /** This method is called to initialize the design                    */
1 public void initialize(Defaults defaults) {}

  /** Override this method to calculate the values at
      the data series.  This method is called from <b>calculateValues(ctx)</b>
      and <b>onBarUpdate(ctx)</b>
      @param index - index in the data series
      @param ctx - Data Context */
2 protected void calculate(int index, DataContext ctx) {}

  /** By default, this method is called on events where the data series has been affected. */
  protected void calculateValues(DataContext ctx)
  {
    DataSeries series = ctx.getDataSeries();
    for(int i = 0; i < series.size(); i++) {
      if (series.isComplete(i)) continue;
      calculate(i, ctx);
    }
  }

  /** This method is called when the latest bar in the data series has been updated. */
  public void onBarUpdate(DataContext ctx) { calculate(ctx.getDataSeries().size()-1, ctx); }
```

There are 3 main properties in the Study class that are important for implementing a study:

1. **Runtime Descriptor** – this describes the runtime behavior of the study
2. **Settings Descriptor** – This describes the user settings
3. ***getSettings()*** – This is typically used in the *calculate* method to get access to the settings that the user has chosen.

Figure 5 - Study Properties




```
  public RuntimeDescriptor getRuntimeDescriptor() {...}
  public void setRuntimeDescriptor(RuntimeDescriptor desc) {...}

  public SettingsDescriptor getSettingsDescriptor() {...}
  public void setSettingsDescriptor(SettingsDescriptor sd) {...}

  public Settings getSettings() {...}
  public void setSettings(Settings settings) {...}

  /** This convenience method gets the StudyHeader annotation defined for thi
  public StudyHeader getHeader() {...}

  /** Gets the display label for this study including setting values. */
  public String getLabel() {...}
```

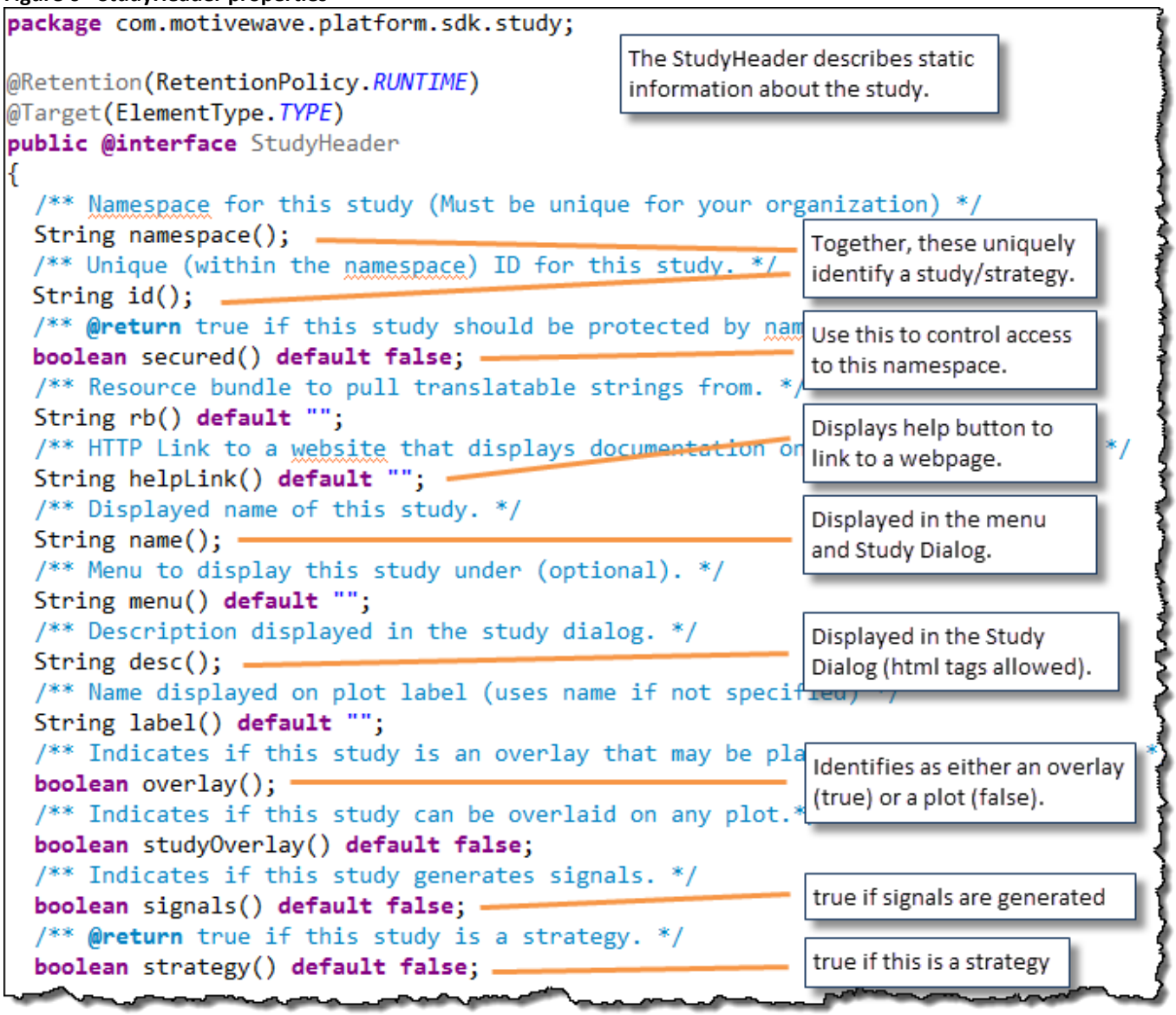
2.3 StudyHeader

The *StudyHeader* is an annotation that is required on every class derived from the *Study* class. The purpose of this annotation is to describe static information about the study/strategy.

The *StudyHeader* is read when the Study class is first loaded and is used to register the study with MotiveWave™ and make it available in the Study menu and the 'Add Study' dialog.

The following screen shot shows some of the important properties of the *StudyHeader*. For a full description of all properties see the API documentation.

Figure 6 - StudyHeader properties

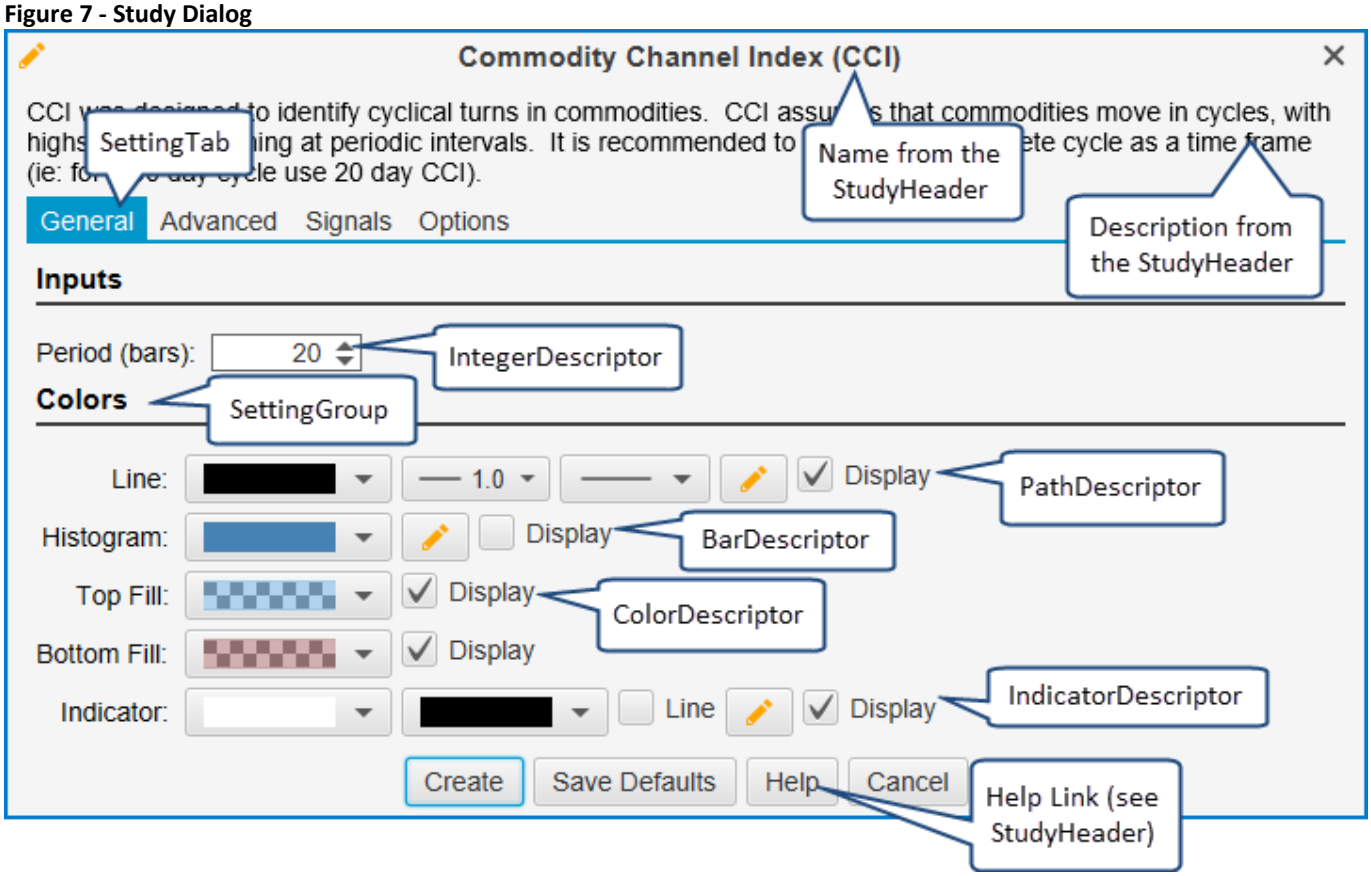


2.4 Describing User Settings

The MotiveWave™ SDK provides a lot of flexibility when describing user settings for a study. Settings may be organized into tabs and groups which are displayed in the study dialog. MotiveWave™ also provides many different setting descriptors to represent different types of settings.

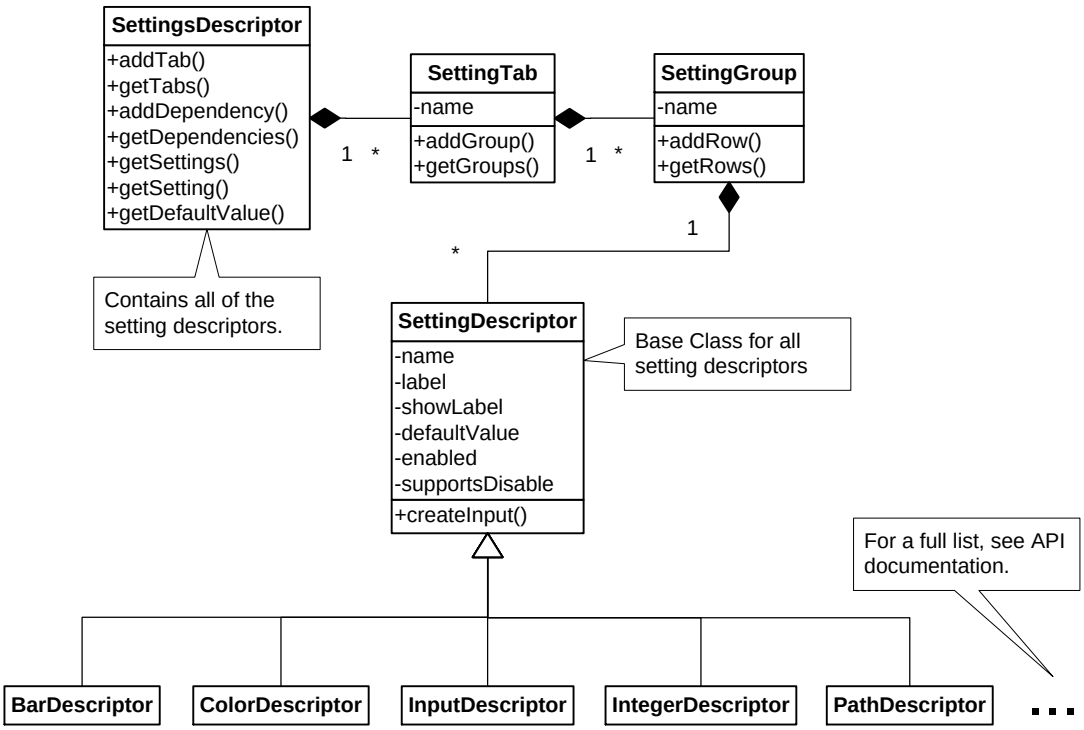
The following screen shot illustrates the study dialog for a CCI study:

Figure 7 - Study Dialog



The classes for describing user settings can be found in the package: **com.motivewave.platform.sdk.common.desc**. The following UML (Universal Markup Language) diagram illustrates the high level classes involved and how they relate to each other. For a full list of the available *SettingDescriptor* classes, see the API documentation.

Figure 8 - Descriptor Classes



2.4.1 SettingsDescriptor class

The *SettingsDescriptor* class contains all of the user configurable settings. An instance of this class should be created in the '*initialize'* method (of the Study class) and assigned to the study using the '*setSettingsDescriptor*' method.

There are two methods in this class that are important:

1. *addTab* – Adds a SettingTab object that contains settings on a tab in the Study Dialog
2. *addDependency* – Used to identify dependencies between settings. For example, an 'EnabledDependency' will enable a setting if a *BooleanSetting* is true or false.

Figure 9 - SettingsDescriptor

2.4.2 SettingTab Class

The *SettingTab* class represents a tab in the study dialog. This simple class consists of a name (to display in the tab) and a set of *SettingGroup* instances.

Figure 10 - SettingTab class

```
package com.motivewave.platform.sdk.common.desc;

/** Identifies a set of groups that may be organized in a tab. */
public class SettingTab
{
  /** Creates a tab with the given name */
  public SettingTab(String name){...}

  /** @return the human readable name of this tab. */
  public String getName() {...}

  /** Adds a group of settings. */
  public void addGroup(SettingGroup grp){...}

  /** @return the list of setting groups in this tab. */
  public List<SettingGroup> getGroups() {...}
```

2.4.3 SettingGroup Class

The *SettingGroup* class organizes related settings into a named group. The group consists of a set of rows that each contains 1 or more setting descriptors.

Figure 11 - SettingGroup class

```
package com.motivewave.platform.sdk.common.desc;

/** Identifies a set of inputs that are to be arranged as a group when
    displaying the configuration dialog to the end user.
    The input group consists of a series of row.  By default each
    input element is placed on a separate row (in the order in which
    they are given).  To place more than one element on the same row,
    pass multiple setting descriptors to the addRow method. */
public class SettingGroup
{
  /** Creates a SettingGroup with the given name. */
  public SettingGroup(String name) { iName = name; }

  /** @return the name of the group (displayed in the Study Dialog. */
  public String getName() { return iName; }

  /** Adds a row with 1 or more inputs. */
  public void addRow(SettingDescriptor... row) { iRows.add(row); }

  /** Gets the rows in this group (each row may contain 1 or more inputs) */
  public List<SettingDescriptor[]> getRows() { return iRows; }
```

2.5 Settings class

The *Settings* class contains all of the information about the settings configured by the user of the study. You can access this class by using the *getSettings()* method in the *Study* base class.

Many of the setting descriptor classes have corresponding 'Info' classes (see **com.motivewave.platform.sdk.common** package) that contain the user specific settings. These may be accessed using a series of 'get' methods on the *Settings* class. The following screen shot illustrates some of these methods. For a complete description of the *Settings* class and the Info classes see the API documentation.

Figure 12 - Settings class

```
package com.motivewave.platform.sdk.common;

/** Encapsulates the configuration information for a study or strategy. */
public class Settings implements Cloneable
{
 /** Gets the SettingsDescriptor object that describes the user settings.
  public SettingsDescriptor getDescriptor()

  /** Gets the double value associated to the given name. */
  public Double getDouble(String name)
  /** Gets the double value associated to the given name. */
  public Integer getInteger(String name)
  /** Gets the boolean value associated to the given name. */
  public Boolean getBoolean(String name)

 /** @return the PathInfo associated to the given name. */
  public PathInfo getPath(String name)
 /** @return a set of all the registered path names. */
  public Set<String> getPaths()

 /** @return the MarkerInfo associated to the given name. */
  public MarkerInfo getMarker(String name)
 /** @return a set of all the registered markers. */
  public Set<String> getMarkers()

 /** @return the IndicatorInfo associated to the given name. */
  public IndicatorInfo getIndicator(String name)
  /** @return a set of all the registered indicator names. */
  public Set<String> getIndicators()

  /** @return the BarInfo associated to the given name. */
  public BarInfo getBars(String name)
  /** @return a set of all the registered bar names. */
  public Set<String> getBars()

  /** @return the ShadeInfo associated to the given name. */
  public ShadeInfo getShade(String name)
  /** @return a set of all the registered shade names. */
  public Set<String> getShades()

  /** @return the GuideInfo associated to the given name. */
  public GuideInfo getGuide(String name)
  /** @return a set of all the registered guides. */
  public Set<String> getGuides()

  /** @return the input key associated to the given name. */
  public Object getInput(String name)
  /** @return a set of all the registered input names. */
  public Set<String> getInputs()

  /** @return the Color associated to the given name. */
  public Color getColor(String name)
```

'Info' classes. See common package.

2.6 Runtime Settings

The *RuntimeDescriptor* (**com.motivewave.platform.sdk.study** package) is used to describe runtime behavior for the study. This includes the following:

1. Label Settings – used to describe how the label is generated
2. Export Values – These are values generated by the study that may be used outside of the study.
3. Declare Elements – These methods associate values generated by the study to visual constructs on the 'default' plot (see Composite Studies below for more information):
    a. Paths – A series of values connected by lines
    b. Bars – Vertical bars displayed on a plot
    c. Signals – Signals generated by the study
    d. Indicators – Indicators displayed on the vertical axis
4. Study Plot Settings (default plot)
    a. Top/Bottom Insets – Used to add space to the top or bottom of the plot
    b. Vertical Range – Range of the vertical axis
    c. Min Tick – precision of the vertical axis values
    d. Horizontal Lines – Horizontal lines displayed on the study plot

Why do I need to declare elements such as a Path?
You may ask yourself, 'why doesn't the *PathDescriptor* (or other descriptor classes) class include the value key?'. While this may make sense in most situations, it does not allow you to use the same path information for multiple paths. Consider for example a case where you have a price bands study and you want to have the same settings for the top and bottom bands. By declaring the path for the top and bottom values as the same path info, you are able to re-use this descriptor object.

Figure 13 - RuntimeDescriptor class

```
package com.motivewave.platform.sdk.study;

/** This class describes 'runtime' settings for the study. */
public class RuntimeDescriptor
{
 /** Use this method to identify which settings should be part of
     the graph label (and to identify the study). */
  public void setLabelSettings(String... vals)

 /** Use this method to identify the numeric values generated by
     this study that are to affect the vertical range of the graph
     (when auto scale is turned on). */
  public void setRangeKeys(Object... keys)

 /** Exports a value so that it may be used outside of the context */
  public void exportValue(ValueDescriptor desc)

 /** Declare a path associated with the given value key.  Settings for the
     path are resolved using the pathSettingsKey.  At runtime a path will
     be drawn (if enabled) using the values defined by the valueKey. */
  public void declarePath(Object valueKey, String pathSettingsKey)

 /** Associates a value key to an indicator. */
  public void declareIndicator(Object valueKey, String indicatorKey)

 /** Declare a bar sequence associated with the given value key.  Settings
     for the bars are resolved using the pathSettingsKey.  At runtime a set
     of bars will be drawn (if enabled) using the values defined by the valueKey.
  public void declareBars(Object valueKey, String settingsKey)

 /** Declare a signal with the given key and user readable string. */
  public void declareSignal(Object key, String label)

 /** Adds a horizontal line to the graph using the information defined in LineInf
  public void addHorizontalLine(LineInfo info)

 /** Sets the top inset (in pixels). */
  public void setTopInsetPixels(int pixels)
 /** Sets the bottom inset (in pixels). */
  public void setBottomInsetPixels(int pixels)

  /** Sets the minimum tick for the vertical axis (if this is not an overlay).
  Set to null (default) to automatically detect the min value. */
  public void setMinTick(Double d)
```

2.6.1 Composite Studies

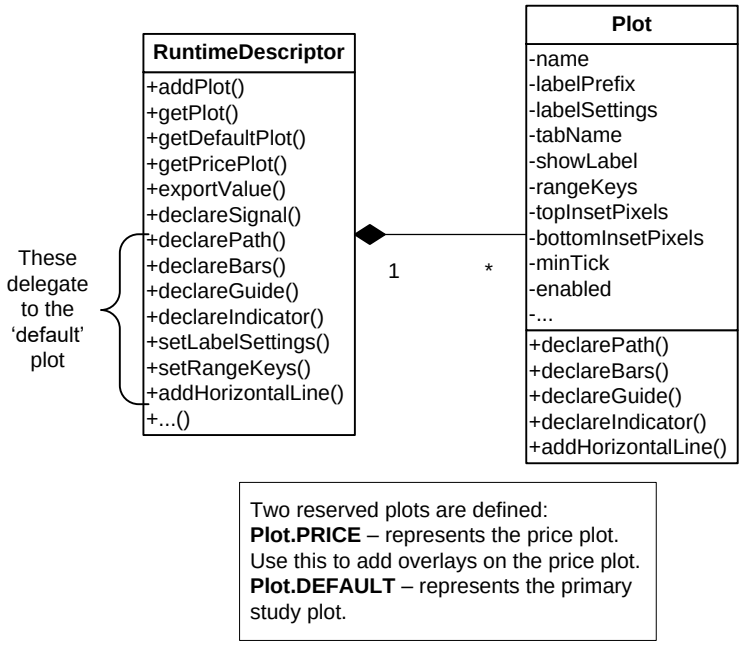
The majority of studies consist of either a single overlay or a single plot. Version 1.1 of the SDK allows you to create studies that consist of multiple study plots and (optionally) overlays on the price plot.

The *RuntimeDescriptor* class enables you to define additional plots for a study. This class has been enhanced in version 1.1 to allow the definition of additional plots using the new *Plot* class (see **com.motivewave.platform.sdk.study** package).

The majority of methods on the *RuntimeDescriptor* class operate on the 'default' plot for the study. In the case of an overlay, the default plot will be the plot where the overlay was added. For example, when you add a simple moving average (SMA) to the price plot, the default plot for the overlay will be the price plot.

Additional plots may be defined using the *Plot* class. Each plot has independent settings for labels, tabs, range keys etc and elements are declared separately for each plot (ie paths, bars etc). The following diagram illustrates the relationship between the *RuntimeDescriptor* and the *Plot* classes.

Figure 14 Runtime Descriptor and Plot classes



2.7 DataContext Interface

The *DataContext* interface provides access to historical data as well as utility methods for interacting with the study framework.

The following diagram illustrates some of the useful methods:

Figure 15 - Data Context Interface

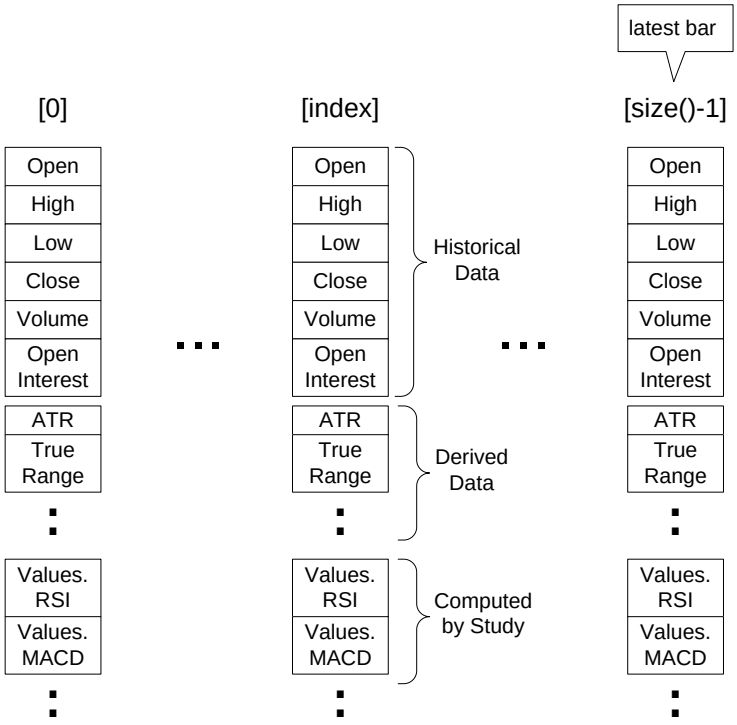
```
package com.motivewave.platform.sdk.common;

/** This context provides an access point to services relating to data. */
public interface DataContext
{
 /** Gets the primary data series. */
  DataSeries getDataSeries();

 /** Gets additional data series objects of a different bar size. */
  DataSeries getDataSeries(BarSize barSize);

 /** Gets the instrument associated with this context. */
  Instrument getInstrument();

 /** Triggers a signal with the given key, message and value.
     Note: An actual signal is only triggered if signals have been configured.
    @param index index of the bar that triggered this signal.  Note: signals
           are only fired for the current bar when it is completed
    @param signalKey event name of the alert (displayed to user)
    @param message describes the signal (dislayed to the user, if an alert)
    @param value value that triggered the alert (displayed to user)  */
  void signal(int index, Object signalKey, String message, Object value);

 /** @return true if this is regular trading hours (rth). */
  boolean isRTH();

 /** Gets the current time.  This is the time synchronized with
  the Broker/Data Service (if supported by the underlying service). */
  long getCurrentTime();
```

2.8 DataSeries Interface

The primary objective of the *DataSeries* interface is to provide a repository for historical price data and data generated by the study. Data stored in this interface is accessed by a numerical index which represents the price bar where the data applies.

The following diagram illustrates the structure of the data in the data series. Essentially the data is an array of tables where the index '0' is the first (oldest) bar and index 'size()-1' is the latest bar.

Figure 16 - Data Structure



The *DataSeries* interface also contains a number of convenience methods for calculating common values such as moving averages, swing points and lowest or highest values.

Figure 17 - DataSeries Interface

```
package com.motivewave.platform.sdk.common;

/**
 * Represents a series of price bars that are displayed on a chart.
 * Values of the price bars are accessed by specifying the index when retrieving a value
 * Study values are stored in this structure as they are computed by a study.
 * This interface also provides many convenience method for calculating moving averages,
 * swing points, highest high, etc.
 */
public interface DataSeries
{
  /** Gets the number of elements in this data series.
      @return the number of elements in this data series. */
  int size();
  /** Gets the size of the bars in this data series.  @return the bar size of this data
  BarSize getBarSize();
  /** Gets the type of data available in this data series. @return the type of bar data
  Enums.BarData getBarData();
  /** Gets the instrument for the data in this data series. @return the instrument for th
  Instrument getInstrument();
  /** @return the high value of the price bar at the given index. */
  float getHigh(int index);
  /** @return the low value of the price bar at the given index. */
  float getLow(int index);
  /** @return the open value of the price bar at the given index. */
  float getOpen(int index);
  /** @return the close value of the price bar at the given index. */
  float getClose(int index);
  /** @return the volume of the price bar at the given index. */
  long getVolume(int index);
  /** @return the start time (in millis) of the bar at the given index. */
  long getStartTime(int index);
  /** Calculates a Moving Average.  Null values and values of Double.NaN are ignored in
  Double ma(Enums.MAMethod method, int index, int period, Object key);
  /** Calculates the average true range based on the most recent complete bars. */
  Double atr(int period);
  /** Returns the highest value over the given sequence of values. Null values and value
  Double highest(int index, int period, Object key);
  /** Returns the lowest value over the given sequence of values. Null values and values
  Double lowest(int index, int period, Object key);
  /** Calculates and returns a list of swing points of a given strength or greater. */
  List<SwingPoint> calcSwingPoints(boolean top, int strength);
```

2.9 Multiple Instruments

Version 1.1 of the SDK offers support for multiple instruments. This allows you to retrieve real time and historical data for one or more instruments (beyond the primary instrument) for studies and strategies. For strategies you may also place orders for multiple instruments (see section on strategies).

Please Note: Not all editions of MotiveWave™ include support for multiple instruments. In these cases, studies requiring multiple instruments will not be accessible to the end user.

2.9.1 Design Time

Usage of multiple instruments requires the declaration of this feature in the *StudyHeader* and usage of the *InstrumentDescriptor* to declare the instruments that will be used at run time.

There are essentially two items that are necessary to enable multiple instruments as part of the design time:

1. **Declare support for multiple instruments** – In the *StudyHeader* set the attribute *multipleInstrument=true*
2. **Declare one or more instruments in the *initialize()* method** – Use the InstrumentDescriptor to declare one or more instruments. For details on how to use this class, see the API documentation.

The following code snippet illustrates the usage of the '*multipleInstrument*' attribute in the built-in *Spread* study:

Figure 18 Multiple Instrument StudyHeader

```
/** Instrument Spread */
@StudyHeader(
    namespace="com.motivewave",
    id="SPREAD",
    rb="com.motivewave.platform.study.nls.strings",
    name="TITLE_SPREAD",
    desc="DESC_SPREAD",
    menu="MENU_INSTRUMENT",
    overlay=false,
    multipleInstrument=true,
    requiresBarUpdates=true)
public class Spread extends com.motivewave.platform.sdk.study.Study
{
```

This attribute must be set to true

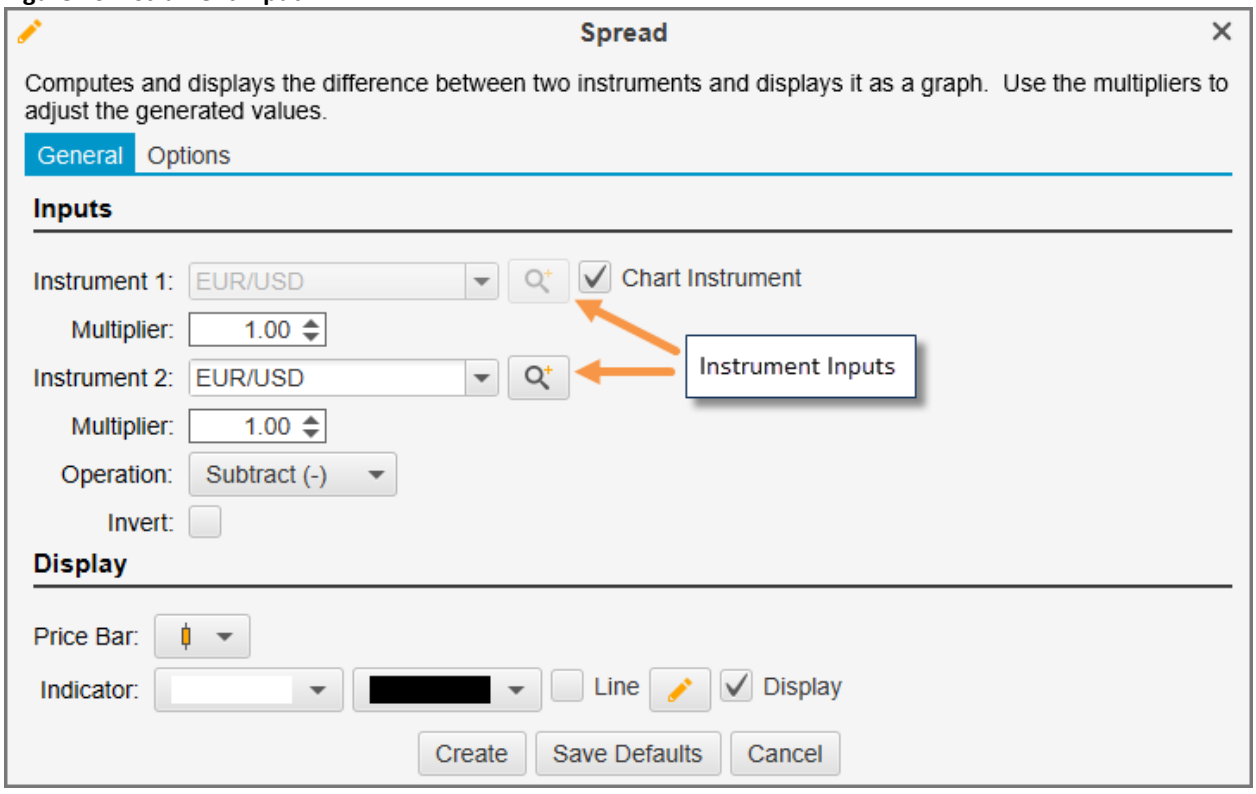
Figure 19 InstrumentDescriptor

```
public class Spread extends com.motivewave.platform.sdk.study.Study
{
  enum Values { SPREAD };
  final static String MULTIPLIER1 = "multiplier1";
  final static String MULTIPLIER2 = "multiplier2";

  @Override
  public void initialize(Defaults defaults)
  {
    SettingsDescriptor sd = new SettingsDescriptor();
    setSettingsDescriptor(sd);
    SettingTab tab = new Se                                    "));
    sd.addTab(tab);

    SettingGroup inputs = n                                    INPUTS"));
    inputs.addRow(new InputDescri or(inputs.INPUT, get("LBL_INPUT"), Enums.BarInput.valu
    inputs.addRow(new InstrumentDescriptor(Inputs.INSTRUMENT1, get("LBL_INSTRUMENT1")));
    inputs.addRow(new DoubleDescriptor(MULTIPLIER1, get("LBL_MULTIPLIER"), 1.0, 0.01, 100
    inputs.addRow(new InstrumentDescriptor(Inputs.INSTRUMENT2, get("LBL_INSTRUMENT2")));
    inputs.addRow(new DoubleDescriptor(MULTIPLIER2, get("LBL_MULTIPLIER"), 1.0, 0.01, 100
    tab.addGroup(inputs);
```

Use the *InstrumentDescriptor* to declare and allow users to choose instruments.

The following screen shot demonstrates how the InstrumentDescriptor enables the user to choose the instrument when they create the study

Figure 20 Instrument Input



2.9.2 Run Time

Several enhancements have been added to the SDK to enable access settings and historical/real time information in the run time portion of the study:

1. ***Settings*** – a new method *getInstrument(key)* on the Settings class allows you to retrieve the instrument that the user chose when they created (or modified) the study.
2. ***DataSeries*** – several new methods have been added to the *DataSeries* interface for retrieving information. Essentially, these are overloaded methods of getDouble(…), getHigh(…), getLow(…) getClose(…) etc.

The following code snippet from the Spread study shows how to retrieve chosen instruments and historical data from the DataSeries interface:

Figure 21 Spread *calculate* method

```
public class Spread extends com.motivewave.platform.sdk.study.Study
{

  @Override
  protected void calculate(int index, DataContext ctx)
  {
    Enums.BarInput input = (Enums.BarInput)getSettings().getInput(Inputs.INPUT);
    Instrument instr1 = getSettings().getInstrument(Inputs.INSTRUMENT1);
    Instrument instr2 = getSettings().getInstrument(Inputs.INSTRUMENT2);
    double mult1 = getSettings().getDouble(MULTIPLIER1, 1.0);
    double mult2 = getSettings().getDouble(MULTIPLIER2, 1.0);
    DataSeries series = ctx.getDataSeries();

    Double value1 = series.getDouble(index, input, instr1);
    if (value1 == null) {
      return;
    }
    Double value2 = series.getDouble(index, input, instr2);
    if (value2 == null) {
      return;
    }

    double spread = value1*mult1 - value2*mult2;

    series.setDouble(index, Values.SPREAD, spread);

    if (index < 1) return;
    Double prev = series.getDouble(index-1, Values.SPREAD);
```

Get the instruments chosen by the user

Get the double value defined by 'input' (ie high, low etc) for the given instrument

2.10 Custom Context Menu

Support for custom context menus was added in version 5.3 of MotiveWave. This feature enables a user to interact with a study without having to open the study dialog. The following screen shot shows an example of a custom context menu in the Trend Line study example. In this example two additional items have been added to the context menu:

1. **Extend Left** – Extends the trend line to the left of the screen
2. **Extend Right** – Extends the trend line to the right of the screen

The following excerpt from the TrendLine example study class demonstrates how to add custom menu items. You can use the "plotName" (for composite studies) and "loc" parameters to customize the items depending on where the context menu is requested (where the user does the right click).

Whenever a menu item is invoked the study is recalculated. Typically the 'action' part of the menu item is to modify a setting in the study. When the study is recalculated, it will pick up the change to the study settings.

Figure 22 TrendLine Example Study

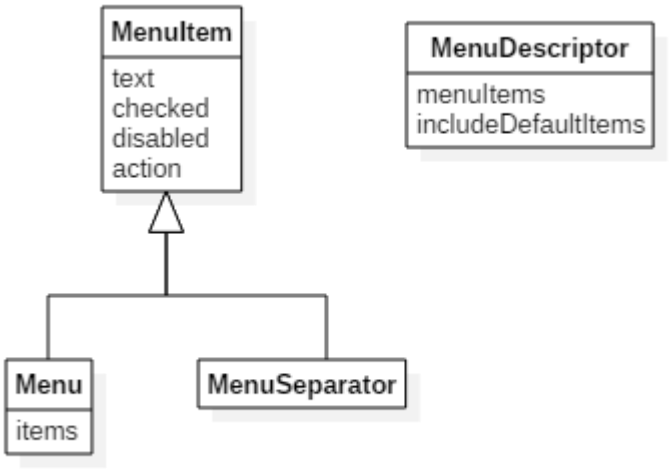
```
/** This study draws a trend line on the price graph and allows the user to move it using the resize points.
    The purpose of this example is to demonstrate advanced features such as using resize points and context menus. */
@StudyHeader(
    namespace="com.motivewave",
    id="TREND_LINE",
    rb="com.motivewave.platform.study.nls.strings",
    name="Trend Line",
    desc="This is an example study that draws a simple trend line and allows the user to resize it",
    overlay=true)
public class TrendLine extends com.motivewave.platform.sdk.study.Study
{
  final static String START="start", END="end";
  final static String EXT_RIGHT="extRight", EXT_LEFT="extLeft";

  @Override
  public void initialize(Defaults defaults)
  {
    ...
  }

  // Adds custom menu items to the context menu when the user right clicks on this study.
  @Override
  public MenuDescriptor onMenu(String plotName, Point loc, DrawContext ctx)
  {
    List<MenuItem> items = new ArrayList();
    items.add(new MenuSeparator());
    // Add some menu items for the user to extend right and left without having to open the study dialog
    boolean extLeft = getSettings().getBoolean(EXT_LEFT);
    boolean extRight = getSettings().getBoolean(EXT_RIGHT);

    // Note: the study will be recalculated (ie call calculateValues(), see below) when either of these menu items is invoked by the user
    items.add(new MenuItem("Extend Left", extLeft, () -> getSettings().setBoolean(EXT_LEFT, !extLeft)));
    items.add(new MenuItem("Extend Right", extRight, () -> getSettings().setBoolean(EXT_RIGHT, !extRight)));
    return new MenuDescriptor(items, true);
  }
```

Override onMenu() to add custom items to the Context Menu

Use the MenuItem class to define a menu item

The following diagram illustrates the classes used to define custom context menus for a study. Submenus can be created by using the Menu class (which contains a list of MenuItem, ie 'items'). The MenuSeparator class may be used to add dividers to the menu. Finally the MenuDescriptor class is used to describe the context menu. Use the 'includeDefaultItems' to show or hide the default menu items that are displayed as part of the context menu.

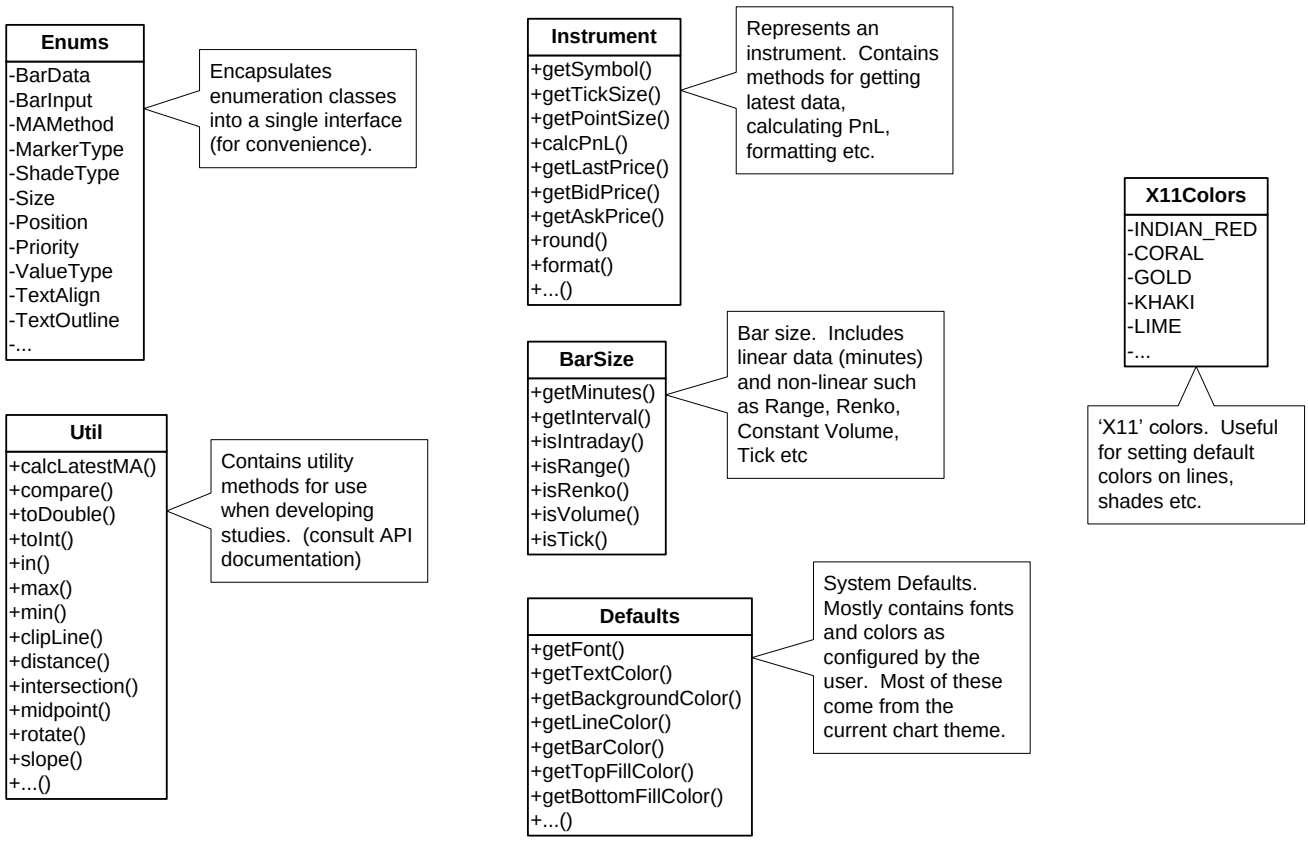
Figure 23 Package: com.motivewave.platform.sdk.common.menu



2.11 Miscellaneous Classes

The following diagram illustrates some additional classes that may be of interest. These classes are available in the common package (**com.motivewave.platform.sdk.common**). For full details on these and other classes, please consult the API documentation.

Figure 24 - Miscellaneous Classes



3 Overlay Example: 'My Moving Average'

In this section we will create a very simple example called 'My Moving Average' that displays an exponential moving average as a path on a plot.

Let's start by looking at the code for this example:

```
package study_examples;

import com.motivewave.platform.sdk.common.*;
import com.motivewave.platform.sdk.common.desc.*;
import com.motivewave.platform.sdk.study.*;

/** This simple example displays an exponential moving average. */
@StudyHeader(
 namespace="com.mycompany",
 id="MY_MA",
 name="My Moving Average",
 label="My MA",
 desc="This simple example displays an exponential moving average",
 menu="My Studies",
 overlay=true,
 studyOverlay=true)
public class MyMovingAverage extends Study
{
  enum Values { MA };

  /** This method initializes the study by doing the following:
      1. Define Settings (Design Time Information)
      2. Define Runtime Information (Label, Path and Exported Value) */
  @Override
  public void initialize(Defaults defaults)
  {
    // Describe the settings that may be configured by the user.
    // Settings may be organized using a combination of tabs and groups.
    SettingsDescriptor sd = new SettingsDescriptor();
    setSettingsDescriptor(sd);

    SettingTab tab = new SettingTab("General");
    sd.addTab(tab);

    SettingGroup inputs = new SettingGroup("Inputs");
    // Declare the inputs that are used to calculate the moving average.
    // Note: the 'Inputs' class defines several common input keys.
    // You can use any alpha-numeric string that you like.
    inputs.addRow(new InputDescriptor(Inputs.INPUT, "Input", Enums.BarInput.CLOSE));
    inputs.addRow(new IntegerDescriptor(Inputs.PERIOD, "Period", 20, 1, 9999, 1));
    tab.addGroup(inputs);

    SettingGroup colors = new SettingGroup("Display");
    // Allow the user to change the settings for the path that will
    // draw the moving average on the plot.  In this case, we are going
    // to use the input key Inputs.PATH
    colors.addRow(new PathDescriptor(Inputs.PATH, "Path", null, 1.0f, null, true, true, false));
    tab.addGroup(colors);

    // Describe the runtime settings using a 'StudyDescriptor'
    RuntimeDescriptor desc = new RuntimeDescriptor();
    setRuntimeDescriptor(desc);

    // Describe how to create the label.  The label uses the
    // 'label' attribute in the StudyHeader (see above) and adds the input values
    // defined below to generate a label.
    desc.setLabelSettings(Inputs.INPUT, Inputs.PERIOD);
    // Exported values can be used to display cursor data
    // as well as provide input parameters for other studies,
    // generate alerts or scan for study patterns (see study scanner).
    desc.exportValue(new ValueDescriptor(Values.MA, "My MA", new String[] {Inputs.INPUT, Inputs.PERIOD}));
    // MotiveWave will automatically draw a path using the path settings
    // (described above with the key 'Inputs.LINE')  In this case
```

```
    // it will use the values generated in the 'calculate' method
    // and stored in the data series using the key 'Values.MA'
    desc.declarePath(Values.MA, Inputs.PATH);
  }

  /** This method calculates the moving average for the given index in the data series. */
  @Override
  protected void calculate(int index, DataContext ctx)
  {
    // Get the settings as defined by the user in the study dialog
    // getSettings() returns a Settings object that contains all
    // of the settings that were configured by the user.
    Object input = getSettings().getInput(Inputs.INPUT);
    int period = getSettings().getInteger(Inputs.PERIOD);

    // In order to calculate the exponential moving average
    // we need at least 'period' points of data
    if (index < period) return;

    // Get access to the data series.
    // This interface provides access to the historical data as well
    // as utility methods to make this calculation easier.
    DataSeries series = ctx.getDataSeries();

    // This utility method allows us to calculate the Exponential
    // Moving Average instead of doing this ourselves.
    // The DataSeries interface contains several of these types of methods.
    Double average = series.ema(index, period, input);

    // Calculated values are stored in the data series using
    // a key (Values.MA).  The key can be any unique value, but
    // we recommend using an enumeration to organize these within
    // your class.  Notice that in the initialize method we declared
    // a path using this key.
    series.setDouble(index, Values.MA, average);
  }
}
```

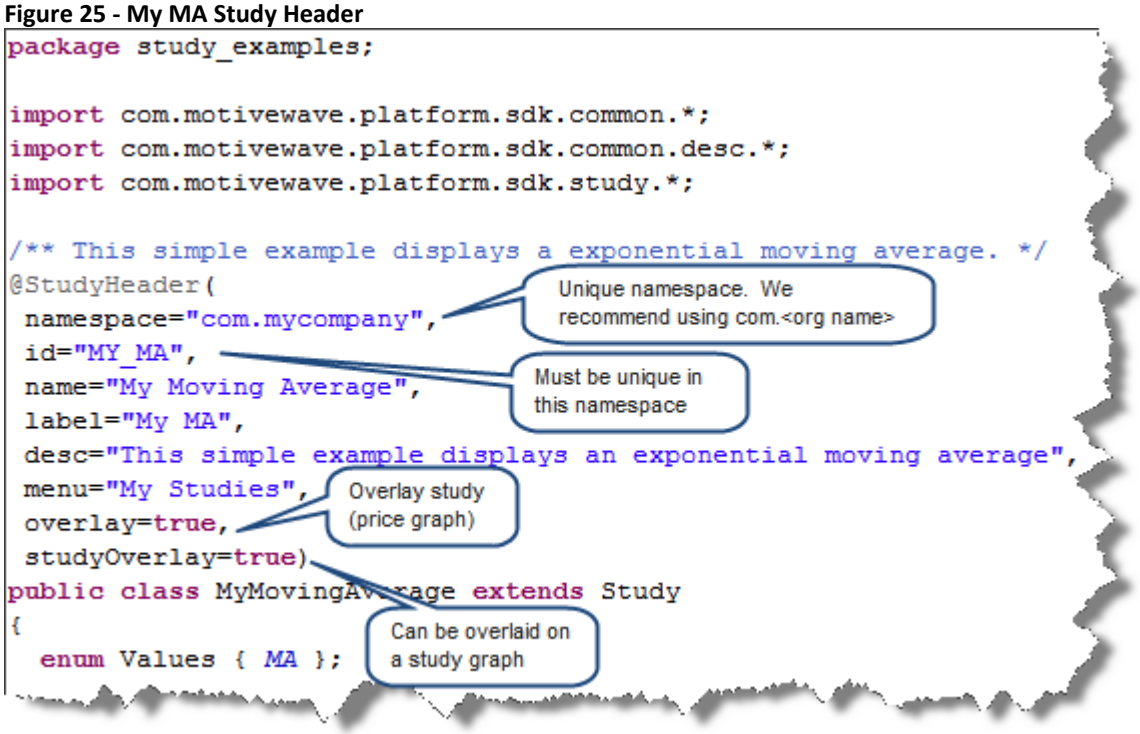
All studies must derive from the base class 'Study' (com.motivewave.platform.sdk.study.Study). This class contains a number of methods that we can override (we will look at these in detail later). For the purposes of this example, we will explore the following:

- StudyHeader
- initialize method
- calculate method

3.1 StudyHeader Annotation (@StudyHeader)

All studies must define a study header. This is an annotation that is placed before declaring the class:

Figure 25 - My MA Study Header



There are a number of important items in this header:

- **namespace** – this is used to qualify related studies and avoid naming conflicts with studies developed by third parties. It is recommended that you use a form similar to 'com.<name of your organization>' Together with the id tag, these form a globally unique identifier for your study
- **id** – this identifies your study and must be unique within your namespace
- **name** – This is the name of your study and is displayed in the study dialog as well as the study menu
- **label** – This is used as part of the study legend (displayed in the top left corner of the plot underneath the plot title). If not specified, the name attribute will be used.
- **desc** – This is the description of your study and is displayed in the study dialog
- **menu** – Identifies the menu (underneath the Study menu) where this study can be found
- **overlay** – If true indicates that this study will be an overlay displayed on another plot
- **studyOverlay** – Indicates that this study can be used as an overlay on a study plot.

3.2 initialize method

The 'initialize' method is used to perform any necessary initialization work when the study is created. This method is given access to system defaults (such as colors or fonts) available through the 'Defaults' class (see API documentation for specific details). The most common usage of this method is to do the following:

1. Describe Design Information (ie: inputs) – The SettingsDescriptor describes settings for the study and how to display this to the user (in the Study Dialog).

2. Describe Runtime Information – The StudyDescriptor describes information to MotiveWave™ so it knows how to handle this study at runtime (ie label settings, paths, exported values etc).

Figure 26 - My MA initialize method

```
public class MyMovingAverage extends Study
{
  enum Values { MA };
  @Override
  public void initialize(Defaults defaults)
  {
    // Describe the settings that may be configured by the user.
    // Settings may be organized using a combination of tabs and groups.
    SettingsDescriptor sd = new SettingsDescriptor();
    setSettingsDescriptor(sd);
    SettingTab tab = new SettingTab("General");
    sd.addTab(tab);

    SettingGroup inputs = new SettingGroup("Inputs");
    // Declare the inputs that are used to calculate the moving average.
    // Note: the 'Inputs' class defines several common input keys.
    // You can use any alpha-numeric string that you like.
    inputs.addRow(new InputDescriptor(Inputs.INPUT, "Input", Enums.BarInput.CLOSE));
    inputs.addRow(new IntegerDescriptor(Inputs.PERIOD, "Period", 20, 1, 9999, 1));
    tab.addGroup(inputs);

    SettingGroup colors = new SettingGroup("Display");
    // Allow the user to change the settings for the path that will
    // draw the moving average on the graph.  In this case, we are going
    // to use the input key Inputs.PATH
    colors.addRow(new PathDescriptor(Inputs.PATH, "Path", null, 1.0f,
                  null, true, true, false));
    tab.addGroup(colors);

    // Describe the runtime settings using a 'StudyDescriptor'
    RuntimeDescriptor desc = new RuntimeDescriptor();
    setRuntimeDescriptor(desc);

    // Describe how to create the label.  The label uses the
    // 'label' attribute in the StudyHeader (see above) and adds the input values
    // defined below to generate a label.
    desc.setLabelSettings(Inputs.INPUT, Inputs.PERIOD);
    // Exported values can be used to display cursor data
    // as well as provide input parameters for other stud
    // generate alerts or scan for study patterns (see study scanner).
    desc.exportValue(new ValueDescriptor(Values.MA, "My MA",
                     new String[] {Inputs.INPUT, Inputs.PERIOD}));
    // MotiveWave will automatically draw a path using the path settings
    // (described above with the key 'Inputs.LINE'
    // it will use the values generated in the 'c
    // and stored in the data series using the ke
    desc.declarePath(Values.MA, Inputs.PATH);
```

This enumeration defines the values generated by this study.

'General' tab

User Configurable 'Settings'

'Display' group

'Path' settings

Runtime 'Settings'

Exports the value 'MA' so it can be used outside this study

Tells MotiveWave to draw a path using the stored value 'MA' and the settings in Inputs.PATH

3.2.1 Design Time Information

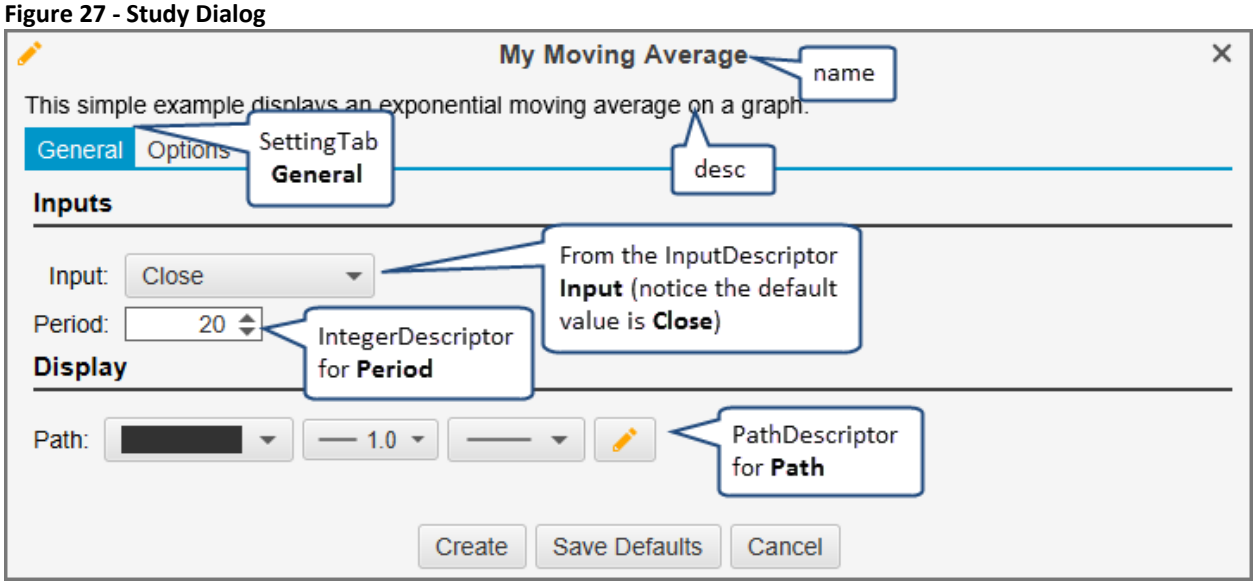
In our case, we need two types of inputs in order to calculate our exponential moving average:

1. **Input** – By default we will use the closing price for the bar (Enums.BarInput.CLOSE), but we will allow the user to choose something different (if they desire).
2. **Period** – This is the number of bars to look back when computing the average

For convenience, we will also allow the user to modify properties of the 'Path' such as the line color, style and weight.

The following diagram illustrates the Study Dialog that is presented to the user when they create or modify our study. Notice how the information described in the StudyHeader and the SettingsDescriptor are used to generate this dialog.

Figure 27 - Study Dialog



The classes used in this section are available from the package 'com.motivewave.platform.sdk.common.desc'. There are a number of classes in this package (see API documentation for full details). In this example we are concerned with the following:

- **SettingsDescriptor** – This class encapsulates all of the settings
- **SettingTab** – Used to organize settings into 'Tabs' that are displayed in the Study Dialog
- **SettingGroup** – Organizes settings within a tab into logical groups
- **Setting Descriptors** – MotiveWave™ has many setting descriptors (base class SettingDescriptor). The ones used in this example are:
  - **InputDescriptor** – Inputs used to calculate values. Typically these are historical data inputs such as open, high, low or close values, but may also include derived values (such as weighted price) or values generated by other studies.
  - **IntegerDescriptor** – Describes an integer input value. This can be constrained to a specified range (1 – 9999 in this case)
  - **PathDescriptor** – Describes how to render the path. In this case the user can choose the line width, style and color

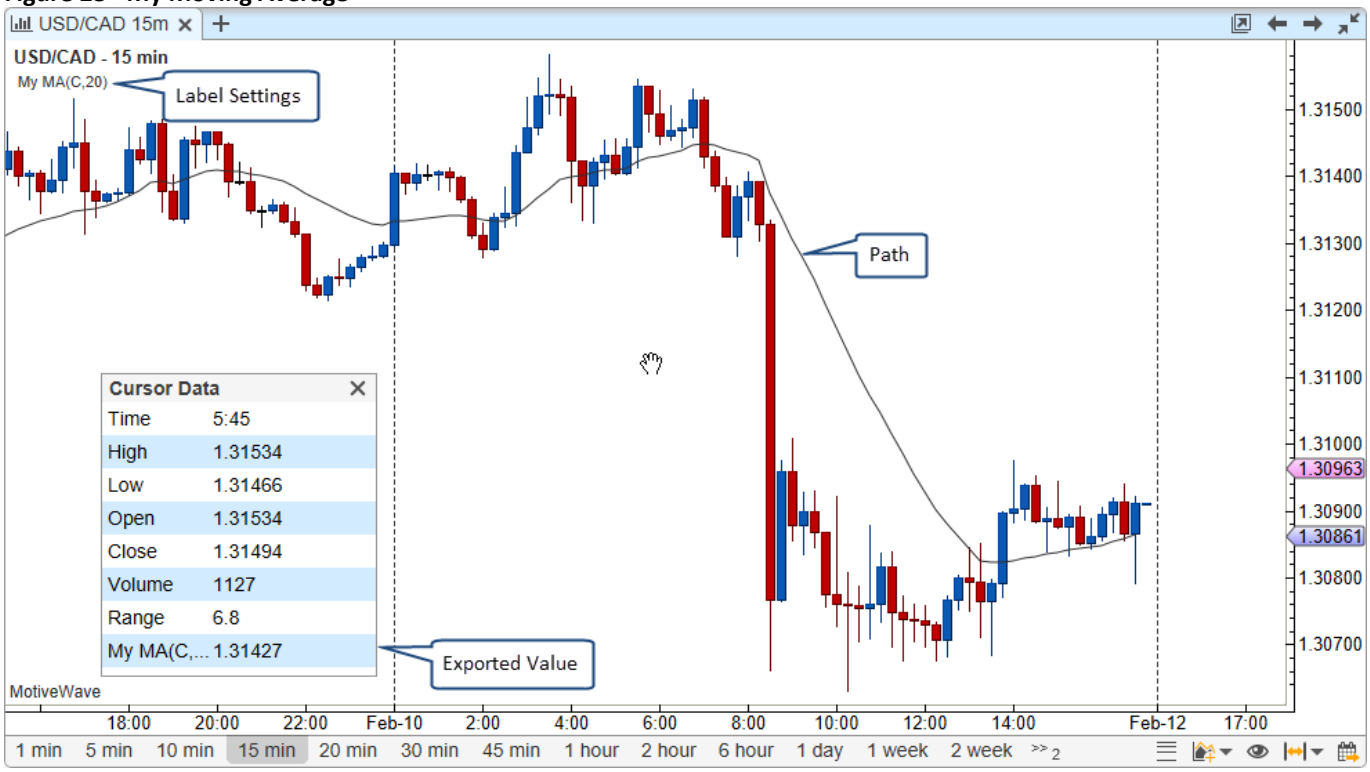
3.2.2 Run Time Information

Run time information is specified using the RuntimeDescriptor. For the purposes of our example, this will include the following:

- **Label Settings** – Describes how to create and display the label (study legend) for this study. In our case we want the label to include the Input and Period. For example, with an input of CLOSE and a period of 20, the label will look like: 'My MA(C,20)'
- **Declare Path** – Tell MotiveWave™ to create and draw a path using the information created by the PathDescriptor and the values generated by the study
- **Export Value** – Exported values may be used for a number of purposes, most notably:
  - **Cursor Data** – Displaying information in the Cursor Data Window
  - **Input for Other Studies** – Exported values can be used as input to other studies
  - **Input for Alerts** – Alerts can be created to be triggered off of study values
  - **Study Scan** – When creating a study scanner, these exported values can be used to find specific conditions.

The following screenshot displays what our study looks like at Runtime:

Figure 28 - My Moving Average



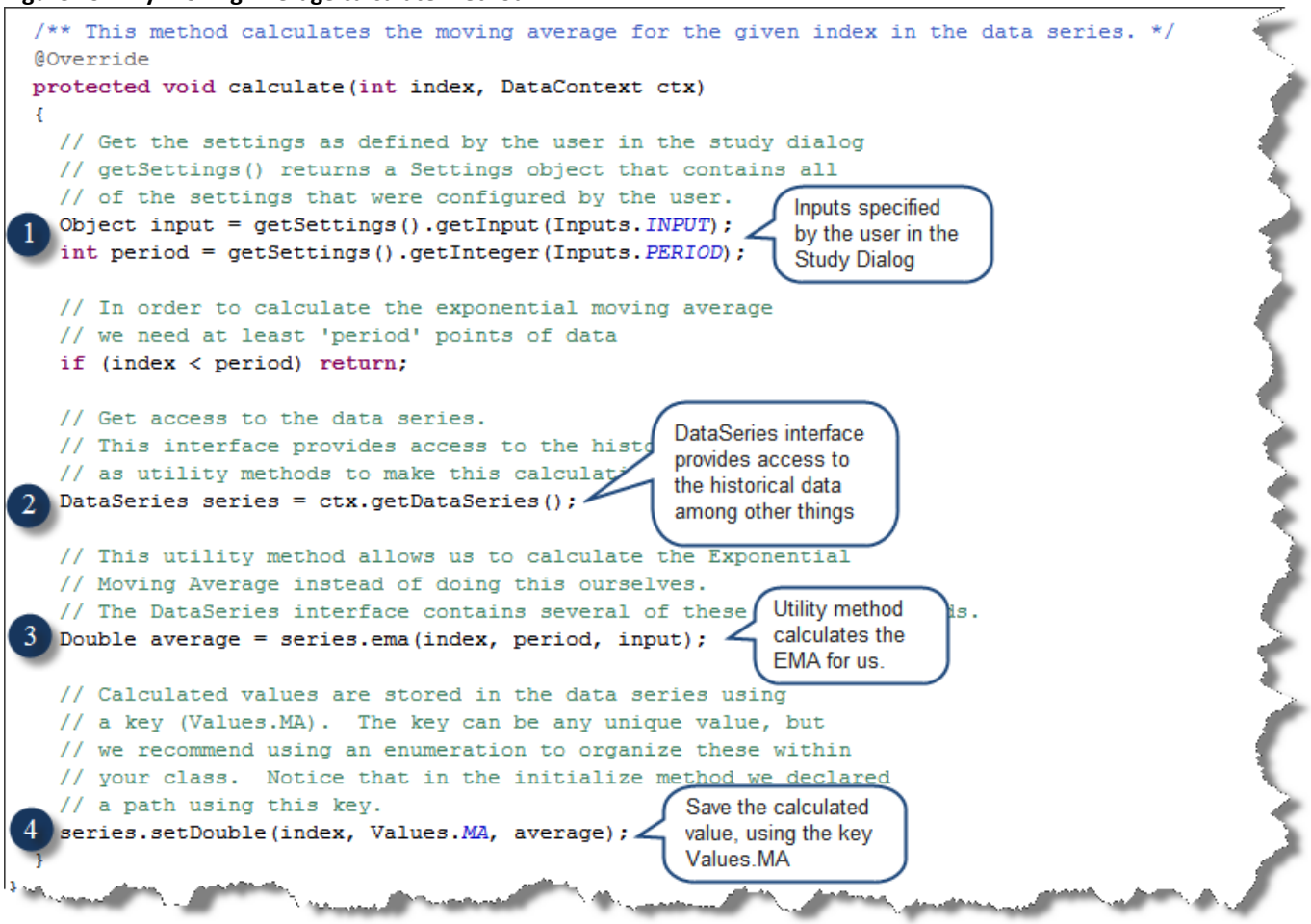
3.3 calculate method

This method is used to calculate the value(s) for a particular bar in the data series (identified by the index parameter). This method is called by the 'calculateValues' method for every bar in the data series. Alternatively, you could override the 'calculateValues' method if you want to handle the creation of all values for the data series.

In this case we are going to do the following:

1. **Retrieve the User Settings** – '*getSettings()*' returns a reference to the Settings object.
2. **Get the *DataSeries*** – This is the interface to the historical data and a repository for any values computed by the study. This also contains several utility methods for computing values such as moving averages.
3. **Compute the EMA** – this is done by calling the utility method '*ema*' with the input specified by the user.
4. **Store the EMA in the data series** – This value is stored at the given index using the key: *Values.MA*

Figure 29 - My Moving Average calculate method



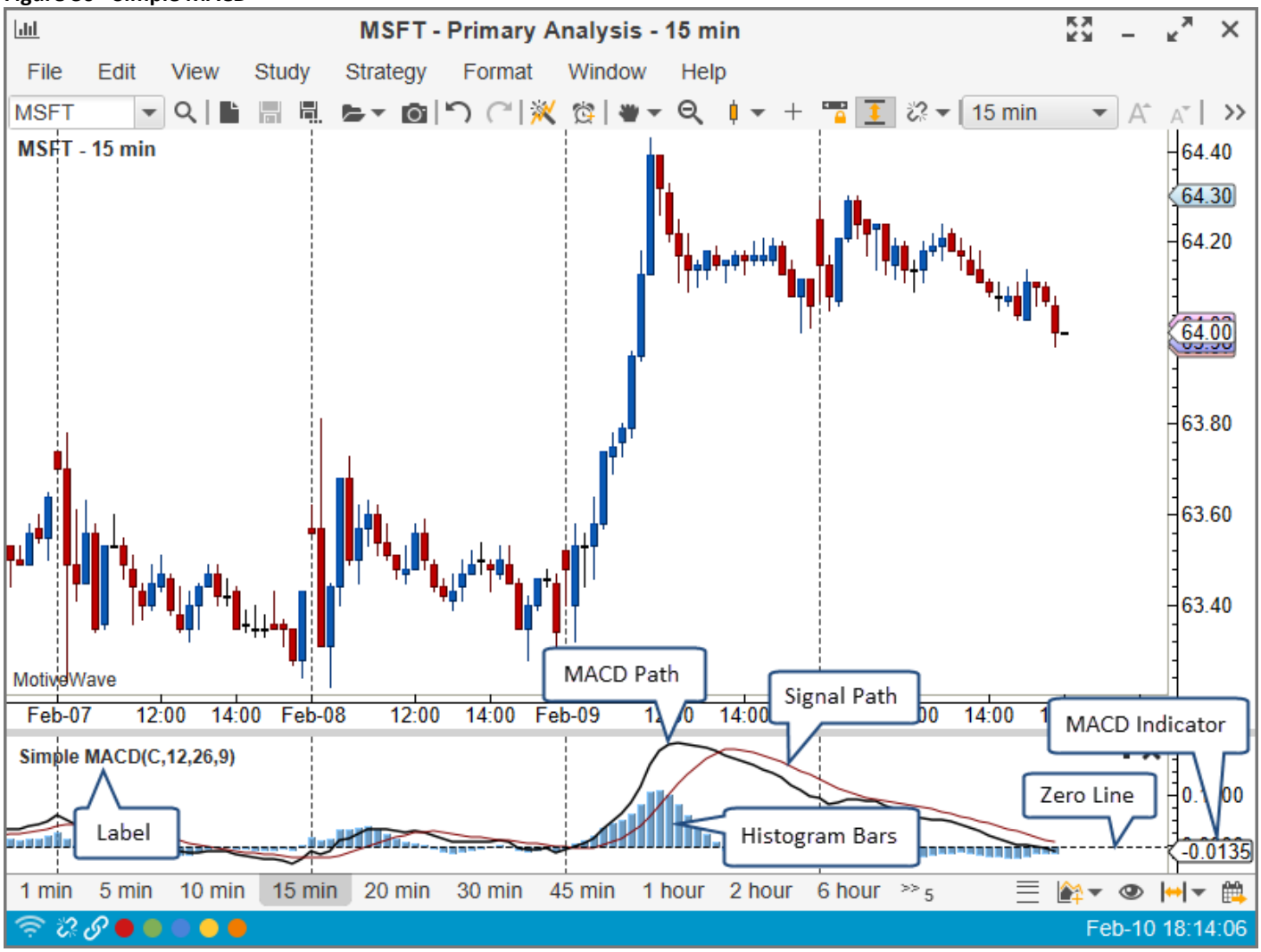
4 Study Plot Example: 'Simple MACD'

In this example we are going to create a *Study Plot* based on a simple MACD. Note: if you would like a more comprehensive MACD example, you can look at the source code for the MACD indicator that exists within MotiveWave™.

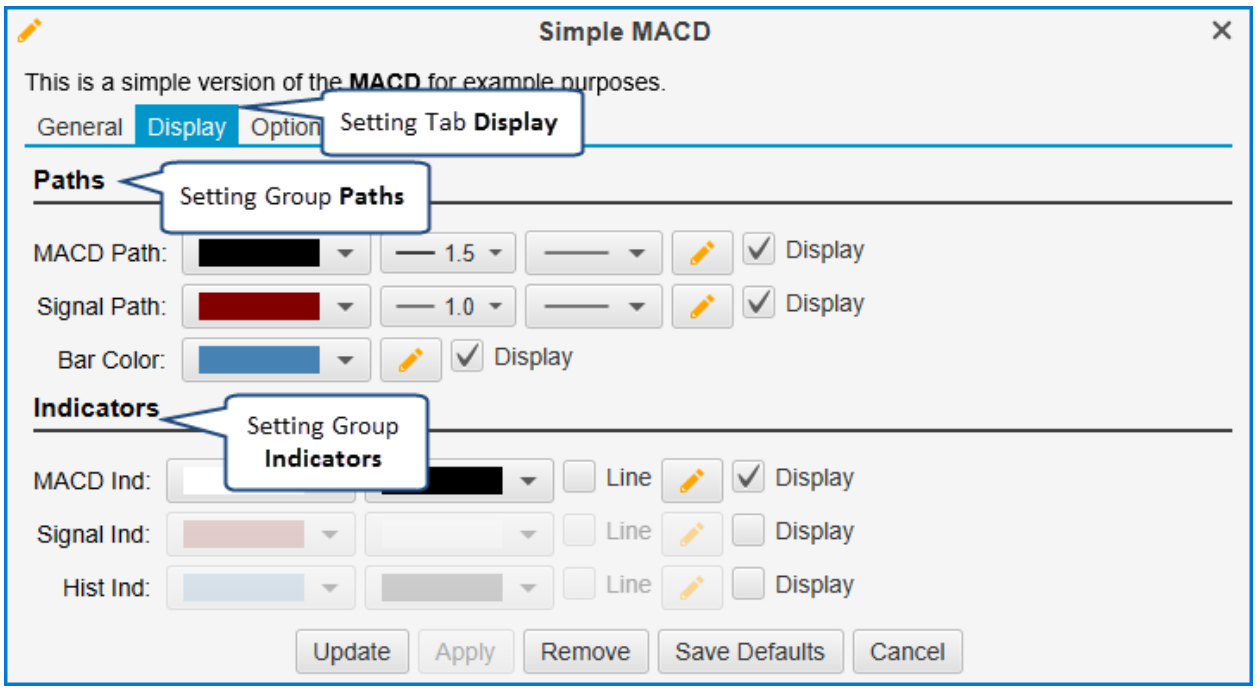
MACD stands for 'Moving Average Convergence/Divergence' and was written by Gerald Appel in the 1970s. If you would like more information on this study go to: http://en.wikipedia.org/wiki/MACD.

Here is a screen shot of what this study looks like:

Figure 30 - Simple MACD



Here is a screen shot of the Study Dialog that the user will use to configure the Simple MACD:



Let us start by looking at the source code for this study:

```
package study_examples;

import com.motivewave.platform.sdk.common.*;
import com.motivewave.platform.sdk.common.desc.*;
import com.motivewave.platform.sdk.study.*;

/** Simple MACD example.  This example shows how to create a Study Plot
    that is based on the MACD study.  For simplicity code from the
    MotiveWave MACD study has been removed or altered. */
@StudyHeader(
    namespace="com.mycompany",
    id="SimpleMACD",
    name="Simple MACD",
    desc="This is a simple version of the <b>MACD</b> for example purposes.",
    menu="My Studies",
    overlay=false)
public class SimpleMACD extends Study
{
  // This enumeration defines the variables that we are going to store in the
  // Data Series
       enum Values { MACD, SIGNAL, HIST };
  final static String HIST_IND = "histInd"; // Histogram Parameter

  /** This method initializes the settings and defines the runtime settings. */
  @Override
  public void initialize(Defaults defaults)
  {
    // Define the settings for this study
    // We are creating 2 tabs: 'General' and 'Display'
    SettingsDescriptor settings = new SettingsDescriptor();
    setSettingsDescriptor(settings);
    SettingTab tab = new SettingTab("General");
    settings.addTab(tab);

    // Define the 'Inputs'
    SettingGroup inputs = new SettingGroup("Inputs");
    inputs.addRow(new InputDescriptor(Inputs.INPUT, "Input", Enums.BarInput.CLOSE));
    inputs.addRow(new IntegerDescriptor(Inputs.PERIOD, "Period 1", 12, 1, 9999, 1));
    inputs.addRow(new IntegerDescriptor(Inputs.PERIOD2, "Period 2", 26, 1, 9999, 1));
    inputs.addRow(new IntegerDescriptor(Inputs.SIGNAL_PERIOD, "Signal Period", 9, 1, 9999, 1));
    tab.addGroup(inputs);
```

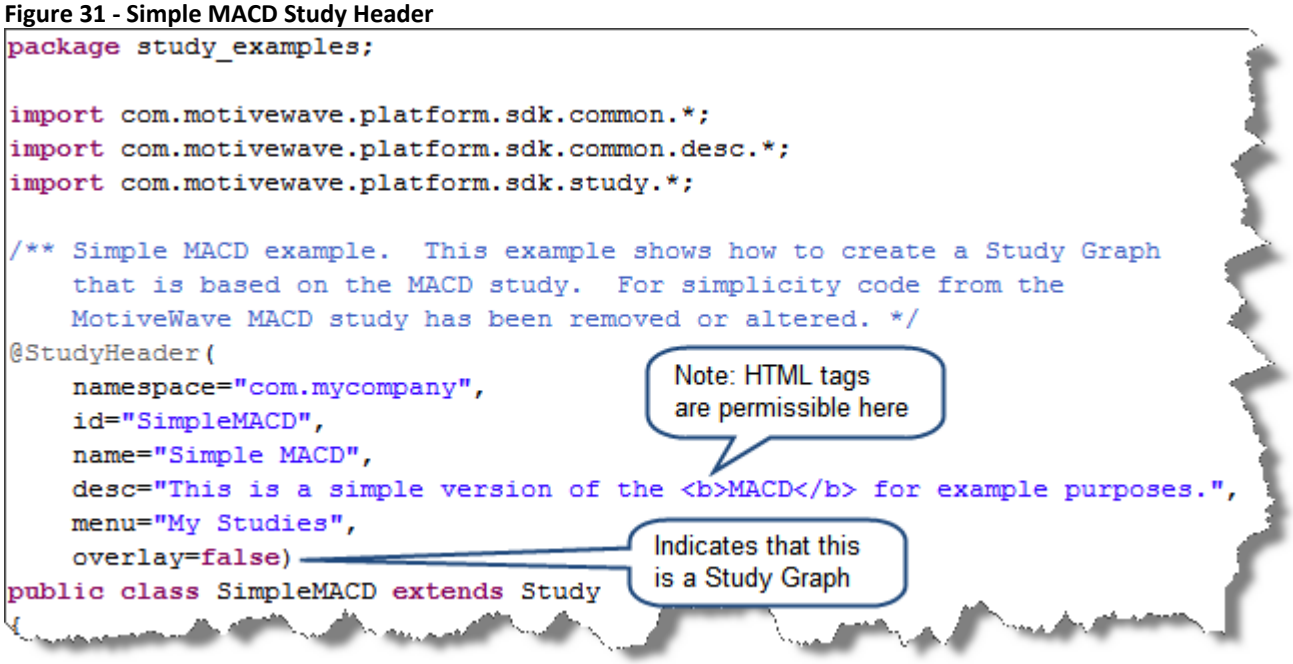
```
    tab = new SettingTab("Display");
    settings.addTab(tab);
    // Allow the user to configure the settings for the paths and the histogram
    SettingGroup paths = new SettingGroup("Paths");
    tab.addGroup(paths);
    paths.addRow(new PathDescriptor(Inputs.PATH, "MACD Path",
                 defaults.getLineColor(), 1.5f, null, true, false, true));
    paths.addRow(new PathDescriptor(Inputs.SIGNAL_PATH, "Signal Path",
                 defaults.getRed(), 1.0f, null, true, false, true));
    paths.addRow(new BarDescriptor(Inputs.BAR, "Bar Color", defaults.getBarColor(), true, true));
    // Allow the user to display and configure indicators on the vertical axis
    SettingGroup indicators = new SettingGroup("Indicators");
    tab.addGroup(indicators);
    indicators.addRow(new IndicatorDescriptor(Inputs.IND, "MACD Ind",
                      null, null, false, true, true));
    indicators.addRow(new IndicatorDescriptor(Inputs.SIGNAL_IND, "Signal Ind",
                      defaults.getRed(), null, false, false, true));
    indicators.addRow(new IndicatorDescriptor(HIST_IND, "Hist Ind",
                      defaults.getBarColor(), null, false, false, true));

    RuntimeDescriptor desc = new RuntimeDescriptor();
    setRuntimeDescriptor(desc);
    desc.setLabelSettings(Inputs.INPUT, Inputs.PERIOD, Inputs.PERIOD2, Inputs.SIGNAL_PERIOD);
    // We are exporting 3 values: MACD, SIGNAL and HIST (histogram)
    desc.exportValue(new ValueDescriptor(Values.MACD, "MACD", new String[]
                     {Inputs.INPUT, Inputs.PERIOD, Inputs.PERIOD2}));
    desc.exportValue(new ValueDescriptor(Values.SIGNAL, "MACD Signal",
                     new String[] {Inputs.SIGNAL_PERIOD}));
    desc.exportValue(new ValueDescriptor(Values.HIST, "MACD Histogram", new String[]
                     {Inputs.PERIOD, Inputs.PERIOD2, Inputs.SIGNAL_PERIOD}));
    // There are two paths, the MACD path and the Signal path
    desc.declarePath(Values.MACD, Inputs.PATH);
    desc.declarePath(Values.SIGNAL, Inputs.SIGNAL_PATH);
    // Bars displayed as the histogram
    desc.declareBars(Values.HIST, Inputs.BAR);
    // These are the indicators that are displayed in the vertical axis
    desc.declareIndicator(Values.MACD, Inputs.IND);
    desc.declareIndicator(Values.SIGNAL, Inputs.SIGNAL_IND);
    desc.declareIndicator(Values.HIST, HIST_IND);

    // These variables are used to define the range of the vertical axis
    desc.setRangeKeys(Values.MACD, Values.SIGNAL, Values.HIST);
    // Display a 'Zero' line that is dashed.
    desc.addHorizontalLine(new LineInfo(0, null, 1.0f, new float[] {3,3}));
  }

  /** This method calculates the MACD values for the data at the given index. */
  @Override
  protected void calculate(int index, DataContext ctx)
  {
    int period1 = getSettings().getInteger(Inputs.PERIOD);
    int period2 = getSettings().getInteger(Inputs.PERIOD2);
    int period = Util.max(period1, period2);
    if (index < period) return; // not enough data to compute the MAs

    // MACD is the difference between two moving averages.
    // In our case we are going to use an exponential moving average (EMA)
    Object input = getSettings().getInput(Inputs.INPUT);
    DataSeries series = ctx.getDataSeries();
    Double MA1 = null, MA2 = null;

    MA1 = series.ema(index, period1, input);
    MA2 = series.ema(index, period2, input);
    if (MA1 == null || MA2 == null) return;

    // Define the MACD value for this index
    double MACD = MA1 - MA2;
    series.setDouble(index, Values.MACD, MACD);

    int signalPeriod = getSettings().getInteger(Inputs.SIGNAL_PERIOD);
    if (index < period + signalPeriod) return; // Not enough data yet
```

```
    // Calculate moving average of MACD (signal path)
    Double signal = series.sma(index, signalPeriod, Values.MACD);
    series.setDouble(index, Values.SIGNAL, signal);
    if (signal == null) return;

    // Histogram is the difference between the MACD and the signal path
    series.setDouble(index, Values.HIST, MACD - signal);
    series.setComplete(index);
  }
}
```

4.1 StudyHeader Annotation (@StudyHeader)

The main difference in the study header from the previous example is the 'overlay' tag is set to false. This indicates to MotiveWave™ that this study should be displayed in a separate study plot. You will notice here as well that we have included some HTML markup in the *'desc'* tag. The description displayed in the Study Dialog supports HTML so you can put any valid HTML tags here (do not include JavaScript, this is not supported).

Figure 31 - Simple MACD Study Header



4.2 initialize method

We have defined a bit more in the initialize section from the previous example. To illustrate the usage of tabs, we have created 2 tabs: 'General' and 'Display'. We have also defined the bars for the histogram (see BarDescriptor).

Indicators are displayed on the vertical axis (right side of the screen). By default, we are only going to show the first indicator (MACD), but we will allow the user to show indicators for the current signal value as well as the histogram. For this we will use the *IndicatorDescriptor* and set the values accordingly. We have organized these into a Setting Group called 'Indicators'

The following screen shot (with markup) shows the part of the *initialize* method where we are describing the settings for the study:

Figure 32 - Simple MACD initialize settings


```
/** This method initializes the settings and de                    */
@Override
public void initialize(Defaults defaults)
{
  // Define the settings for this study
  // We are creating 2 tabs: 'General' and 'Dis
  SettingsDescriptor settings = new SettingsDescriptor();
  setSettingsDescriptor(settings);
  SettingTab tab = new SettingTab("General");
  settings.addTab(tab);

  // Define the 'Inputs'
  SettingGroup inputs = new SettingGroup("Inputs");
  inputs.addRow(new InputDescriptor(Inputs.INPUT, "Input", Enums.BarInput.CLOSE));
  inputs.addRow(new IntegerDescriptor(Inputs.PERIOD, "Period 1", 12, 1, 9999, 1));
  inputs.addRow(new IntegerDescriptor(Inputs.PERIOD2, "Period 2", 26, 1, 9999, 1));
  inputs.addRow(new IntegerDescriptor(Inputs.SIGNAL_PERIOD, "Signal Period",
              9, 1, 9999, 1));
  tab.addGroup(inputs);

  tab = new SettingTab("Display");
  settings.addTab(tab);
  // Allow the user to configure the settings for the paths and the histogram
  SettingGroup paths = new SettingGroup("Paths");
  tab.addGroup(paths);
  paths.addRow(new PathDescriptor(Inputs.PATH, "MACD Path",
              defaults.getLineColor(), 1.5f, null, true, false, true));
  paths.addRow(new PathDescriptor(Inputs.SIGNAL_PATH, "Signal Path",
              defaults.getRed(), 1.0f, null, true, false, true));
  paths.addRow(new BarDescriptor(Inputs.BAR, "Bar Color",
              defaults.getBarColor(), true, true));
  // Allow the user to display and configure indicators on the vertical axis
  SettingGroup indicators = new SettingGroup("Indicators");
  tab.addGroup(indicators);
  indicators.addRow(new IndicatorDescriptor(Inputs.IND, "MACD Ind",
                  null, null, false, true, true));
  indicators.addRow(new IndicatorDescriptor(Inputs.SIGNAL_IND, "Signal In
                  defaults.getRed(), null, false, false, true));
  indicators.addRow(new IndicatorDescriptor(HIST_IND, "Hist Ind",
                  defaults.getBarColor(), null, false, false, true));
```

Next, we need to describe the runtime parameters using the *RuntimeDescriptor*. For the label, we want to append the input, period, period2 and the signal period.

In this case, we are going to export 3 values: MACD, SIGNAL and HIST.

In order to display the histogram as bars, we use the '*declareBars*' method on the study descriptor. This will tell MotiveWave™ to show vertical bars using the *BarDescriptor* identified by *Inputs.BAR.*

Figure 33 - Simple MACD initialize runtime

```
/** This method initializes the settings and defines the runtime settings. */
@Override
public void initialize(Defaults defaults)
{
  // Define the settings for this study
  // We are creating 2 tabs: 'General' and 'Display'
  SettingsDescriptor settings = new SettingsDescriptor();
  setSettingsDescriptor(settings);
    . . .
  RuntimeDescriptor desc = new RuntimeDescriptor();
  setRuntimeDescriptor(desc);
  desc.setLabelSettings(Inputs.INPUT, Inputs.PERIOD,
                        Inputs.PERIOD2, Inputs.SIGNAL_PERIOD);
  // We are exporting 3 values: MACD, SIGNAL and HIST (histogram)
  desc.exportValue(new ValueDescriptor(Values.MACD, "MACD", new St
                   {Inputs.INPUT, Inputs.PERIOD, Inputs.PERIOD2}))
  desc.exportValue(new ValueDescriptor(Values.SIGNAL, "MACD Signal
                   new String[] {Inputs.SIGNAL_PERIOD}));
  desc.exportValue(new ValueDescriptor(Values.HIST, "MACD Histogram", new String[
                   {Inputs.PERIOD, Inputs.PERIOD2, Inputs.SIGNAL_PERIOD}));
  // There are two paths, the MACD path and the Signal path
  desc.declarePath(Values.MACD, Inputs.PATH);
  desc.declarePath(Values.SIGNAL, Inputs.SIGNAL_PATH);
  // Bars displayed as the histogram
  desc.declareBars(Values.HIST, Inputs.BAR);
  // These are the indicators that are displayed in the vertical axis
  desc.declareIndicator(Values.MACD, Inputs.IND);
  desc.declareIndicator(Values.SIGNAL, Inputs.SIGNAL_IND);
  desc.declareIndicator(Values.HIST, HIST_IND);

  // These variables are used to define the range of the vertic
  desc.setRangeKeys(Values.MACD, Values.SIGNAL, Values.HIST);
  // Display a 'Zero' line that is dashed.
  desc.addHorizontalLine(new LineInfo(0, null, 1.0f, new float[] {3,3}));
}
```

Export Values. These can be displayed in the Cursor Data Window or used as inputs to other studies.

Declare Paths

Declare the Indicators

This values determine the range of the vertical axis

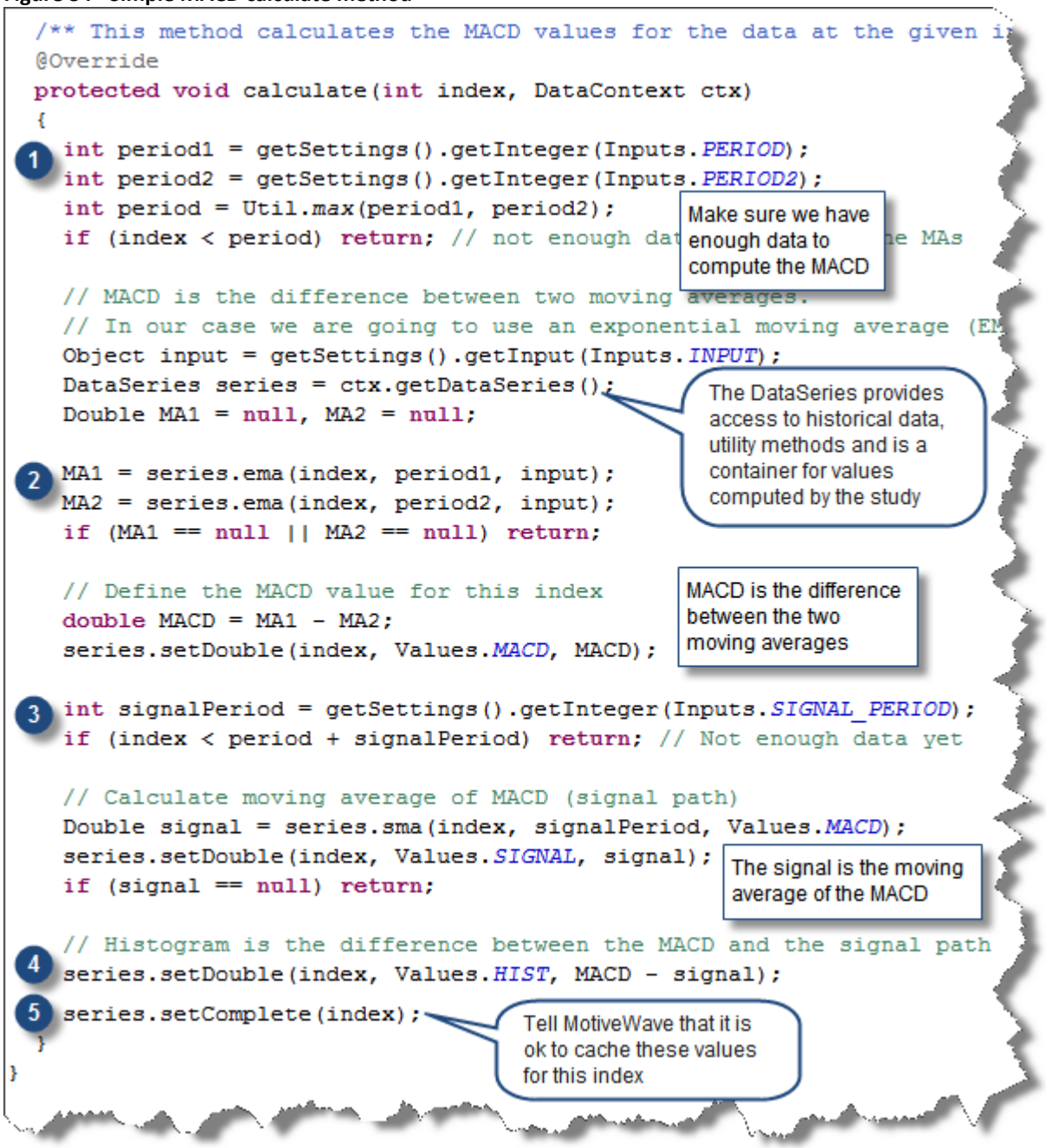
4.3 calculate Method

The calculate method is used to compute the values for each historical bar in the data series. In our case, we are going to do the following:

1. **Retrieve User Settings** – these are accessed from the *getSettings()* method.
2. **Compute and Store the MACD** – The *DataSeries* object contains the historical data as well as the utility methods for computing moving averages. The MACD value is stored in the data series at the given index using the key *Values.MACD*.

3. **Compute and Store the signal** – The signal is a moving average of the MACD. Use the data series to compute the moving average with *Values.MACD* as the key. The signal value is stored in the data series at the given index using the key: *Values.SIGNAL*.
4. **Compute and store the histogram** – The histogram is simply the difference between the MACD and the signal. This is stored in the data series at the given index using the key: *Values.HIST*.
5. **Mark the index as 'Complete'** - Finally, indicate that this index is 'complete'. This allows MotiveWave™ to cache these values (to improve performance).

Figure 34 - Simple MACD calculate method

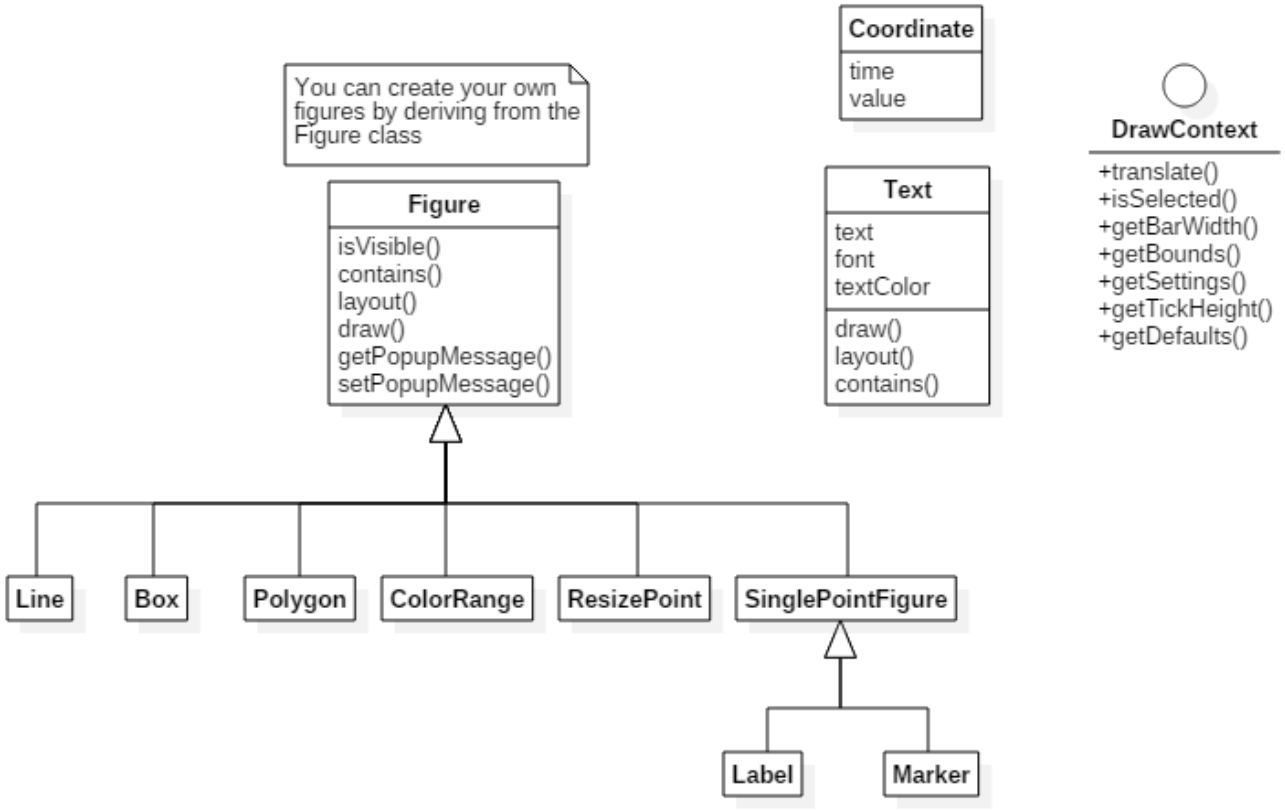


5 Drawing Figures

The draw package (**com.motivewave.platform.sdk.draw**) contains classes for drawing figures (markers, lines etc) as part of the study. Additional classes will likely be added to this package as the SDK evolves.

All figures have one or more *Coordinate* values (see common package) that are used to specify the location of the figure. These coordinates are composed of a 'real' time and value that are translated to plot (x,y) points before they are drawn.

Figure 35 - draw classes



The following methods are available on the Study class for working with figures:

- clearFigures() – clears all figures from the study
- addFigure(Figure f) - adds a figure to the study
- removeFigure(Figure f) – removes an existing figure
- getFigures() – gets all of the figures added to the study

5.1 Figure Class

The *Figure* class is the base class for all figures that may be drawn as part of the study. You may derive from the class to create a custom figure to display as part of the study. This class consists of the following methods:

1. **isVisible(DrawContext ctx)** – returns true if this figure is currently visible in the given draw context. This is used by the study framework to improve performance by only working with figures that are currently visible.
2. **contains(double x, double y, DrawContext ctx)** – returns true if the figure contains the given (x,y) coordinates. This is used by the study framework to determine if the mouse pointer is currently above the study (and is selectable).
3. **layout(DrawContext ctx)** – This method is used to prepare the figure to be drawn. Typically coordinates are translated to plot values (x,y pixel locations) and any intermediate draw figures are created.
4. **draw(Graphics2D gc, DrawContext ctx)** – This method draws the figure on the plot.
5. **getPopupMessage(double x, double y, DrawContext ctx)** – Gets a popup message to display when the user hovers above the figure. The (x,y) parameters are the coordinates of the mouse on the chart.
6. **setPopMessage(String msg)** – Sets the message to display when the mouse is hovering over the figure. If this method is called there is no need to override the getPopupMessage(…) method above.
7. **get/setBounds()** – Use these methods to define the bounding rectangle for the figure. By default the contains(…) method will use this to test if the figure contains the given (x,y) parameters.

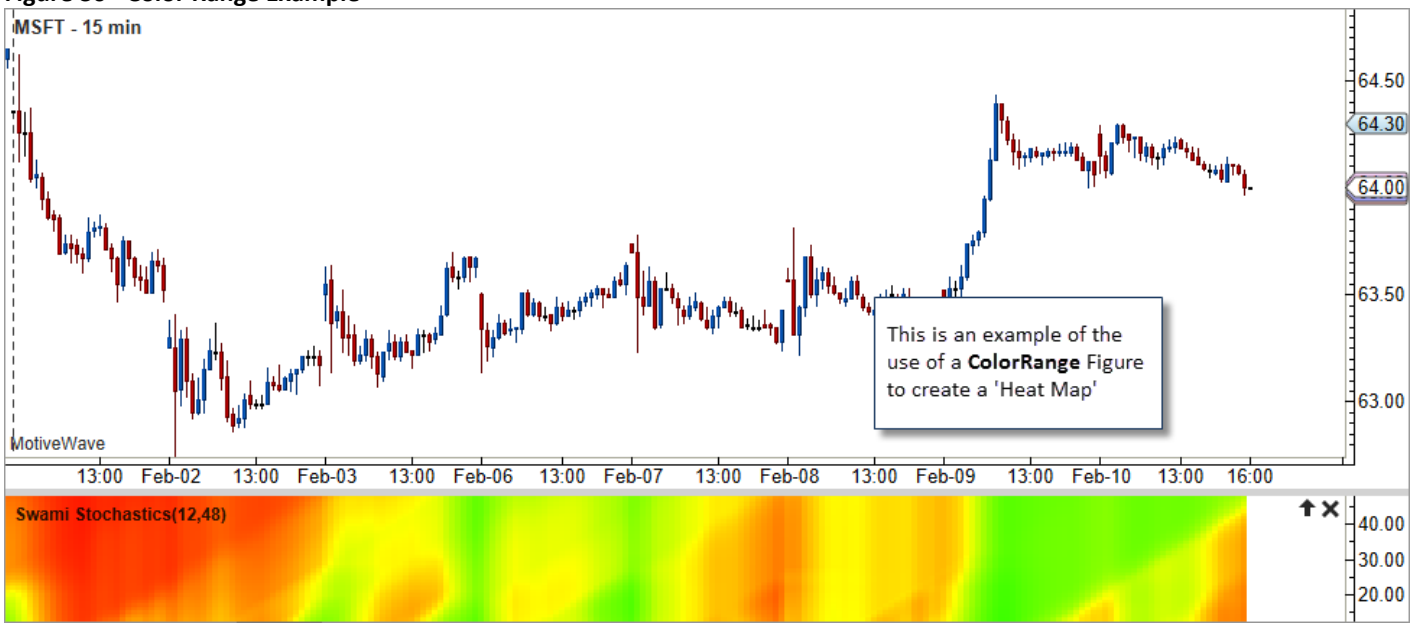
5.2 Box

Use this class to draw a rectangular box with an optional fill color.

5.3 ColorRange Class

This class is convenient for creating 'Heat Map' studies. A good example of this is the Swami Stochastics study. Each ColorRange object is essentially a bar that has a series of colors regions defined for a range of values. The following screen shot illustrates what this looks like:

Figure 36 - Color Range Example



5.4 Line Class

The *Line* class is useful for drawing trend lines or vertical/horizontal lines. There are several convenience options in this class for extending the line, setting the color and line style. You can even have the line draw a different color above and below a given value.

5.5 Polygon

Draws a shape with 3 or more points.

5.6 ResizePoint

This is a special type of figure that enables users to interact with a study using the mouse. Users can drag a resize point to a specific location. The study will receive the following resize events:

1. onBeginResize(ResizePoint rp, DrawContext ctx) – This is called when the user begins a drag operation on the resize point
2. onResize(ResizePoint rp, DrawContext ctx) – This is called as the user drags the resize point. It gives the study an opportunity to provide visual feedback as the user moves the mouse.
3. onEndResize(ResizePoint rp, DrawContext ctx) – This is called when the drag operation is completed. Override this method to store changes in the settings. The study will be recalculated after this method is called.

The follow excerpt from the TrendLine example study (see Study Examples project) shows an example of using the onResize() and onEndResize() methods:

```
package study_examples;

import java.awt.Graphics2D;

/** This study draws a trend line on the price graph and allows the user to move it using the resize points.
    The purpose of this example is to demonstrate advanced features such as using resize points and context menus. */
@StudyHeader(
    namespace="com.motivewave",
    id="TREND_LINE",
    name="Trend Line",
    desc="This is an example study that draws a simple trend line and allows the user to resize it",
    overlay=true)
public class TrendLine extends com.motivewave.platform.sdk.study.Study
{
  final static String START="start", END="end";
  final static String EXT_RIGHT="extRight", EXT_LEFT="extLeft";

  @Override
  public void initialize(Defaults defaults)
  {
    ...
  }

  // This method is called when the user is moving a resize point but has not released the mouse button yet.
  // This does not cause the study to be recalculated until the resize operation is completed.
  @Override
  public void onResize(ResizePoint rp, DrawContext ctx)
  {
    // In our case we want to adjust the trend line as the user moves the resize point
    // This will provide visual feedback to the user
    trendLine.layout(ctx);
  }

  // This method is called when the user has completed moving a resize point with the mouse.
  // The underlying study framework will recalculate the study after this method is called.
  @Override
  public void onEndResize(ResizePoint rp, DrawContext ctx)
  {
    // Commit the resize to the study settings, so it can be used in calculateValues() (see below)
    // We will store this in the settings as a string: "<price>|<time in millis>"
    getSettings().setString(rp == startResize ? START : END, rp.getValue() + "|" + rp.getTime());
  }
```

Shows the trend line moving with the resize point

Operation completed. Save the new position in the settings.

5.6.1 Resize Types

There are 3 types of resize points supported to constrain how they can be adjusted by the user. These types are defined in the enumeration ResizeType (found in the Enums interface):

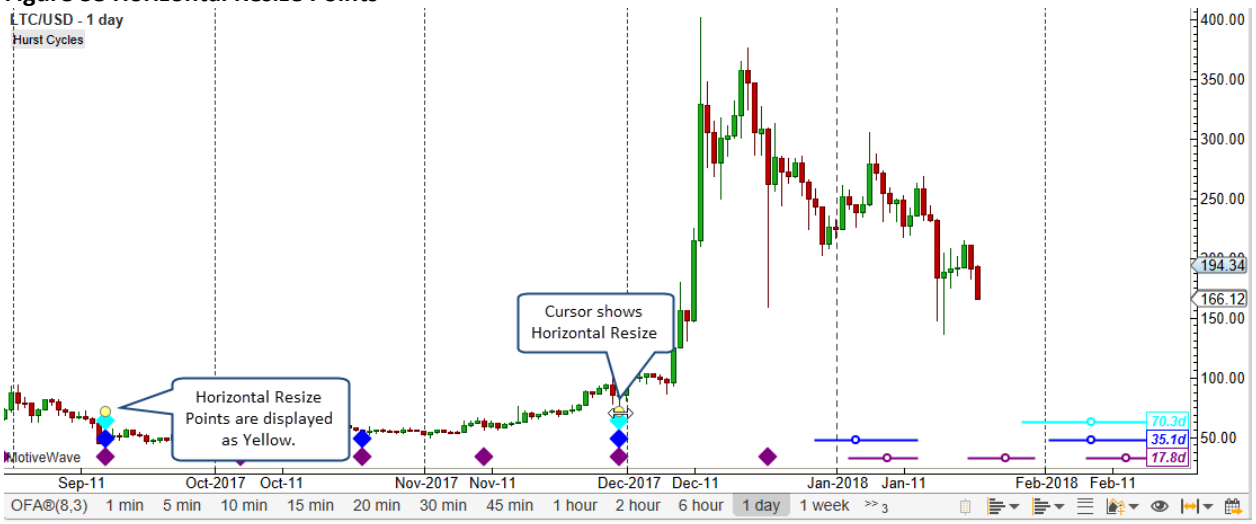
1. Horizontal – This type of resize point can only be moved left or right. Its vertical position will remain constant. By default, these resize points will be colored yellow.
2. Vertical – Only up and down movement is allowed. Its horizontal (x) position will remain constant . By default, these resize points will be colored yellow.
3. All – Unrestricted. The user can move these types of points anywhere on the screen. By default these types of resize points will be colored green.

The screen shot below shows an example of using resize points that have a type of 'All'. This is from the example TrendLine study:

Figure 37 Resize Points of Type 'All'

The screen shot below illustrates the use of a 'Horizontal' resize point to position the cycles in the Hurst Cycles study:

Figure 38 Horizontal Resize Points



5.6.2 Absolute Positioning

The location of a resize point can be relative or absolute. If a resize point is absolute (see 'absolute' property on the ResizePoint class) then its position is defined by a specific time and value (usually price). If relative positioning is used then the resize point is specified using the (x,y) screen coordinates on the chart.

5.7 SinglePointFigure

A SinglePointFigure is a special type of Figure that defines figures that are located on the chart by a single point (coordinate). For convenience, multiple figures that are located at the same coordinate may be may be "stacked" above or below each other to improve readability.

This class currently has two subclasses:

1. Marker
2. Label

5.7.1 Marker Class

The *Marker* class makes it convenient to highlight points of interest on the plot. Often this class is used in conjunction with signals. There are several different types of markers (triangle, arrow, circle etc). These types are defined in the enumeration MarkerType (found in the Enums interface).

5.7.2 Label Class

This class makes it easy to draw text labels at specific points on the study.

6 Signals

All studies and strategies may generate signals. Signals are events that occur at points of interest in a study. Often signals are used as indicators of buy or sell points.

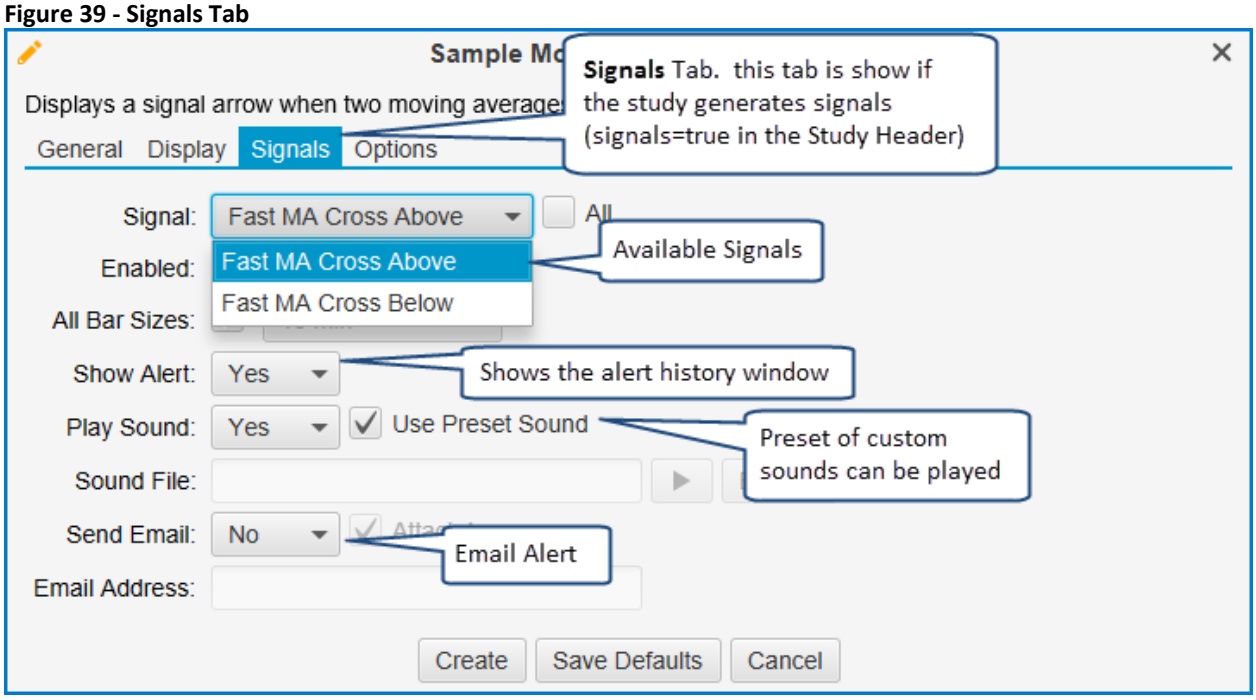
The end user may configure the study to create alerts from the signals generated by the study. To provide a high level of flexibility, the user may choose which signals they want alerts for and how the alerts behave.

The Sample Moving Average Cross (see sample project) is one example of a study that generates signals. This study generates two signals:

1. **Fast MA Crossed Above** – This occurs when the Fast MA (shorter period) crosses above the Slow MA
2. **Fast MA Crossed Below** – This occurs when the Fast MA crosses below the Slow MA

By default, these signals do not do anything other than show an up or down marker where the crosses occur on the plot. The user can configure alerts for these signals from the 'Signals' tab of the Study Dialog.

Figure 39 - Signals Tab



The following steps are required to generate signals for a study:

1. *signal* tag – set the 'signal' property in the StudyHeader to true
2. declare signals – There are two signals, cross above and cross below.
3. call 'signal' method – this generates the signals.

Figure 40 - signal tag (StudyHeader)

```
package study_examples;

/** Moving Average Cross. This study consists of two moving averages:
    Fast MA (shorter period), Slow MA. Signals are generated when the
    Fast MA moves above or below the Slow MA. Markers are also displayed
    where these crosses occur. */
@StudyHeader(
  namespace="com.mycompany",
  id="MACROSS",
  name="Sample Moving Average Cross",
  label="MA Cross",
  desc="Displays a signal arrow when two moving averages (fast and slow) cros
  menu="Examples",
1 overlay=true,
  signals=true)
public class SampleMACross extends Study
{
  enum Values { FAST_MA, SLOW_MA };
  enum Signals { CROSS_ABOVE, CROSS_BELOW };

  @Override
  public void initialize(Defaults defaults)
  {
    // User Settings
    SettingsDescriptor sd=new SettingsDescriptor();
    setSettingsDescriptor(sd);
      . . .
    // Runtime Settings
    RuntimeDescriptor desc=new RuntimeDescriptor();
    setRuntimeDescriptor(desc);
      . . .
    // Signals
2   desc.declareSignal(Signals.CROSS_ABOVE, "Fast MA Cross Above");
    desc.declareSignal(Signals.CROSS_BELOW, "Fast MA Cross Below");

    desc.setRangeKeys(Values.FAST_MA, Values.SLOW_MA);
  }
```

signals property must be set to **true**

Declare each type of signal.

Figure 41 - Generating Signals

```
@Override
protected void calculate(int index, DataContext ctx)
{
  int fastPeriod=getSettings().getInteger(Inputs.PERIOD);
  int slowPeriod=getSettings().getInteger(Inputs.PERIOD2);
  if (index < Math.max(fastPeriod, slowPeriod)) return; // not enough data

  DataSeries series=ctx.getDataSeries();

  // Calculate and store the fast and slow MAs
  Double fastMA=series.ma(getSettings().getMAMethod(Inputs.METHOD), index, fastPeriod, getSett
  Double slowMA=series.ma(getSettings().getMAMethod(Inputs.METHOD2), index, slowPeriod, getSet
  if (fastMA == null || slowMA == null) return;

  series.setDouble(index, Values.FAST_MA, fastMA);
  series.setDouble(index, Values.SLOW_MA, slowMA);

  if (!series.isBarComplete(ind

  // Check to see if a cross oc
  Coordinate c=new Coordinate(series.getStartTime(index), slowMA);
  if (crossedAbove(series, index, Values.FAST_MA, Values.SLOW_MA)) {
    MarkerInfo marker=getSettings().getMarker(Inputs.UP_MARKER);
    if (marker.isEnabled()) addFigure(new Marker(c, Enums.Position.BOTTOM, marker));
    ctx.signal(index, Signals.CROSS_ABOVE, "Fast MA Crossed Above!", series.getClose(index));
  }
  else if (crossedBelow                    FAST_MA, Values.SLOW_MA)) {
    MarkerInfo marker=g                    Inputs.DOWN_MARKER);
    if (marker.isEnabled())                er(c, Enums.Position.TOP, marker));
    ctx.signal(index, Signals.CROSS_BELOW, "Fast MA Crossed Below!", series.getClose(index));
  }

  series.setComplete(index);
}
```

crossedAbove(...) and crossedBelow(...) are convenience methods for determining if two paths have crossed

Generate signals. Note: these are only triggered when the last bar is completed

7 Tick Data

Version 5 of MotiveWave™ includes support for handling live and historical tick data (if supported by the broker and/or data service). Live and historical ticks are described using the **Tick** interface (see below). The time of the tick can be accessed via the getTime() method. This returns the number of milliseconds since January 1, 1970 (epoch time).

Figure 42 Tick Interface

```
package com.motivewave.platform.sdk.common;

/** Represents a tick (trade) that occurred. */
public interface Tick
{
  /** Gets the trade price for the tick.
   @return trade price for the tick*/
  float getPrice();
  /** Gets the number of units traded.
   @return number of units traded */
  int getVolume();
  /** Gets the ask(offer) price when this tick occurred.
   @return ask price when the tick occurred. */
  float getAskPrice();
  /** Gets the ask size (number of units offered) when this tick occurred.
  @return ask size when the tick occurred. */
  int getAskSize();
  /** Gets the bid price when this tick occurred.
  @return bid price when the tick occurred. */
  float getBidPrice();
  /** Gets the ask size (number of units bid) when this tick occurred.
  @return bid size when the tick occurred. */
  int getBidSize();
  /** Gets the time when this tick occurred (in milliseconds since 1970).
  @return the time when this tick occurred (in milliseconds since 1970). */
  long getTime();
  /** Indicates if this trade occurred at the ask price (bid price if false).
   @return true if this trade occurred at the ask price */
  boolean isAskTick();
}
```

Historical ticks can be requested at any time from the Instrument interface using the getTicks(startTime, endTime) method. This will return a list of ticks that occurred on the instrument between the start and end times.

Figure 43 Instrument getTicks()

```
package com.motivewave.platform.sdk.common;

import java.util.List;

/** Represents an Instrument. */
public interface Instrument
{

  /** Gets the ticks that occurred between the given startTime and endTime
      @param startTime - start time (in milliseconds since 1970)
      @param endTime - end time (in milliseconds since 1970)
      @return list of ticks that occurred in the given time */
  List<Tick> getTicks(long startTime, long endTime);
}
```

Live ticks can be processed through the onTick(dataContext, tick) method in the Study class. This method will be called every time a new tick is generated on the instrument.

Figure 44 Study onTick Method

```
package com.motivewave.platform.sdk.study;

import java.beans.PropertyChangeListener;

/** This is the base class for all studies and strategies. */
public class Study implements Cloneable
{

  /** This method is called when a tick (trade) occurs.
     @param tick latest tick (trade)  */
  public void onTick(DataContext ctx, Tick tick)
  {
  }

}
```

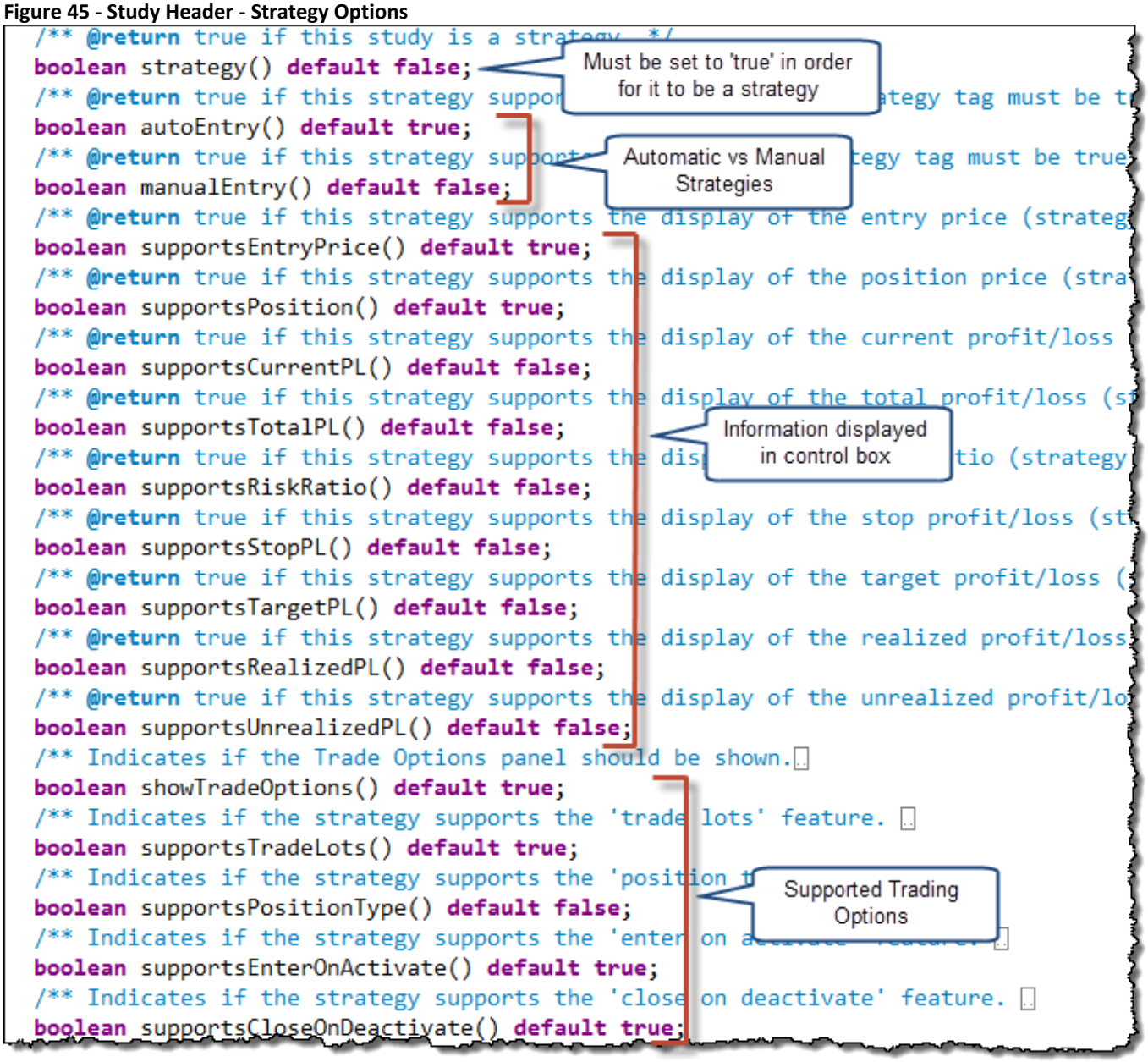
8 Strategies

Strategies allow you to automate (or partially automate) the buying and selling of instruments. The strategy APIs build upon the study classes and interfaces described in the preceding sections.

8.1 StudyHeader

Let's start buy looking at what is needed in the StudyHeader to declare a strategy:

Figure 45 - Study Header - Strategy Options




```
/** @return true if this study is a strategy */
boolean strategy() default false;
/** @return true if this strategy suppor                    ategy tag must be t
boolean autoEntry() default true;
/** @return true if this strategy suppor                    egy tag must be true
boolean manualEntry() default false;
/** @return true if this strategy supports the display of the entry price (strateg
boolean supportsEntryPrice() default true;
/** @return true if this strategy supports the display of the position price (stra
boolean supportsPosition() default true;
/** @return true if this strategy supports the display of the current profit/loss
boolean supportsCurrentPL() default false;
/** @return true if this strategy supports the display of the total profit/loss (s
boolean supportsTotalPL() default false;
/** @return true if this strategy supports the dis                    tio (strategy
boolean supportsRiskRatio() default false;
/** @return true if this strategy supports the display of the stop profit/loss (st
boolean supportsStopPL() default false;
/** @return true if this strategy supports the display of the target profit/loss (
boolean supportsTargetPL() default false;
/** @return true if this strategy supports the display of the realized profit/loss
boolean supportsRealizedPL() default false;
/** @return true if this strategy supports the display of the unrealized profit/lo
boolean supportsUnrealizedPL() default false;
/** Indicates if the Trade Options panel should be shown.
boolean showTradeOptions() default true;
/** Indicates if the strategy supports the 'trade lots' feature.
boolean supportsTradeLots() default true;
/** Indicates if the strategy supports the 'position t
boolean supportsPositionType() default false;
/** Indicates if the strategy supports the 'enter on a                    feature.
boolean supportsEnterOnActivate() default true;
/** Indicates if the strategy supports the 'close on deactivate' feature.
boolean supportsCloseOnDeactivate() default true;
```

The most important property to have set is "**strategy=true**". The "autoEntry" and "manualEntry" properties may be used to indicate that the strategy is automatic or manual (Note: Trade Manager is an example of a manual entry strategy).

8.2 Study Class

There are a number of other methods available on the *Study* class that may be used for strategies. The following excerpt from the Study class illustrates the strategy event methods:

Figure 46 - Strategy Events

```
//
// Strategy Methods
//
/** This method is called when the strategy is activated. */
public void onActivate(OrderContext ctx) {}
/** This method is called when the current bar is first opened.*/
public void onBarOpen(OrderContext ctx) {}
/** This method is called when the current bar has been updated. */
public void onBarUpdate(OrderContext ctx) {}
/** This method is called when the current bar has been closed.*/
public void onBarClose(OrderContext ctx) {}
/** This method is called when a signal is generated by the study. T
public void onSignal(OrderContext ctx, Object signalKey){}
/** This method is called when the strategy is deactivated. */
public void onDeactivate(OrderContext ctx) {}
/** This method is called when the strategy is reset. */
public void onReset(OrderContext ctx) {}
/** This method is called when the current open position is closed.
public void onPositionClosed(OrderContext ctx) {}
/** This method is called on response to the 'E          tton on t
public void onEnterNow(OrderContext ctx) {}
```

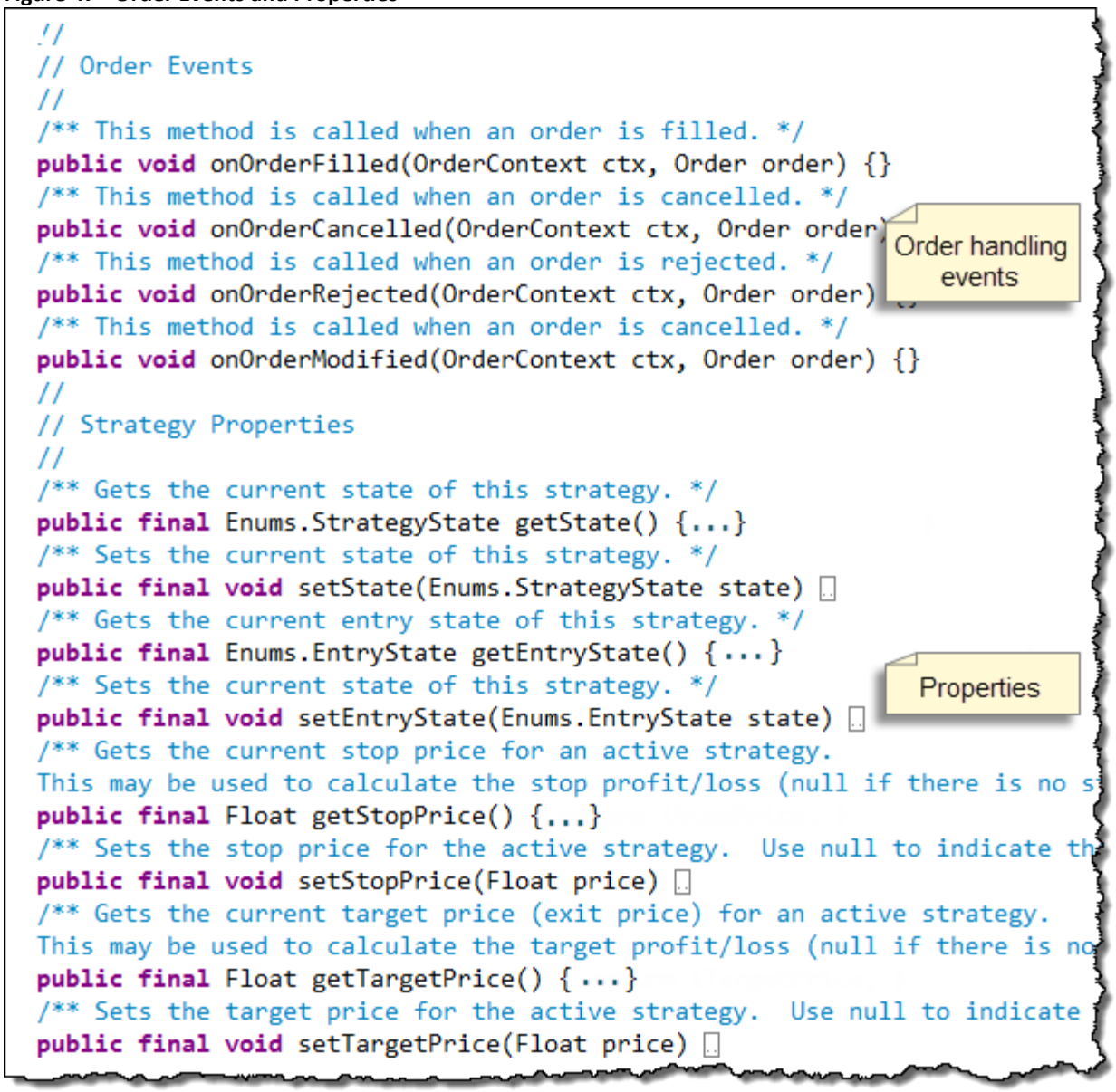
Manual Entry Method Only

- **onActivate(OrderContext ctx)** – This method is called when the user presses the 'Activate' button in the Control Box. If the user has chosen the 'Enter on Activate' option this method should create an entry order for the appropriate direction.
- **onBarOpen(OrderContext ctx)** – This method is called when the price bar is first opened. Note: live bar updates must be enabled for this method to be called.
- **onBarUpdate(OrderContext ctx)** – This method is called when the current price bar is updated. Note: live bar updates must be enabled for this method to be called.
- **onBarClose(OrderContext ctx)** – This method is called when the current price bar is closed (just before the next price bar is opened).
- **onSignal(OrderContext ctx, Object signal)** – This method is called when a signal is raised by a study. This is a convenient method to override if your strategy is based on signals from an existing study (see Sample MA Cross Strategy).
- **onDeactivate(OrderContext ctx)** – Called when the user presses the 'Deactivate' button. By default this method will close the open position (if enabled by the user).
- **onReset(OrderContext ctx)** – This is called when the user presses the 'reset' button on the control box.
- **onPositionClosed(OrderContext ctx)** – Called when an open position is closed.

- **onEnterNow(OrderContext ctx)** – Called when the user presses the 'Enter Now' button on the Control Box. Note: this is only applicable for manual strategies.

In addition to the events described above, there are also a set of methods for handling orders and a set of properties available to strategies. For a full list of available methods, please consult the API documentation.

Figure 47 - Order Events and Properties



8.3 OrderContext Interface

The *OrderContext* interface is passed to most of the strategy events and provides functionality for managing orders and positions. This interface also manages the current position state for the strategy and provides methods for getting the unrealized profit/loss, average entry price etc. A number of convenience methods also exist such as:

- **buy(int qty)** – places a market order to buy the given quantity and waits for the order to be filled.
- **sell(int qty)** – places a market order to sell the given quantity and waits for the order to be filled.
- **closeAtMarket()** – closes the current position at market price.

Figure 48 - OrderContext Interface

```
package com.motivewave.platform.sdk.order_mgmt;
/** This interface provides the capability to create and manage orders. */
public interface OrderContext
{
  /** Gets the data context associated with this strategy.  This provides
  DataContext getDataContext();
  /** Gets the instrument associated with the data provided in this contex
  Instrument getInstrument();
  /** Convenience Method: Places a BUY order for the current instrument at
  void buy(int qty);
  /** Convenience Method: Places a SELL order for the current instrument a
  void sell(int qty);
  /** Closes the position held by this strategy. This method will wait unt
  void closeAtMarket();
  /** Gets all of the active orders that are associated with this strategy
  List<Order> getActiveOrders();
  /** Gets the current open position.  A negative number is returned if th
  int getPosition();
  /** Gets the average entry price for the current position. */
  float getAvgEntryPrice();
  /** Gets the total realized pnl since this strategy was opened (or last
  double getTotalRealizedPnL();
  /** Gets the realized PnL for the current 'leg' of the strategy. */
  double getRealizedPnL();
  /** Gets the PnL for the open position.  This value will change with eve
  double getUnrealizedPnL();
  /** Convenience Method.  Calculates the current profit/loss from the giv
  double calcPnl(float entryPrice, int qty);
  /** Convenience Method.  Calculates the profit/loss from the given entry
  double calcPnl(float entryPrice, float exitPrice, int qty);
  /** Converts the given amount to the amount in the base currency using t
  double convertToBaseCurrency(double pnl);
```

The following methods may be used to manually create and manage stop, limit and market orders:

Figure 49 - OrderContext Order Mgmt Methods

```
/** Creates a new 'Market' order. */
Order createMarketOrder(Enums.OrderAction action, int qty);
/** Creates a new 'Market' order. */
Order createMarketOrder(Instrument instr, Enums.OrderAction action, int qty);
/** Creates a new 'Limit' order. */
Order createLimitOrder(Enums.OrderAction action, Enums.TIF tif, int qty, float limitPrice);
/** Creates a new 'Limit' order. */
Order createLimitOrder(Instrument instr, Enums.OrderAction action, Enums.TIF tif, int qty, fl
/** Creates a new 'Stop' order. */
Order createStopOrder(Enums.OrderAction action, Enums.TIF tif, int qty, float stopPrice);
/** Creates a new 'Stop' order. */
Order createStopOrder(Instrument instr, Enums.OrderAction action, Enums.TIF tif, int qty, flo
/** Use this method to submit one or more orders to the broker.
  If one or more of the orders are new, the orders will be created otherwise the
  existing order will be modified. Please note: this is a synchronous call and
  may take a significant amount of time to return.*/
void submitOrders(Order... orders);
/** Use this method to submit one or more orders to the broker.
If one or more of the orders are new, the orders will be created otherwise the
existing order will be modified. Please note: this is a synchronous call and
may take a significant amount of time to return.*/
void submitOrders(List<Order> orders);
/** Use this method to cancel one or more existing orders. Please note: this is a synchronous
  may take a significant amount of time to return. */
void cancelOrders(Order... orders);
/** Use this method to cancel one or more existing orders. Please note: this is a synchronous
may take a significant amount of time to return. */
void cancelOrders(List<Order> orders);
/** Cancels all of the open orders for this strategy. */
void cancelOrders();
```

8.4 Order Interface

Strategies that simply buy and sell positions using the buy/sell methods will not have to deal with orders directly.

Market Orders vs Stop/Limit Orders
It can be very tempting to use stop and/or limit orders in place of market orders when implementing a strategy since these types of orders are already placed at the exchange and they can help guarantee execution at a particular price. There are however several behaviors to be aware of when using these types of orders especially with fully automated strategies: • Limit Orders are not guaranteed to be executed. Even if the price action has traded through your limit price, it may not have been executed in a live environment if there was not enough demand to fill your order at the specified price. • Stop Orders are often triggered on Bid/Ask. It's a common misconception that stop orders are triggered by last price. Most (if not all) brokers trigger stop orders using the bid or ask price (depending on whether it's a buy or sell). This can cause your stop order

to be executed unexpectedly early especially if there is a significant spread in the bid/ask prices.

- Stop Orders are filled at market. Once a stop order is triggered, it is filled at market price. Stop Limit orders do exist, but are not currently supported by this API. Also note that not all brokers support Stop Limit orders.

If you choose to implement a fully automated strategy using non-market orders, you will need to consider these behaviors and add the appropriate code to handle cases where your orders do not get filled, or do not get filled at your expected price.

Ultimately, the choice you make will be a trade-off between order executions vs. fill price.

The following diagram illustrates some of the methods available in the *Order* interface. For a full list of methods, consult the API documentation.

Figure 50 - Order Interface

```
package com.motivewave.platform.sdk.order_mgmt;
/** Represents an order to buy or sell an instrument. */
public interface Order
{
  /** Gets the account ID for this order. @return account ID for this
  String getAccountId();
  /** @return the unique identifier for this order. Note: on some brok
  String getOrderId();
  /** @return the instrument of this order. */
  Instrument getInstrument();
  /** @return the type of this order (Stop, Limit etc) */
  Enums.OrderType getType();
  /** Gets the action of this order (Buy or Sell) */
  Enums.OrderAction getAction();
  /** Gets the limit price for the order (null if not a limit order).
  Float getLimitPrice();
  /** Gets the stop price for the order (null if not a stop order). */
  Float getStopPrice();
  /** Gets the Time In Force for this order. */
  Enums.TIF getTIF();
  /** @return the size of this order (ie number of shares, contracts e
  int getQuantity();
  /** Gets the average fill price for this order.
   * @return the average fill price for this order. */
  float getAvgFillPrice();
  /** @return the last price that this order was filled at. */
  float getLastFillPrice();
  /** @return the number of shares/contracts etc that have been filled
  int getFilled();
  /** Gets the time (in millis) of the last fill on this order. */
  long getLastFillTime();
```

8.5 Trading Sessions

Version 1.1 of the SDK introduces the ability for the user to define trading sessions for a strategy.

A 'trading session' is simply a valid time period during the day in which trading is allowed for the strategy. By default, all strategies support up to 2 trading sessions. This behavior can be modified in the *StudyHeader*:

Figure 51 StudyHeader trading session options

```
package com.motivewave.platform.sdk.study;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface StudyHeader
{
  /** Namespace for this study (Must be unique for your organization) */
  String namespace();
  /** @return true if this study should be protected by namespace. */
  boolean secured() default false;
  /** Unique (within the namespace) ID for this study. */
  String id();
  /** Resource bundle to pull translatable strings from. */
  String rb() default "";
    ...
  /** @return true if this study is a strategy. */
  boolean strategy() default false;
  /** Indicates if the strategy supports sessions.
   * @return true if this strategy supports
  boolean supportsSessions() default true;
  /** Indicates if the number of sessions supported by this strategy (default 2).
   * @return number of supported sessions. */
  int sessions() default 2;
  /** Indicates if the strategy supports the 'enter on
   * @return true if this strategy supports the 'enter
  boolean supportsEnterOnSessionStart() default false;
  /** Indicates if the strategy supports the 'exit on session close' feature.
   * @return true if this strategy supports the 'exit
  boolean supportsCloseOnSessionEnd() default true;
```

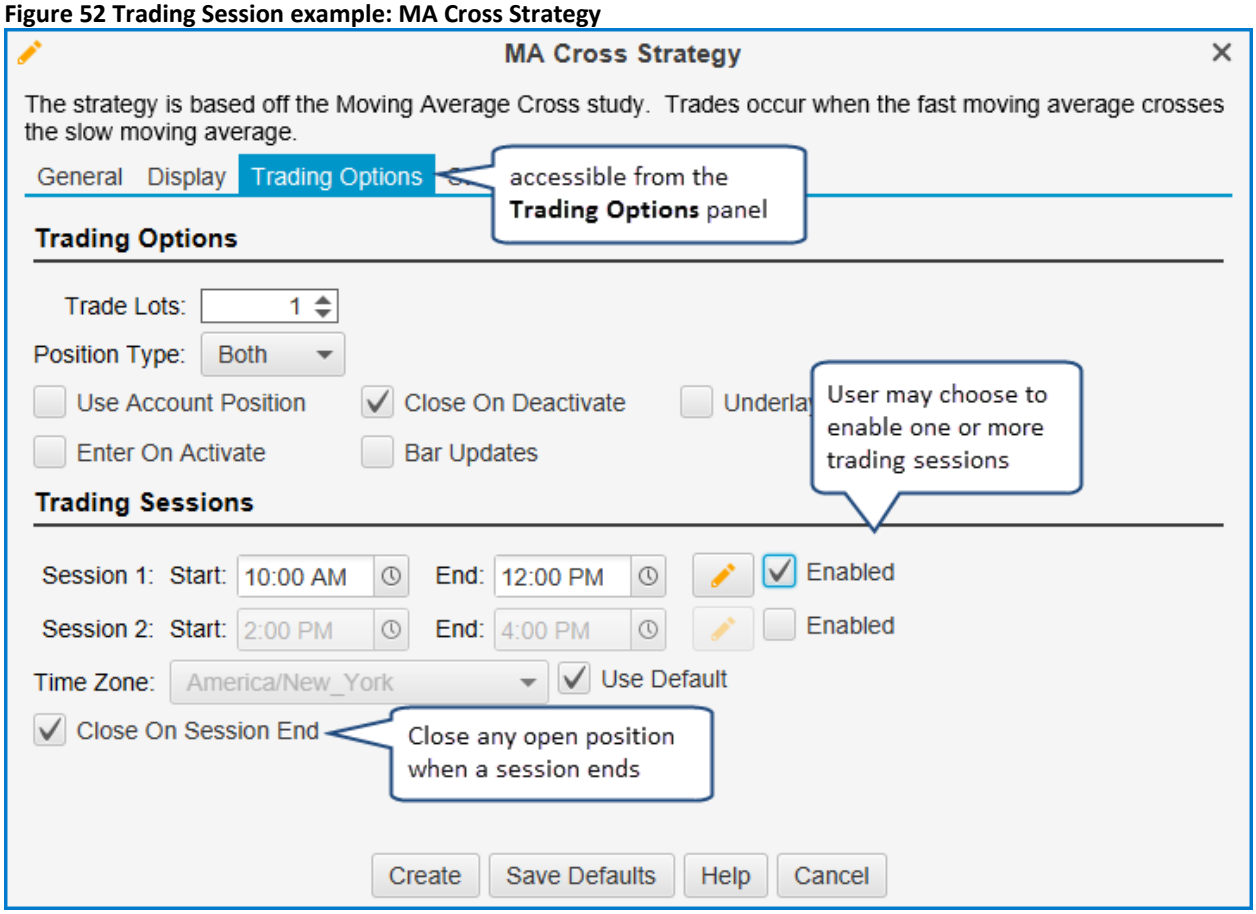
set to false to disable trading sessions

allows the user to choose to automatically enter when a session starts

allows the user to choose to automatically close an open position when a session ends

The following screen shot illustrates an example of the MA Cross Strategy with the default settings for Trading Sessions.

Figure 52 Trading Session example: MA Cross Strategy



8.5.1 Runtime Support

The following additional methods have been added to the *Settings* class to access information chosen by the user at runtime within the strategy:

```
package com.motivewave.platform.sdk.common;

public class Settings implements Cloneable
{
  ...
  /** Gets the trading sessions (strategies only). */
  public List<TimeFrame> getSessions()
  /** @return true if this strategy should enter automatically when a trading
  public boolean isEnterOnSessionStart()
  /** @return true if this strategy should exit an open position automatically
  public boolean isCloseOnSessionEnd()
  /** Gets the timezone for sessions (null for local time zone). */
  public TimeZone getTimeZone()
```

The following methods are also available on the Study class that may be optionally overridden. Note: if 'enter on session start' is enabled the strategy must override and implement the onSessionStart(...) method to implement the entry logic.

```
package com.motivewave.platform.sdk.study;
public class Study implements Cloneable
{
  ...
  /** This method is called when a trading session is started. */
  public void onSessionStarted(OrderContext ctx, TimeFrame session)
  /** This method is called when a trading session is ended. */
  public void onSessionEnded(OrderContext ctx, TimeFrame session)
  ...
```

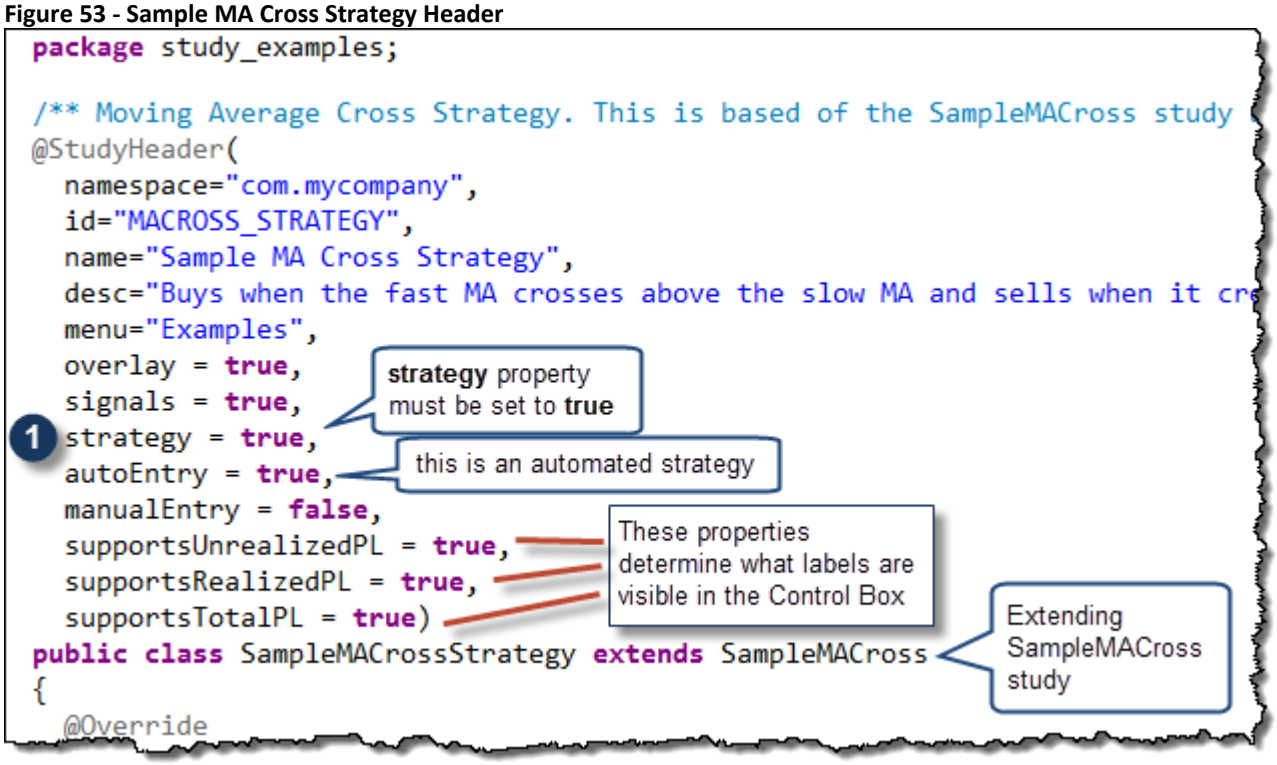
8.6 Sample MA Cross Strategy

The following example illustrates a simple strategy based on the SampleMACross study (see sample project and signals in Section 6). This strategy will buy when the fast moving average crosses above the slow moving average and sell when it crosses below.

For convenience, this strategy will subclass the SampleMACross study and rely on the signals generated for 'Fast MA Crossed Above' (Signals.CROSS_ABOVE) and 'Fast MA Crossed Below' (Signals.CROSS_BELOW).

Let's take a look at the StudyHeader. The key properties to note here are: **strategy=true** and **autoEntry=true** (1 below).

Figure 53 - Sample MA Cross Strategy Header



For this strategy, we are going to override two methods:

- **onActivate(OrderContext ctx)** – If the user chooses to open a position on activate (see Trading Options panel), we will open a long or short position depending on whether the fast MA is above or below the slow MA (see 2 below)
- **onSignal(OrderContext ctx, Object signal)** – In this method, we will use the signals generated in the SampleMACross class under the keys: Signals.CROSS_ABOVE and Signals.CROSS_BELOW (see calculate method). Note: we are reversing a position if it is open. IE: a long position becomes a short position and vice versa. (3 & 4 below)

```java
@Override
public void onActivate(OrderContext ctx)
{
  if (getSettings().isEnterOnActivate()) {
    DataSeries series = ctx.getDataContext().getDataSeries();
    int ind = series.isLastBarComplete() ? series.size()-1 : series.size()-2;
    Double fastMA = series.getDouble(ind, Values.FAST_MA);
    Double slowMA = series.getDouble(ind, Values.SLOW_MA);
    if (fastMA == null || slowMA == null) return;
    int tradeLots = getSettings().getTradeLots();
    int qty = tradeLots *= ctx.getInstrument().getDefaultQuantity();
    // Create a long or short position if we are above or below the signal line
    if (fastMA > slowMA) ctx.buy(qty);
    else ctx.sell(qty);
  }
}

@Override
public void onSignal(OrderContext ctx, Object signal)
{
  Instrument instr = ctx.getInstrument();
  int position = ctx.getPosition();
  int qty = (getSettings().getTradeLots() * instr.getDefaultQuantity());

  qty += Math.abs(position); // Stop and Reverse if there is an open position
  if (position <= 0 && signal == Signals.CROSS_ABOVE) {
    ctx.buy(qty); // Open Long Position
  }
  if (position >= 0 && signal == Signals.CROSS_BELOW) {
    ctx.sell(qty); // Open Short Position
  }
}
```

2 Open the initial position (if the user chose 'Enter On Activate')

8.7 Strategy States

A strategy can be in one of three different states (defined in *Enums.StrategyState*):

1. **Inactive** – No trades are active and the strategy will not place any trades.
2. **Active** – The strategy may place trades to open or close positions
3. **Dormant** – In this state, the strategy is still active but does not place any new trades

The current state of the strategy can be queried/set from the following methods (on the Study Class):

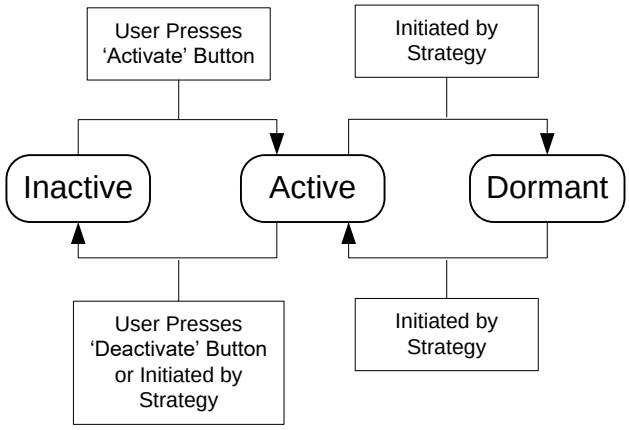
1. **getState()** – returns the current state of the strategy
2. **setState(Enums.StrategyState state)** – Sets the new state for the strategy.

In most cases, the strategy state is initiated by the user by pressing the 'Activate' or 'Deactivate' button from the Strategy Control Box. However, you can set the state from your strategy. This is most common when switching the strategy to the 'Dormant' state. You may want to use this state to indicate that the

strategy is waiting for a specify condition to happen before placing trades again. This is often used when you just want the strategy to be active during specific hours of the day.

The following diagram illustrates these states and the transitions between them:

Figure 54 - Strategy States



8.8 Manual Strategies

MotiveWave™ allows you to create strategies that respond to user input to enter or exit a position. This can be very useful as a way to help direct and manage exit points for user initiated trades. For an example of how this works, see the Trade Manager strategy.

The following screen shots illustrate the Trade Manager strategy in action:

Figure 55 - Trade Manager

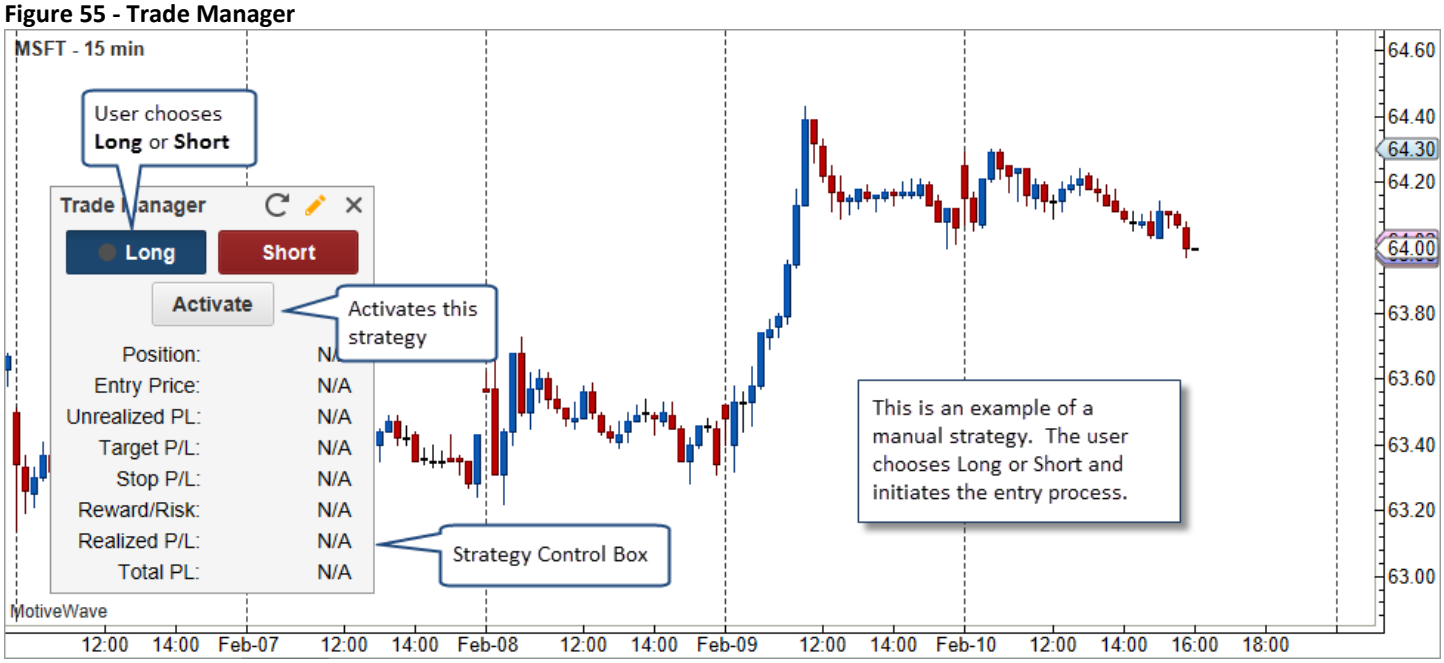


Figure 56 - Trade Manager - Entering a Position

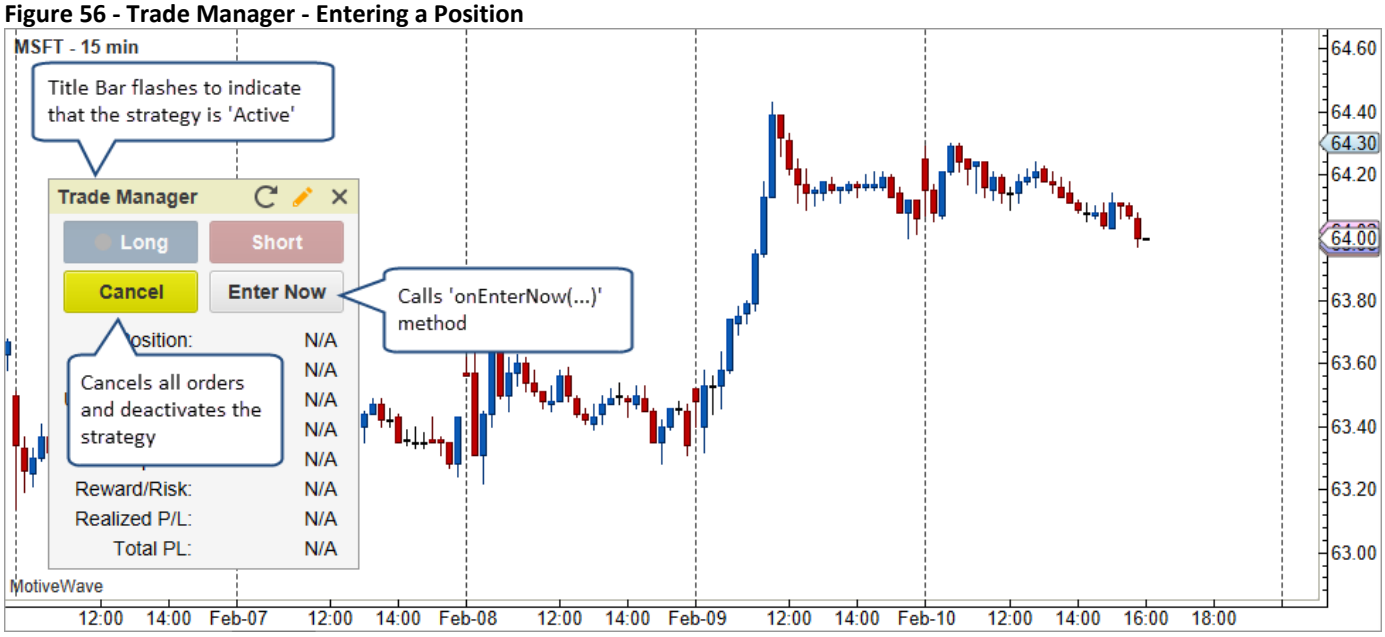


Figure 57 - Trade Manager Open Position

8.8.1 Entry States

In order to manage the orders for manual strategies, entry states have been defined to indicate the current stage the strategy is in. These states are defined in the *Enums.EntryState* enumeration.

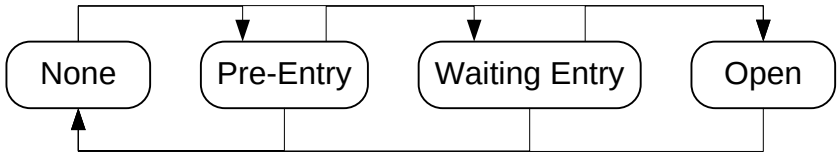
1. **None** – No entry state, waiting for the user to initiate the entry process
2. **Pre-Entry** – The user has initiated the entry process and the strategy is preparing to create the entry order(s).
3. **Waiting Entry** – Waiting for entry orders to be filled (this state can be skipped if using market orders).
4. **Open** – Position is open, waiting for the position to be closed.

These states can be queried/set from the following methods in the Study Class:

1. **getEntryState()** – returns the current entry state for the strategy.
2. **setEntryState(Enums.EntryState state)** – sets the entry state for the strategy.

The following diagram illustrates these states and their transitions:

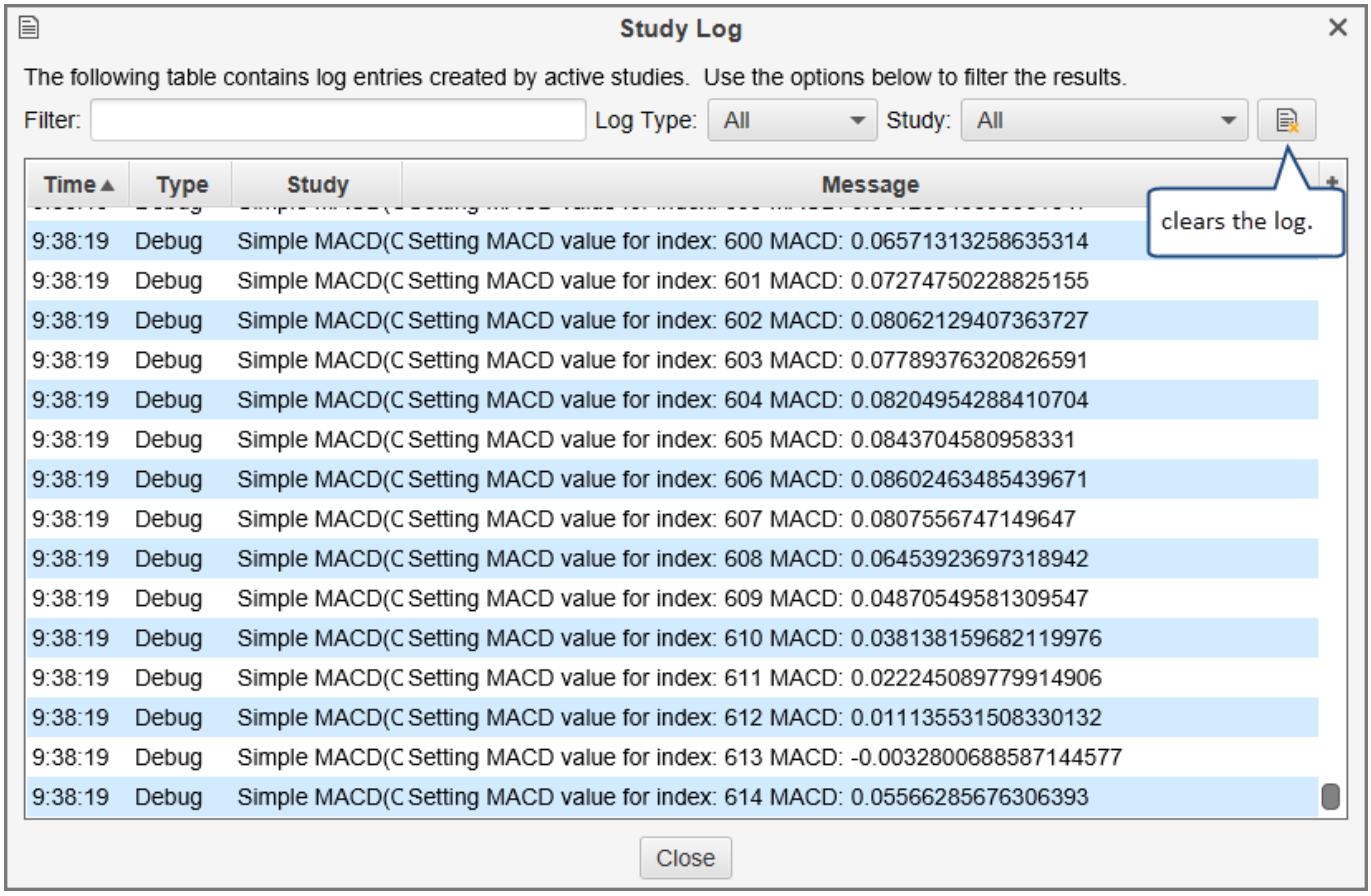
Figure 58 - Entry States



9 Logging

Often as part of debugging, you will want to write information to a log. MotiveWave™ includes a study log utility. This can be accessed from the Console menu bar: *View -> Display -> Study Log*.

The following diagram illustrates what the Study Log looks like:



There are 4 methods available (from the base class Study) for creating log entries:

Figure 59 - Study logging methods

```
package com.motivewave.platform.sdk.study;

/** This is the base class for all studies and strategies. */
public class Study
{
   . . .
  //
  // Log methods
  //
  /** Logs a debug message to the study log. */
  public void debug(String msg)

  /** Logs an info message to the study log. */
  public void info(String msg)

  /** Logs a warning message to the study log. */
  public void warning(String msg)

  /** Logs an error message to the study log. */
  public void error(String msg)
```

10 Internationalization

For simplicity, the examples provided so far in this guide have translatable text embedded in the code directly. Although MotiveWave™ does not currently support multiple languages (at least as of writing this guide), it is inevitable that this will happen at some point in the near future.

All of the studies available in MotiveWave™ have the translatable text separated into a *Resource Bundle*. Resource Bundles are a standard mechanism built into Java for internationalization. If you are unfamiliar with this construct, there are many tutorials available on the internet. Here is a general tutorial available on the Oracle™ website:
http://docs.oracle.com/javase/tutorial/i18n/resbundle/concept.html

Separating text in a study is very simple and only requires you to do the following:

1. Declare the Resource Bundle – In the Study Header, specify the package and name of the resource bundle using the ***rb*** property
2. Use the ***get("LABEL_ID", …)*** to retrieve text. This method available from the *Study* class pulls text from the resource bundle associated with the given ID. Values in the text can be replaced by specifying these values after the label ID (named %1, %2, %3…)

10.1 Example: MACD

The following example shows the Study Header for the MACD study. In this case the **rb** property is pointing to the resource bundle: **com.motivewave.platform.study.nls.strings**. This will resolve to the *strings.properties* file (for English translation) in the com*/motivewave/platform/study/nls* directory.

Once the rb property is defined in the Study Header, MotiveWave™ will assume that the other properties (that expect displayable text) are actually IDs that need to be resolved from the resource bundle.

Figure 60 - Internationalization Study Header

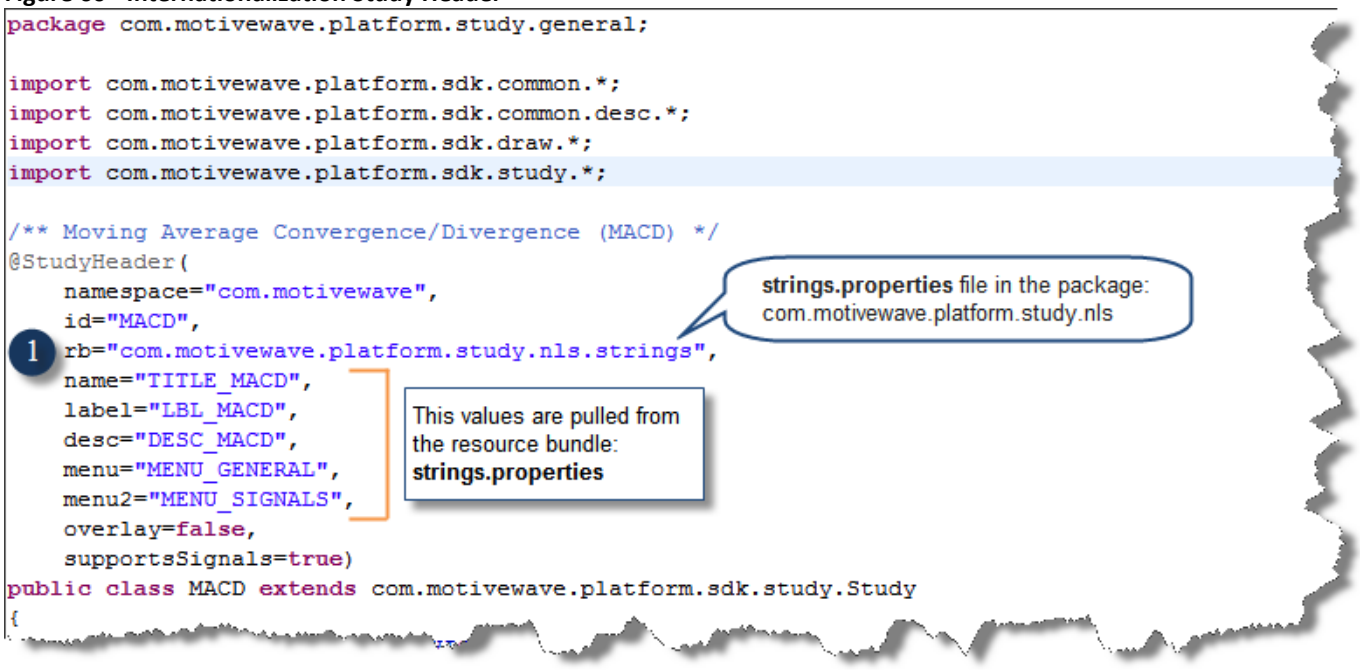


Figure 61 - Resolving text using the 'get' method

```
@Override
public void initialize(Defaults defaults)
{
  SettingsDescriptor sd = new SettingsDescriptor();
  setSettingsDescriptor(sd);
2 SettingTab tab = new SettingTab(get("TAB_GENERAL"));
  sd.addTab(tab);

  SettingGroup inputs = new SettingGroup(get("LBL_INPUTS"));
  inputs.addRow(new InputDescriptor(Inputs.INPUT, get("LBL_INPUT"), Enums.BarInput.CLOSE));
  inputs.addRow(new MAMethodDescriptor(Inputs.METHOD, get("LBL_METHOD"), Enums.MAMethod.EMA));
  inputs.addRow(new MAMethodDescriptor(Inputs.SIGNAL_METHOD, get("LBL_SIGNAL_METHOD"), Enums.MAMethod.SMA));
  inputs.addRow(new IntegerDescriptor(Inputs.PERIOD, get("LBL_PERIOD1"), 12, 1, 9999, 1));
  inputs.addRow(new IntegerDescriptor(Inputs.PERIOD2, get("LBL_PERIOD2"), 26, 1, 9999, 1));
  inputs.addRow(new IntegerDescriptor(Inputs.SIGNAL_PERIOD, get("LBL_SIGNAL_PERIOD"), 9, 1, 9999, 1));
  tab.addGroup(inputs);

  tab = new SettingTab(get("TAB_DISPLAY"));
  sd.addTab(tab);
```

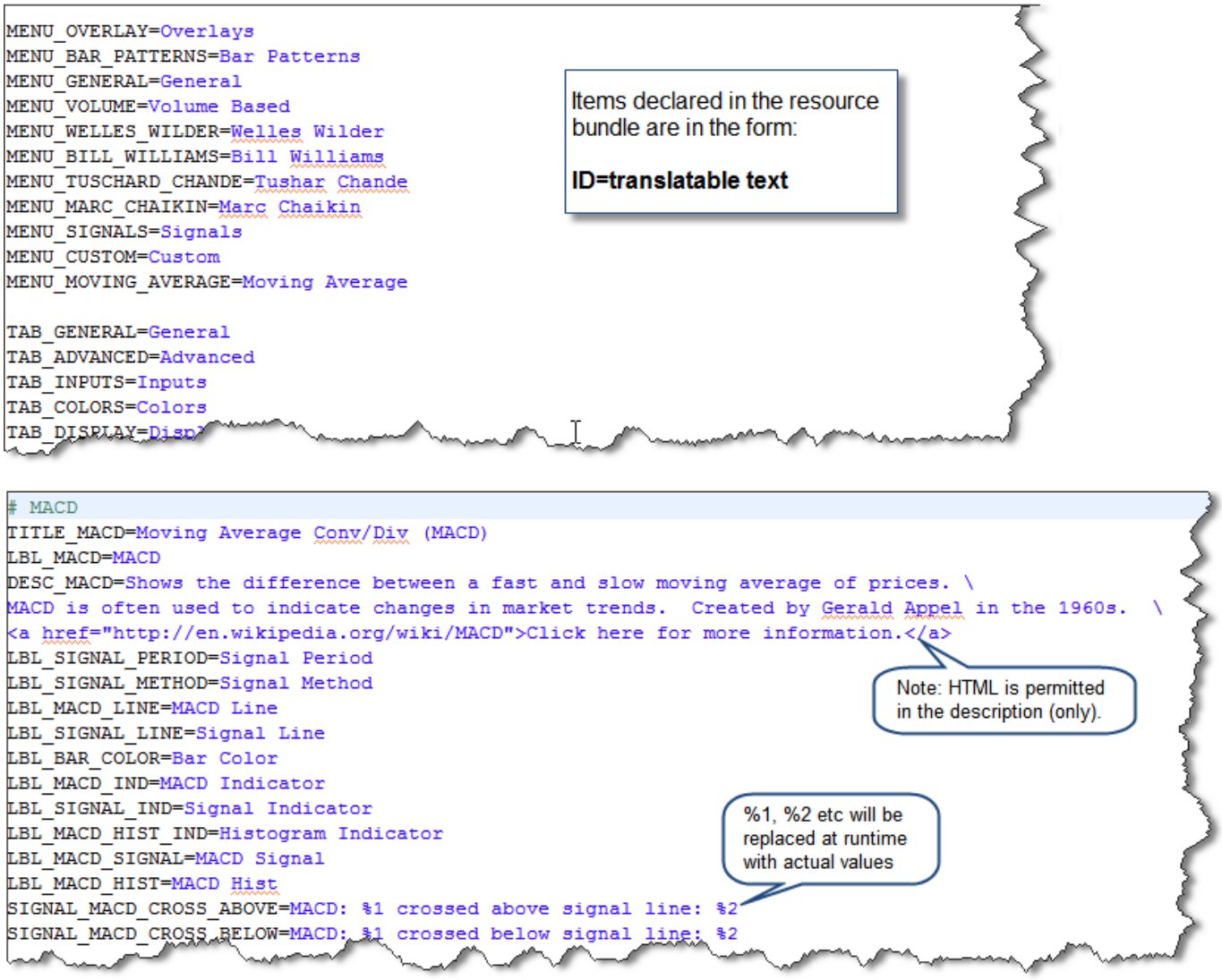
Use the 'get' method to retrieve text from **strings.properties**

Figure 62 - Resolving text using the get method with parameters

```
    if (pMACD <= pSignal && MACD > signal) {
      MarkerInfo marker = getSettings().getMarker(Inputs.UP_MARKER);
      if (marker.isEnabled() && !latest) {
        addFigure(new Marker(c, Enums.Position.BOTTOM, marker));
      }
3     ctx.signal(index, Signals.CROSS_ABOVE, get("SIGNAL_MACD_CROSS_ABOVE", MACD, signal), signal);
    }
    else if (pMACD >= pSignal && MACD < signal) {
      MarkerInfo marker = getSettings().getMarker(Inputs.DOWN_MARKER);
      if (marker.isEnabled() && !latest) {
        addFigure(new Marker(c, Enums.Position.TOP, marker));
      }
      ctx.signal(index, Signals.CROSS_BELOW, get("SIGNAL_MACD_CROSS_BELOW", MACD, signal), signal);
    }
```

get method using parameter replacements (%1, %2 in **strings.properties**)

Figure 63 - strings.properties file



11 Deployment

The process of installing your extensions in MotiveWave™ is referred to as 'Deployment'. There are essentially two use cases for deploying extensions:

1. **Development** – As you are coding your extension, you will want to deploy your changes to MotiveWave™ so you can test your changes.
2. **Distribution** – When you have completed development you will want to package your extensions and make them available to other users.

11.1 Packaging

You may distribute your extensions by providing the .class (and .properties) files directly to your customers but you may find this awkward if you have more than one.

The preferred way to distribute these files is to package them together in a Jar (Java ARchive) file. This is a standard Java mechanism for distributing Java libraries or applications. If you would like to know more about this format you can visit this website address:
http://java.sun.com/developer/Books/javaprogramming/JAR/

The sample Eclipse project includes the ability to create a Jar file for distribution in the ANT build script. You may also use the deployment features of Eclipse to create your Jar file.

11.2 Loading Extensions

MotiveWave™ will dynamically load extensions from the directory '**MotiveWave Extensions**'. This directory is created by MotiveWave™ when it first starts. Depending on the environment you have, it will be found:

1. **Windows** – C:\Documents and Settings\<username>\MotiveWave Extensions
2. **Mac OSX -** /Users/<username>/MotiveWave Extensions

This directory is searched (recursively) for the following types of files:

1. **JAR Files (.jar)** – These are essentially 'zip files' that contain .class and .properties files
2. **Class Files (.class)** – These files are generated by the javac compiler. Note: you must preserve the directory structure when copying these files into the *'MotiveWave Extensions'* directory. For example classes in the 'study_examples' package must be put in the *'MotiveWave Extensions\study_examples'* directory.
3. **Properties Files (.properties)** – These files contain the translatable text that has been separated from the code (see section on Internationalization). Similar to the class files, you must preserve the directory structure when copying these files into *'MotiveWave Extensions'* directory.

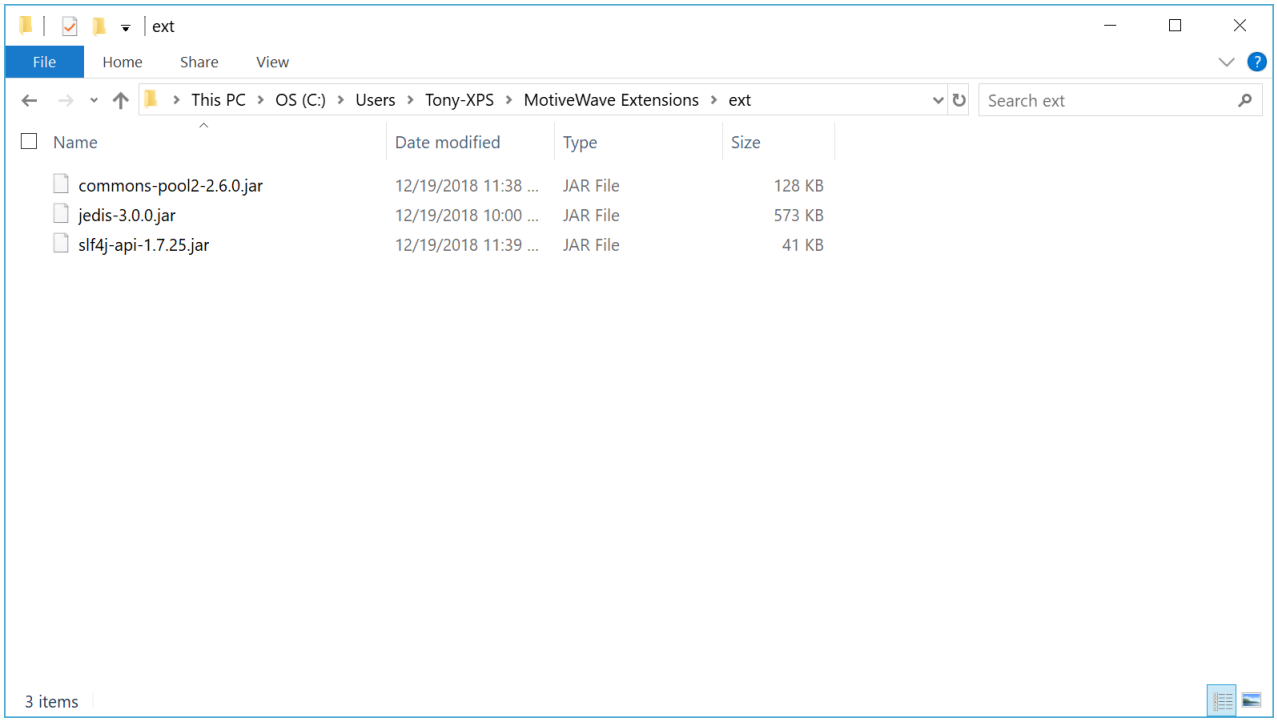
'.last_updated' File
If you look in the *'MotiveWave Extensions'* directory (Note: this is a hidden file on Mac OSX) you will see a file called '.last_updated'. MotiveWave™ uses this file to determine is any of the files in this directory have been changed since its last scan. If you want to test your changes without restarting MotiveWave™, you will need to copy your changed files to *'MotiveWave Extensions'* and then modify the timestamp on this file (for example using the Unix 'touch' command). The sample build.xml file (Apache ANT script) shows an example of how to modify this file to get

MotiveWave™ to reload extensions.

11.3 Third-Party Libraries (jars)

Starting with version 5.4.21 third-party libraries (jar files) can now be used in the SDK. These jar files must be added in the **ext** subdirectory of the **MotiveWave Extensions** directory. On startup of MotiveWave, all jar files in this subdirectory will be added to the class path. If any jar files are added or modified in this directory, MotiveWave will need to be restarted to pick up the new changes.

The screen shot below shows an example of third-party libraries added to the **ext** directory in the **Motive Wave Extensions**.



12 Environment Setup

You may use any Java™ development environment you wish to develop extensions for MotiveWave™. This section will explain how to get up and running with the Eclipse Integrated Development Environment (IDE). We have also included a sample Eclipse Project that you may use as a starting point for your own development. This sample project contains a build script (Apache ANT based) that makes it easy to deploy your changes to MotiveWave™ and package your extensions for distribution.

Eclipse (www.eclipse.org) is the most popular tool for Java development and best of all its free! There are many different environments for Java development, some of the more notable tools include:

1. **NetBeans** – This Open Source development environment is free as well and is developed by Sun (now Oracle)
2. **IntelliJ** – http://www.jetbrains.com/idea
3. **JCreator** – http://www.jcreator.com

12.1 Where do I get the SDK?

The SDK (Software Development Kit) is built directly into MotiveWave™, but if you want to download the mwave_sdk.jar, java doc and sample project you can get it from here:
http://support.motivewave.com/sdk/

12.2 Installing Java

If you have not done so already, you will need to download and install the Java Development Kit (JDK). Please note: this is different than the Java Runtime Environment (JRE) as it contains development tools such as the Java compiler (javac).

Version 5 of MotiveWave™ supports Java 1.8.0_121 and higher. If you are using an older version of MotiveWave, you should check the installed Java version by choosing *Help -> About* from the console menu bar.

The Java Development Kit can be downloaded here:
http://www.oracle.com/technetwork/java/javase/downloads/index.html

12.3 Installing Eclipse

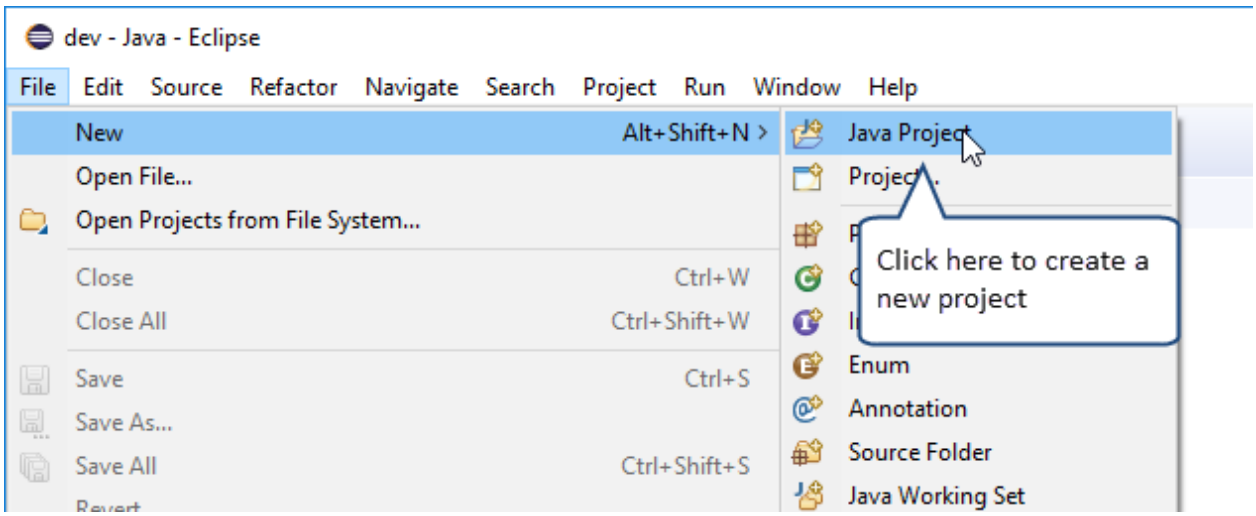
Eclipse comes in many different versions (and flavors). For our purposes we just need basic Java functionality so we will download 'Eclipse IDE of Java Developers'. This can be found at the following website: http://www.eclipse.org/downloads/

There are many different tutorials and books available to help you get started with Eclipse. If you don't want to search the internet, you can start here: http://www.eclipse.org/resources/?category=Tutorial

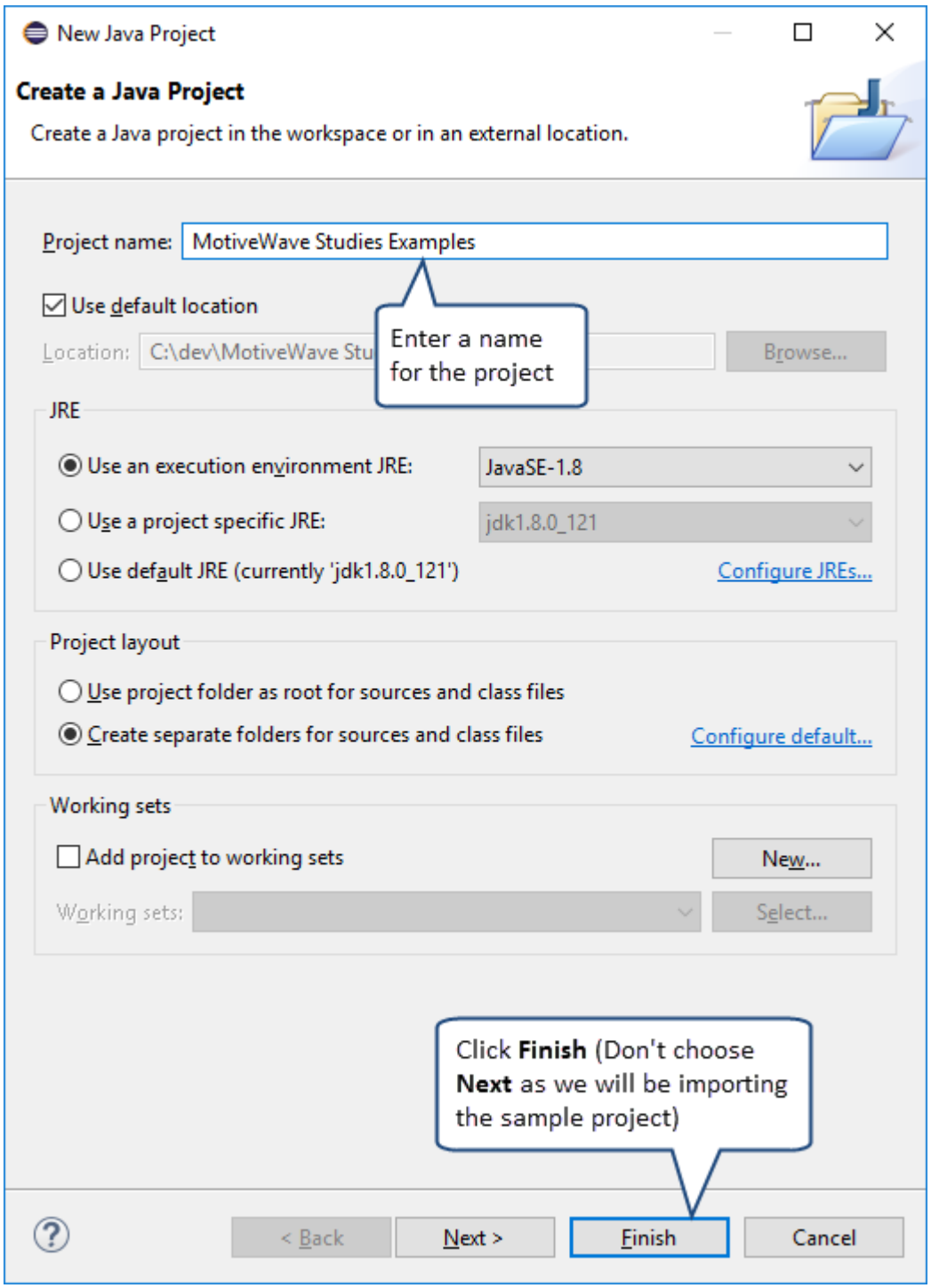
Here is a link to an introduction of the Java IDE: http://www.eclipse.org/resources/resource.php?id=505

12.4 Creating a Project

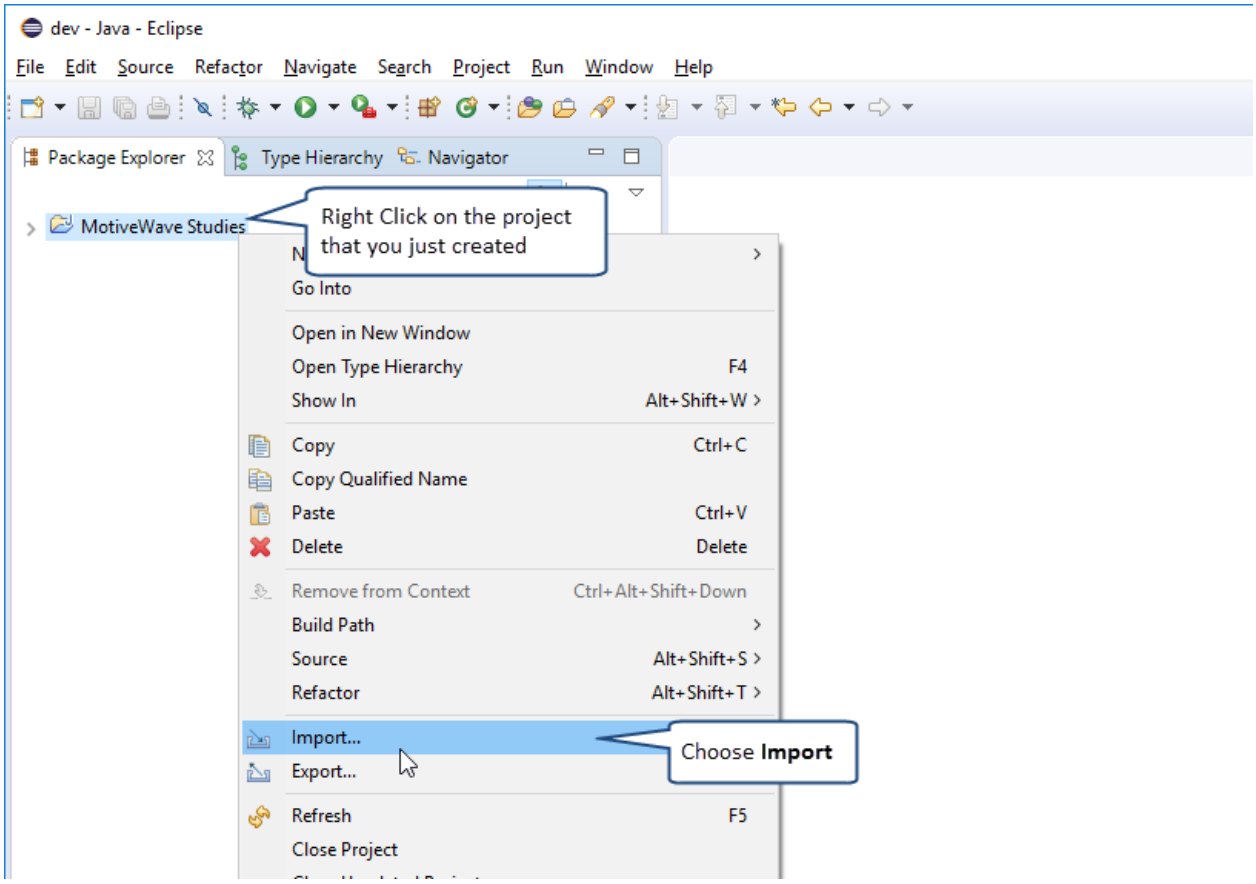
The first step to creating your own extensions is to create a project in Eclipse. From the top menu bar of Eclipse choose: *File -> New -> Java Project*



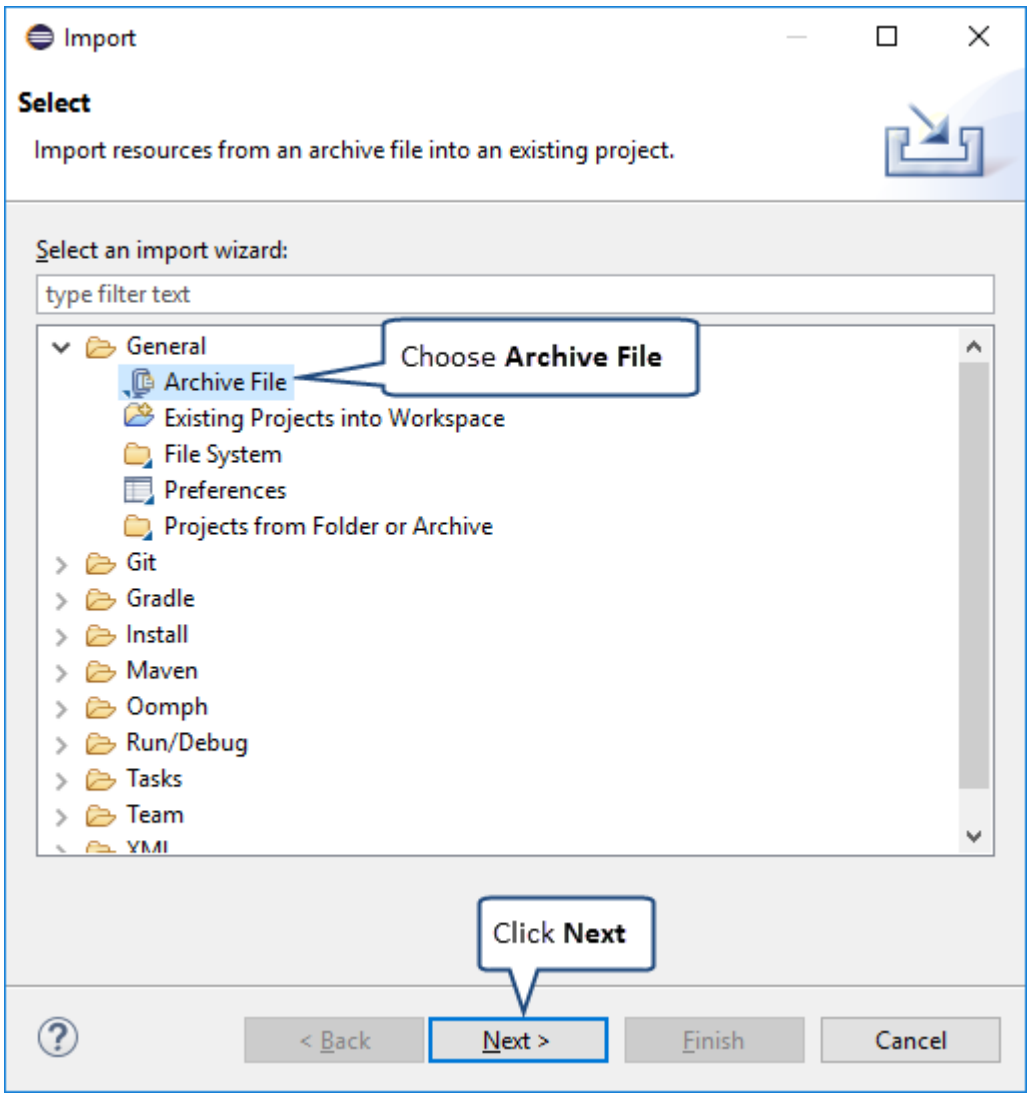
This will launch the New Project Dialog (see below). Enter a name for the project and click the **Finish** button. In the next step we will be importing the sample project so there is no need to configure anything specific for this project.



Now that you have created an initial project you can import the sample project files from the zip file 'MotiveWave Studies.zip'. Right click on the 'MotiveWave Studies' project that you just created and choose 'Import…' from the menu.

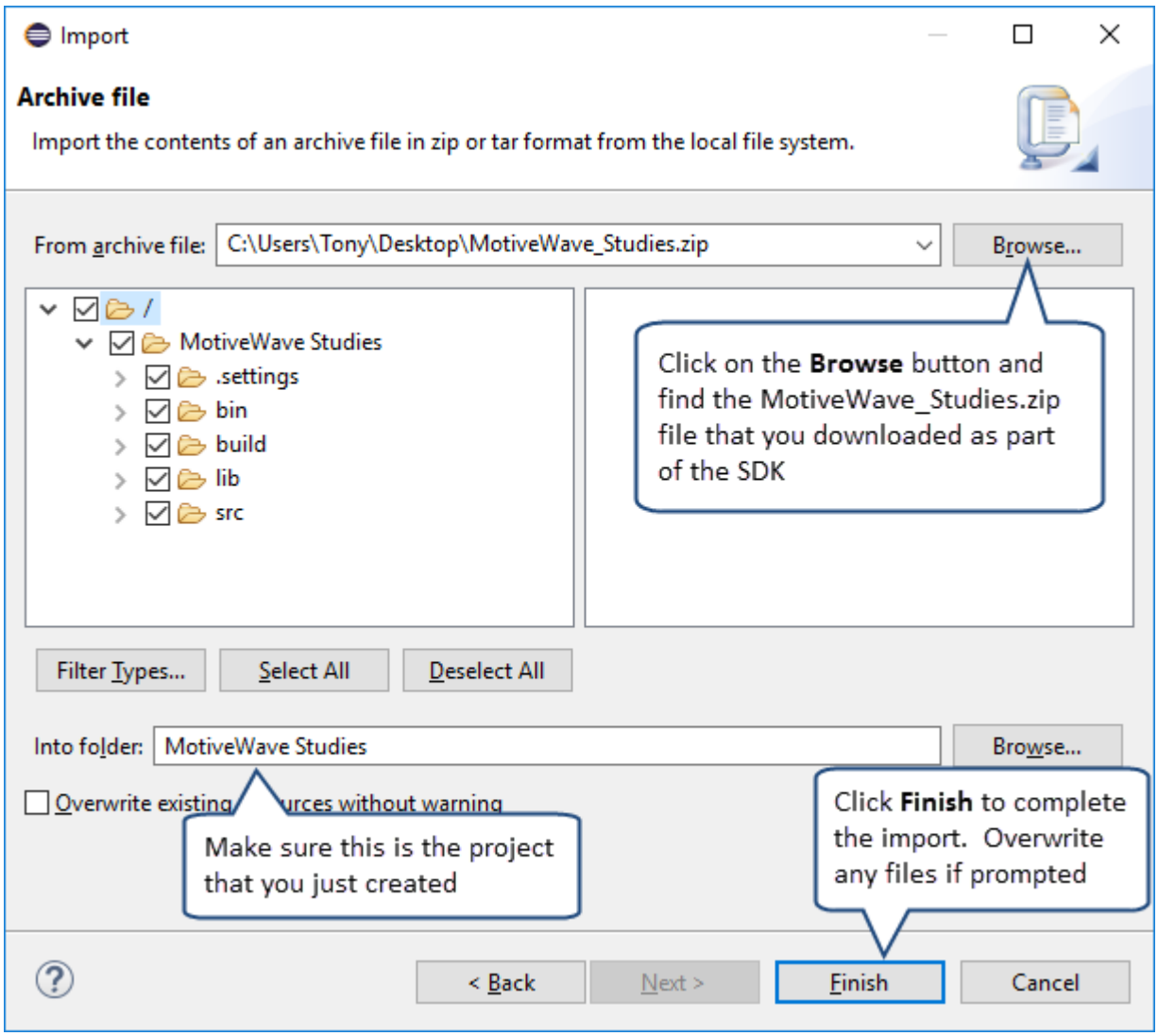


From the Import Dialog, open the 'General' folder and choose 'Archive File'

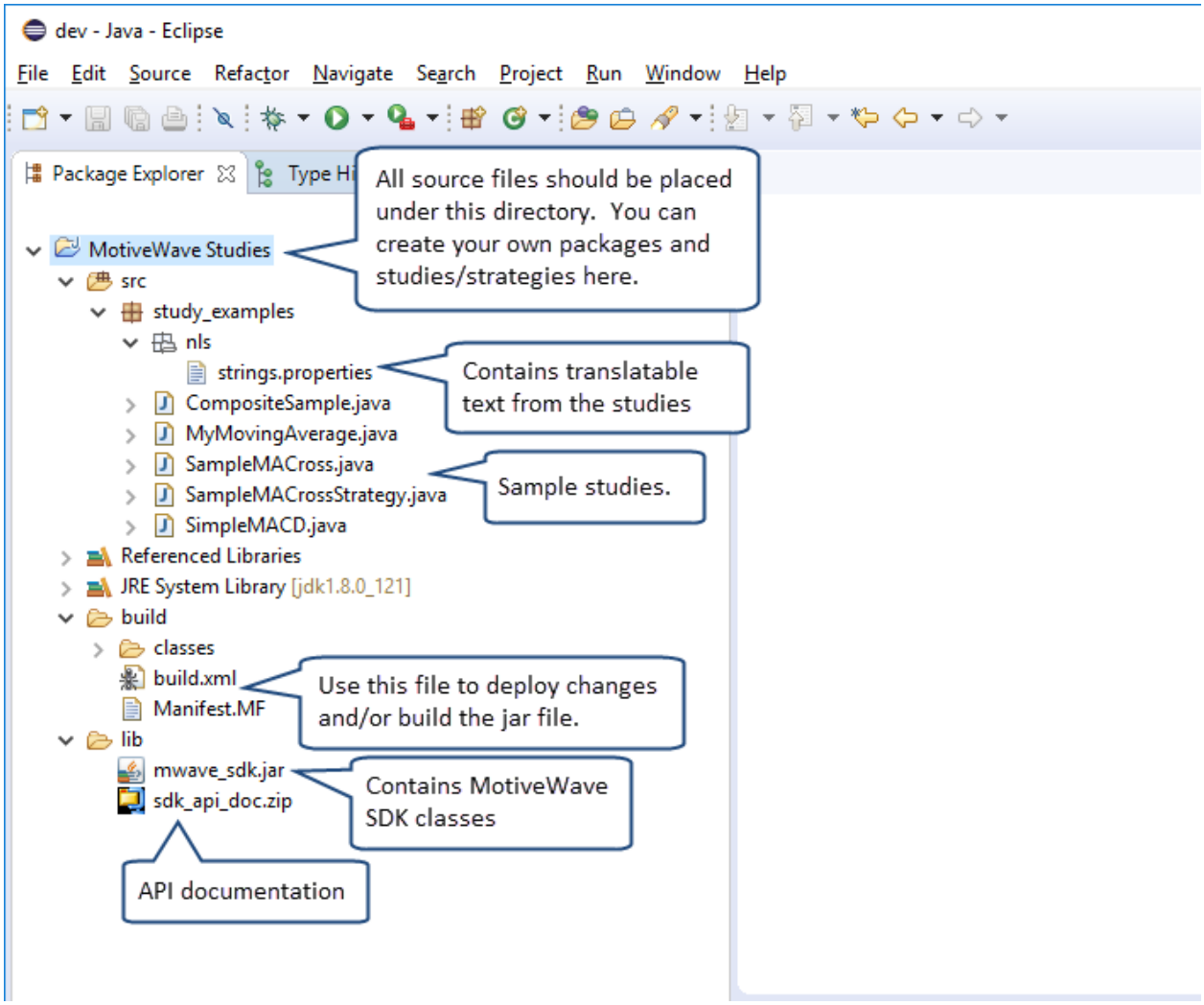


The next step is to specify the archive (.zip) file. In this case it will be 'MotiveWave Studies.zip'. If you have not done so already, download this sample project from the MotiveWave™ (see: http://www.motivewave.com/support/sdk.htm)

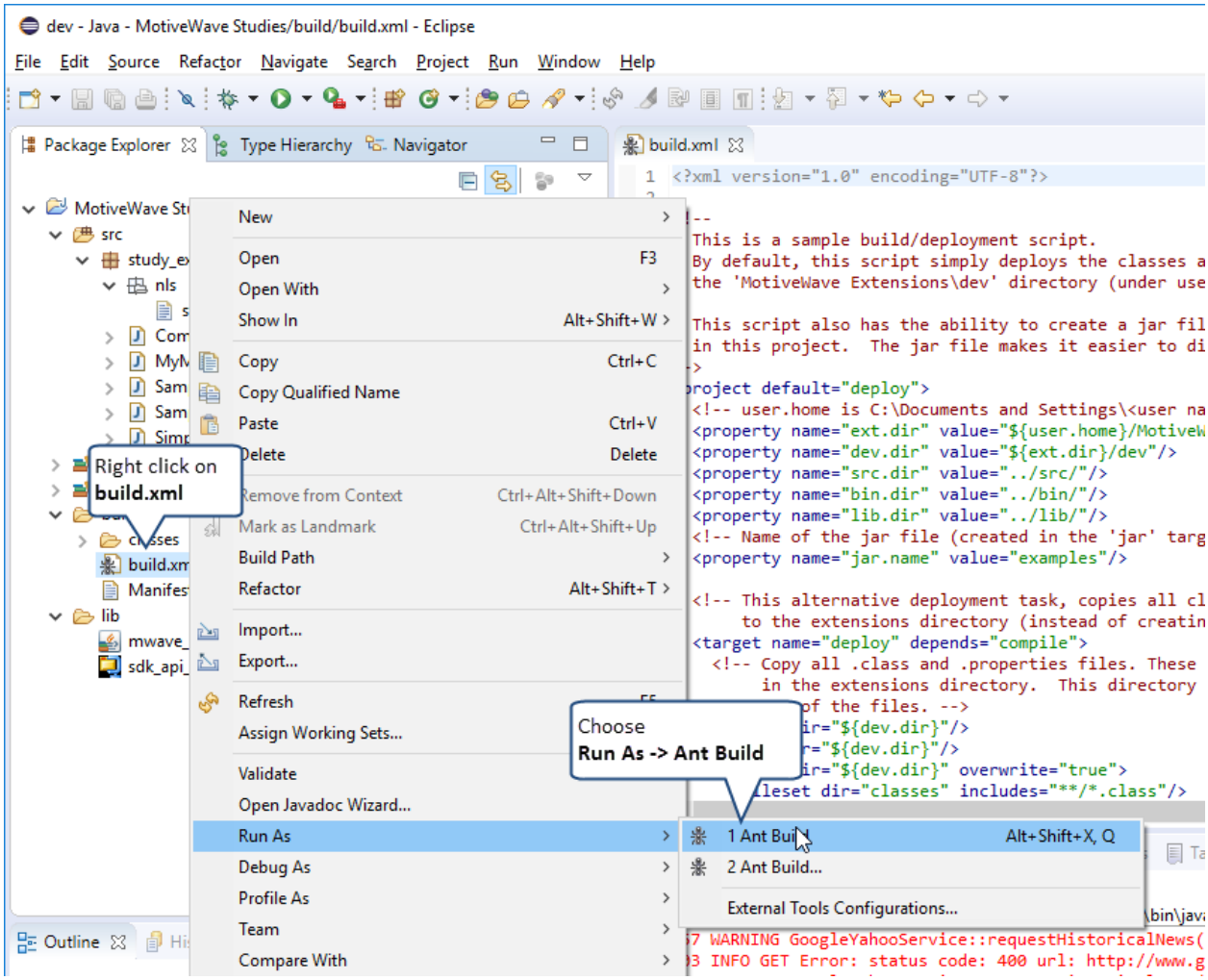
You may be asked to confirm the overwriting of some files like '.classpath'. If this happens, press OK to accept the changes.



Once the import is complete, the structure of you project should look like the screenshot below.



Now that you have the project created, you can deploy this to MotiveWave™. The 'build.xml' file (under the 'build' folder can be used to compile your code and copy the files to the 'MotiveWave Extensions' directory. Right click on this file and choose '*Run As -> Ant Build*'



By default this will run the ‘deploy’ target. This task will compile all of the source code under the ‘src’ folder and then copy it to the MotiveWave Extensions directory. Finally, it will modify the ‘.last_updated’ file to signal MotiveWave™ that it should scan for changes and load them.

The ‘Console’ tab will show the output from this action. It should look similar to the following screen shot:

```
Problems  Javadoc  Search  Console  Progress  Task List  (x)= Variables  Breakpoints
<terminated> MotiveWave Studies build.xml [Ant Build] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Feb 13, 2017, 10:06:05 AM)
Buildfile: C:\dev\MotiveWave Studies\build\build.xml
clean:
   [delete] Deleting directory C:\dev\MotiveWave Studies\build\classes
compile:
    [mkdir] Created dir: C:\dev\MotiveWave Studies\build\classes
    [javac] Compiling 5 source files to C:\dev\MotiveWave Studies\build\classes
deploy:
   [delete] Deleting directory C:\Users\Tony\MotiveWave Extensions\dev
    [mkdir] Created dir: C:\Users\Tony\MotiveWave Extensions\dev
     [copy] Copying 12 files to C:\Users\Tony\MotiveWave Extensions\dev
BUILD SUCCESSFUL
Total time: 1 second
```

Finally, if you have MotiveWave™ running, you should see an extra menu 'Examples' under the 'Study' menu that contains the two sample studies (see below)