Writing an Indicator Cookbook

Thomas Weigert, weigert@mst.edu
Extensively revised by Robert A. Schmied, ras@acm.org

Last author update: May 2008
This revision: *Author* : *ras Revision* : 1.6.1.19 *Date* : 2012/02/2623 : 26 : 49

Abstract

The process to write (code) GeniusTrader Indicators is described along with an explanation of the GT::Indicator::BOL and other examples. Analyses that must be performed in order to produce a correct indicator are discussed.

1 Introduction

This note attempts to summarize lessons learned writing GT indicators, in the hope that this will be useful to other GT developers.[1] The Bollinger Band indicator is used as a starting point and its implementation is explained in detail. Subsequent sections illustrate more advanced aspects of writing GT Indicators. Lastly, programmatic indicator testing strategies are discussed.

[1] The general techniques discussed here apply equally to the writing of signals, albeit the name of some of the key functions changes. For example, the main function to be written for a signal is `detect()`, which takes the place of `calculate()`.

Text like the following is used to denote indicator code

```
package GT::Indicators::BOL;

# Copyright 2000-2002 Raphaël Hertzog, Fabien Fulhaber
# This file is distributed under the terms of the General Public License
# version 2 or (at your option) any later version.

# Standards-Version: 1.0

use strict;
```

Text like the following is used to denote pod code

```
=pod

  this is perl pod or plain old documentation
  try $ perldoc perlpod for all the dirty details

=cut
```

Text like the following is used to denote suggested indicator testing code

```
#!/usr/bin/perl
#
# indicator testing code
#
```

With respect to terminology, a GT Indicator defines a time series. In general, a time series is a sequence of data values, ordered linearly by time. The prices of a market (i.e. a specific marketable security, such as IBM or AAPL) is also a time series (and also an indicator; consider `{I:Prices CLOSE}`). A GT Indicator is constructed by some or all of the following:

- one or more time series.
- an application of a transformation to a time series.
- the sequential application of a computation on the individual values of one or more time series.

2 Header

The top few lines of an indicator module file sets up the module as an indicator and loads the minimally required GT dependencies.

2.1 Package definition and object initialization

Line 1 defines the formal perl name of this module. Lines 3, 4 and 6 **use** clauses are standard for all indicator modules.

```
1  package GT::Indicators::BOL;
2
3  use strict;
4  use vars qw(@ISA @NAMES @DEFAULT_ARGS);
5
6  use GT::Indicators;
```

2.2 Including dependent packages

The next few lines load the packages this indicator depends on. For example, Bollinger Bands are moving averages that envelope a securities price. It consists of three series: A simple moving average of the securities price, and two series *n*-standard deviations above and below the price moving average. Thus, in this particular case, we need to include these three perl modules: GT::Indicator::SMA, GT::Indicator::StandardDeviation, and GT::Prices.

```
7  use GT::Indicators::SMA;
8  use GT::Indicators::StandardDeviation;
9  use GT::Prices;
```

For the most part all indicators will require line 9, because indicators almost always manipulate price data directly.

2.3 Input parameters

The **@ISA** assignment specifically defines this package to be an instance of an *indicator object*. The default arguments statement (**@DEFAULT_ARGS**) defines the default values for the input parameters of the indicator. These are either constant values or they name another indicator (data series) that supplies the current value of the its' data for this indicator.

```
10
11 @ISA = qw(GT::Indicators);
12 @DEFAULT_ARGS = (20, 2, "{I:Prices CLOSE}");
```

In this example, the first two parameters are by default given the constant values 20 and 2, respectively. The third parameter is the result of evaluating the indicator **{I:Prices CLOSE}** at the current period, yielding the current close of the prices array.

An indicators' input arguments are accessed via the methods defined in module GT::ArgsTree.

```
$self->{args}->get_arg_constant($n)
$self->{args}->get_arg_names($n)
$self->{args}->get_arg_values($calc, $i, $n)
```

where **$n** is the argument number (starting at 1), **$calc** is a calculator object, and **$i** is the current period.

The first form requires that the argument is a constant, which can be tested by

```
$self->{args}->is_constant($n)
```

Refer to module GT::ArgsTree documentation for more detail on the methods dealing with calculation objects (e.g. indicators, signals and systems).

The second form will obtain names of the corresponding argument (for constant arguments, the name is the same as its value).[2] The final form obtains both constant and non-constant values for a given period (of course, constants are the same for all periods).

Most indicators as currently defined perform no type checking on their parameters, resulting in fatal errors when parameters of the wrong type are passed. Care must be taken to pass constant parameters when such are expected, and time series as parameters, where those are required.[3]

[2] Several indicators use **get_arg_names()** in a context where the argument is not guaranteed to be constant, and thus will fail when a non-constant parameter is given (e.g. the name of a series).

[3] Proper run time checking of parameter types is advisable when writing a new indicator.

3 Output values

Indicators produce one or more output values (e.g. data series). In other words, for each period, an indicator will output one or more values. Output values are defined in the names clause:

```
13 @NAMES = ("BOL[#1,#3]","BOLSup[#1,#2,#3]","BOLInf[#1,#2,#3]");
```

In this instance, we define three output series for Bollinger Bands: the moving average, and the upper and lower bands. These values can be referred to by the names given in the above clause, where the arguments (hashed numbers in brackets) are replaced by the corresponding input parameters of the indicator. The symbol **#n** is replaced by the corresponding input parameter of the indicator.

The default name of an indicator (a data series) is the name of the first output series from the **@NAMES** clause, so in this example the name is **I:BOL[#1,#3]**. Note that the first and third arguments are included in the indicator name so when the arguments are substituted, this indicators' name becomes: **BOL[12 {I:Prices CLOSE}]**[4]. As a consequence of this naming scheme **{I:BOL 12 2}** and **{I:BOL 12 10}** cannot be distinguished when they are both used at the same time. Therefore care should be taken to select the the output series name wisely so that there are no conflicts between indicators. In the code the name of an output series can be either used literally as a string (e.g. **"I:BOL[12 {I:Prices CLOSE}]"**), or it can be obtained by the **get_name()** method (module GT::Registry)

```
$self->get_name
$self->get_name($n)
```

where **$n** refers to the position of the output series (e.g. an @NAMES element) (starting at **0**).

The values of the output series are set and read via a calculator object **$calc**, where **$name** is the name of the output series and **$i** is the period:

```
$calc->indicators->get($name, $i)
$calc->indicators->set($name, $i, $value)
```

See modules GT::Calculator and GT::CacheValues for details on calculator objects and methods that apply to them.

[4] Actually the GT::ArgsTree hash for the I::BOL entry key 'full_name' will not include the indicator name (e.g. BOL) nor the square brackets, just the string '12 {I:Prices CLOSE}'.

4 Initialization Method

If an indicator requires intermediate series to compute its value or requires data from past periods, these are set up in the **`initialize()`** method. This method is passed an indicator object as the single parameter:

```
14
15 sub initialize {
16     my ($self) = @_;
```

4.1 Intermediate series

Many indicators depend in their computation on other series. For example, Bollinger bands need the simple moving average of the price of the security for each period, as well as the standard deviation of the price of the security for each period. Both of these intermediate values form a series. Each intermediate series must be created in the **`initialize()`** method and be assigned to an attribute of the indicator. A series is created by calling the new method on its class and passing the appropriate arguments, or by evaluating the textual representation of the indicator defining the series. These intermediate series may rely on output values or on temporary data, see Section 7.2. For Bollinger Band we define the I:SMA (lines 18 .. 20) and I:StandardDeviation (lines 21..24) as intermediate series, passing both the first (period) and third (data array the indicator is applied to, typically I:Prices CLOSE) arguments.

```
17
18     $self->{'sma'} =
19      GT::Indicators::SMA->new([ $self->{'args'}->get_arg_names(1),
20                                 $self->{'args'}->get_arg_names(3) ]);
21     $self->{'sd'} =
22      GT::Indicators::StandardDeviation->new(
23       [ $self->{'args'}->get_arg_names(1),
24         $self->{'args'}->get_arg_names(3) ]);
```

If the indicator is not given any parameters upon creation, the default values are used.

Note that when such an intermediate series uses other series as its arguments, these cannot be defined by their constructor functions but must be given in their textual representation. For example, the following doubly smoothes the SMA defined above:

```
$self->{'sma'} = GT::Indicators::SMA->new(
 [ $self->{'args'}->get_arg_names(1),
   '{I:SMA ' . $self->{'args'}->get_arg_names(1)
             . ' ' . $self->{'args'}->get_arg_names(3) . '}'
 ]);
```

by creating an additional I:SMA data series in 'textual context' (observe the concatentation operators in the 2nd and 3rd lines of the example)

1. the indicator name ({I:SMA) a literal string
2. the time period name (a string) (**`$self->{'args'}->get_arg_names(1)`**)
3. a single whitespace ( )
4. the data series name (a string) (**`$self->{'args'}->get_arg_names(3)`**)
5. the indicator string closing curly bracket (})

yielding **`{I:SMA 12 {I:Prices CLOSE}}`**. This data series is then used as input for the **`$self->{'sma'}`** intermediate series.

An intermediate series can also conveniently be constructed using the **`GT::Eval::create_standard_object()`** method:

```
$self->{'sma2'} = GT::Eval::create_standard_object(
 "I:SMA", "12 {I:Prices CLOSE}");
```

Note that the arguments in this example are constant strings, and therefore cannot be modified by user arguments.

4.2 Intermediate data series computation

During the computation of an indicator, intermediate series' are either computed via the dependency mechanism (see Section 4.3) or by explicitly computing the series via:

```
$self->{'sma'}->calculate($calc, $i)
$self->{'sma'}->calculate_interval($calc, $i, $j)
```

where **`$calc`** is a calculator object, and **`$i`** and **`$j`** are time period index values. Note these method-invocation statements are not in the I:BOL example, but similar ones can be found in other GT indicator modules. Also note that the reference **`$self->{'sma'}`** shown here is just an example, it can be any intermediate series object defined by the indicator.

The values of these series is obtained via the standard get method from package GT::CacheValues (automatically imported via GT::Indicator via GT::Calculator). For example, the i-th value of the SMA is obtained via:

```
my $sma_value = $calc->indicators->get($self->{'sma'}->get_name, $i);
```

and when `$self->{'sma'}->get_name` is a object reference, as in the I:BOL example, the get method can appear as

```
my $sma_value = $calc->indicators->get($sma_name, $i);
```

as will be seen in Section 5.2.

4.3 Dependencies

Many indicators depend on past data to calculate their current value, either on past price information, or on the previous values of the indicator or on the previous value of intermediate series. A key feature of GT is that the computation of those past values can be largely driven automatically through a *dependency mechanism*. We can declare the current value of an indicator to be dependent on the previous values of its parameters, or of other series, or of the price information it is operating on. Such dependencies are declared for p periods of data; when updating dependencies those p values will be ensured to be available. To satisfy dependencies may in turn require additional data, the computing the dependencies may in turn depend on other values. The dependency mechanism propagates automatically until all dependencies are satisfied or it determines a dependency cannot be satisfied.

Determining the correct dependencies is important to be able to compute the indicator both correctly and efficiently. If too little data is available, an indicator may not be able to be computed, at best, or may give incorrect results, at worst. If too much data is required, less history of an indicator can be computed.

When the dependencies are known at the time the indicator is created (*static dependency*), the dependencies are defined in `initialize()` method (see Section 4). For *volatile dependencies*, which allow dependencies to be computed dynamically during the computation of the indicator itself refer to Section 7.3.

4.4 Static dependencies

The following methods can be used to define *static dependencies*:

```
$self->add_indicator_dependency($indic, $p)
$self->add_arg_dependency($n, $p)
$self->add_prices_dependency($p)
```

where **$indic** is an indicator object, **$n** refers to the *n*-th parameter of the indicator (counting from 1), and **$p** is the number of (prior) periods of data this value depends on. The first form states that the current value of the indicator depends on **$p** periods of data of indicator **$indic**. The second form states that the indicator depends on **$p** periods of data referenced by parameter **$n**. The third form states that the indicator depends on **$p** periods of data of the input series (this form of dependency is only needed when the indicator depends on more data periods than is established by the dependency mechanism).[5]

For the Bollinger Bands, each value depends on the current value of the moving average and the standard deviation. However, dependencies require at least one day of data, and thus the statements below declare the current value to be dependent on only 1 day of data of the moving average and the standard deviation.

```
25
26      $self->add_indicator_dependency($self->{'sma'}, 1);
27      $self->add_indicator_dependency($self->{'sd'}, 1);
```

Note that each of these intermediate series in turn require data to be computed, possibly with additional dependency requirements. However, the definition of these intermediate indicators already include their dependency needs therefore their dependency requirements (if any) are automatically accounted for by the dependency mechanism.

The Bollinger Band indicator also establishes a dependency on the period passed as the first parameter, assuming that parameter is constant. However, this declaration is technically not necessary, as this dependency is already established by the dependencies of the intermediate series.

```
28      if ($self->{'args'}->is_constant(1)) {
29          $self->add_prices_dependency($self->{'args'}->get_arg_constant(1));
30      }
31  }
```

[5] While it can be found in a number of indicators, this dependency is rarely (if ever) actually needed.

4.5 Indicator dependency analysis

from multiple gt-devel list emails posted by Thomas Weigert in March 2008

When developing an indicator, with respect to dependencies:

1. first you need to decide whether it makes sense to allow the period it depends on to vary during the computation of the indicator.
2. if not, use static dependencies.
3. if so, use volatile dependencies. As a consequence it is imperative to make sure that the implementation of **calculate_interval()** properly handles volatile dependencies.

With regards to the existing indicators, there are some indicators that will fail when given non-constant parameters for the period(s) they depend on. To avoid this, two things should be done:

1. if an indicator only supports static dependencies, one should verify that the given parameter is a constant (using **$self->{args}->is_constant**).
2. if an indicator wants to support volatile dependencies, the **calculate_interval()** computation needs to either manually retrieve the changing arguments or forward to calculate.

It turns out that not all these indicators work with non-constant period parameters either. Two [sic] problems occur:

1. due to protection against non-constant arguments, dependencies are not defined (example: I:BOL)
2. an intermediate series is defined that depends on a parameter which is treated as constant in initialize, and therefore, is defined with incorrect dependencies example: I:BOL)
3. due to lack of protection against non-constant arguments, dependencies are incorrectly defined as typically huge numbers (example: I:RSquare)
4. when dynamic arguments are passed in, these could be negative, resulting in bizarre output (example: I:ENV)
5. or these arguments may be real numbers, and the indicator might not take proper account of these (example: I:AROON; this case is really subtle: I:AROON actually protects arguments to make sure they are retrieved even if non-constant, but then in its computation of MinInPeriod and MaxInPeriod, real numbers are thrown away as period)

In summary, using dynamically changing parameters for dependency periods requires that arguments are carefully treated. One needs to ensure that

```
  1. non-constant arguments are correctly retrieved
  2. illegal values (e.g. negative) are discarded
  3. potentially illegal values (e.g. reals) are converted to integers

Most indicators do not pay attention to these dependency issues,
and assume that the arguments are correct.
```

5 Calculating the value of the indicator

The GT framework provides two means of calculating the value of an indicator: we can either compute a single value of the indicator, given its dependencies, or we can compute the value of the indicator throughout a given interval. One or the other of these methods must be defined[6], albeit often both methods are given. Typically, calculating the value of the indicator over the full interval required will be faster, potentially much faster as calculating the value of the indicator one period at a time may often repeat much of the computation needlessly.

5.1 Calculating a single value of the indicator

Define calculate method. The current value of the indicator is computed by the **`calculate()`** method, which takes as arguments a calculator object **`$calc`** and the current period **`$i`**. This method typically follows the following steps:

```
32
33 sub calculate {
34     my ($self, $calc, $i) = @_;
35     my $nsd = $self->{'args'}->get_arg_values($calc, $i, 2);
36     my $sma_name = $self->{'sma'}->get_name;
37     my $sd_name = $self->{'sd'}->get_name;
38     my $bol_name = $self->get_name(0);
39     my $bolsup_name = $self->get_name(1);
40     my $bolinf_name = $self->get_name(2);
```

[6] Note that if the **`calculate()`** method is omitted, the indicator may fail if this method is indirectly invoked (e.g. when running **anashell.pl**), as this method is not defined in the superclass. It is safer to omit the **`calculate_interval()`** method.

Define temporary variables. Note that several temporaries are defined for convenience: The distance of the upper and lower bands from the moving average, as determined by the second parameter (`$nsd`), the names of the intermediate series used (`$sma_name` and `$sd_name`), and the names of the output values (`$bol_name`, `$bolsup_name` and `$bolinf_name`).

Return if the values of the indicator are already available. These may have been computed earlier, thus there is no need to proceed further.

```
41
42      return if (
43          $calc->indicators->is_available($bol_name, $i)
44       && $calc->indicators->is_available($bolsup_name, $i)
45       && $calc->indicators->is_available($bolinf_name, $i) );
```

Return if the dependencies required can not be computed. This conditional statement invokes the `$self->check_dependencies` method, (the dependency mechanism). It attempts to resolve and then compute all of this indicators dependencies. The method returns true if it succeeds or false if one or more of the dependencies cannot be computed.

```
46      return if ( ! $self->check_dependencies($calc, $i) );
```

Compute the current value of the indicator. For the Bollinger Band indicator, we first obtain the values of the moving average and the standard deviation. The upper band is obtained by adding the appropriate factor of the standard deviation to the moving average; the lower band is calculated similarly.

```
47
48      my $sma_value = $calc->indicators->get($sma_name, $i);
49      my $sd_value = $calc->indicators->get($sd_name, $i);
50
51      my $bolsup_value = $sma_value + ($nsd * $sd_value);
52      my $bolinf_value = $sma_value - ($nsd * $sd_value);
```

Note that computing the current value of the indicator may in fact require iterating over past periods.

Update the output values for the current period. For the Bollinger Band store the moving average value into the first output series, the upper band value into the second output series, and the lower band value into the last output series.

```
53
54      $calc->indicators->set($bol_name, $i, $sma_value);
55      $calc->indicators->set($bolsup_name, $i, $bolsup_value);
56      $calc->indicators->set($bolinf_name, $i, $bolinf_value);
57  }
```

5.2 Calculating a the indicator throughout an interval

The `calculate_interval()` method computes the value of the indicator over a given interval. It is passed a calculator as well as the beginning and end of the interval of interest. This method can be obtained systematically from the `calculate()` method by the following steps:

1. Change all occurrences of `get_arg_values` method to the corresponding `get_arg_constant()` method.
2. Change all occurrences of `is_available()` method to the corresponding `is_available_interval()` method.
3. Change all occurrences of `check_dependencies()` method to the corresponding `check_dependencies_interval()` method.
4. Compute the current value of the indicator within a loop over the interval range (e.g. generally with the time period index variables `$first` and `$last`).

Each of these method changes are highlighted in black in `calculate_interval()` listing that follows. Note that the loop (line 74) has a range from `$first` to `$last`

```
58
59  sub calculate_interval {
60      my ($self, $calc, $first, $last) = @_;
61      my $nsd = $self->{'args'}->get_arg_constant(2);
62      my $sma_name = $self->{'sma'}->get_name;
63      my $sd_name = $self->{'sd'}->get_name;
64      my $bol_name = $self->get_name(0);
65      my $bolsup_name = $self->get_name(1);
66      my $bolinf_name = $self->get_name(2);
67
```

```
68      return if (
69          $calc->indicators->is_available_interval($bol_name, $first, $last)
70       && $calc->indicators->is_available_interval($bolsup_name, $first, $last)
71       && $calc->indicators->is_available_interval($bolinf_name, $first, $last) );
72      return if ( ! $self->check_dependencies_interval($calc, $first, $last) );
73
74      for (my $i = $first; $i <= $last; $i++) {
75        my $sma_value = $calc->indicators->get($sma_name, $i);
76        my $sd_value = $calc->indicators->get($sd_name, $i);
77
78        my $bolsup_value = $sma_value + ($nsd * $sd_value);
79        my $bolinf_value = $sma_value - ($nsd * $sd_value);
80
81        $calc->indicators->set($bol_name, $i, $sma_value);
82        $calc->indicators->set($bolsup_name, $i, $bolsup_value);
83        $calc->indicators->set($bolinf_name, $i, $bolinf_value);
84      }
85  }
86
87  1;
```

If method **`calculate_interval()`** is not provided by a particular indicator module, it is inherited from the indicator object module (GT::Indicators) which simply calls **`calculate()`** for each time-period index in the interval. Typically, a provided **`calculate_interval()`** method would ***not*** invoke the **`calculate()`** method directly since there is nothing to be gained over the inherited version.

6 Module end of file

As common practice in Perl modules, conclude the file with a perl 'true' value (line 87).

7 Additional capabilities

There are a number of additional tools provided by GT which are not leveraged in the Bollinger Bands indicator illustrated above. These are discussed below.

7.1 Temporary series

In addition to storing results in output values, as discussed in Section 3, an indicator may also store data into temporary series that are not visible outside of the indicator. To create a temporary series, assign the I:G:Container indicator to an attribute of the indicator object. For example:

```
my $name = $self->get_name;
$self->{'temp'} = GT::Indicators::Generic::Container->new(
 [ "temp($name)" ] );
```

The example creates a new temporary series with the name **temp($name)**, where **$name** has been assigned the first argument from the **@NAMES** array. (get_name() without parameter defaults to first argument) (refer to Section 3 for details of the **@NAMES** clause). The reason for inserting **$name**(parent indicators' name) into the name of the temporary series indicator is to ensure its uniqueness. Often this will not matter, but if several instances of this indicator are used at the same time, collisions may occur (for example, when this indicator is used in the long and short signals of a [GT-trading] system or a closestrategy).

This temporary series is an indicator and thus values can be read and written to this series as to any indicator:

```
$calc->indicators->get($self->{'temp'}->get_name, $i)
$calc->indicators->set($self->{'temp'}->get_name, $i, $value)
```

7.2 Constructing intermediate series from other series

An intermediate series may rely on another intermediate series, on a temporary series, or on an output series. In this situation, when defining an intermediate series, the dependent series are provided as parameters.

For example, to define a standard moving average of the upper band of the Bollinger Band indicator (within the computation of the Bollinger Band indicator) use:

```
$self->{'upper'} = GT::Indicators::SMA->new(
 [ $self->{'args'}->get_arg_names(1),
   "{I:Generic:ByName " . $self->get_name(1) . "}"
 ]);
```

This constructs an intermediate SMA from the second output series of the current indicator, with the period taken from the first parameter of the current indicator and assigns it to an attribute of the indicator object. The indicator **I:Generic:ByName** references another series by its name (i.e. the name of the first output series, see Section 3). Care must be taken that the correct name is used.

Similarly one can construct a series that depends on a temporary series or an intermediate series. For example, the simple moving average of the temp indicator from Section 7.1 is defined as follows:

```
$self->{'sma1'} = GT::Indicators::SMA->new(
 [ $self->{'args'}->get_arg_names(1),
   "{I:Generic:ByName temp}"
 ]);
```

The further smoothing of the simple moving average of the upper Bollinger Band (see first example this section 7.2) can be defined by:[7] [8]

```
$self->{'sma2'} = GT::Indicators::SMA->new(
 [ $self->{'args'}->get_arg_names(1),
   "{I:Generic:ByName " . $self->{'upper'}->get_name . "}"
 ]);
```

If the intermediate series has multiple outputs and the default value (first one) **is not** the desired value the name used must refer to the correct output value to be accessed. The method **$indicator_object->get_name($n)** (from GT::Indicator) can be used to construct a series based on the n-th output value of the intermediate series defined by the specified **$indicator_object**. Two examples: the first from GT::Indicators::ADX

```
$self->{'sma'} = GT::Indicators::SMA->new(
 [ $self->{'args'}->get_arg_names(1),
   "{I:Generic:ByName ". $self->get_name(3) . "}",
 ] );
```

and the second from GT::Indicators::VOSC

```
$self->{'sma'} = GT::Indicators::SMA->new
 ( [ $self->{'args'}->get_arg_names(1),
   "{I:Generic:ByName ". $self->get_name(1) ."}"
 ] );
```

[7] This requires a correction to the I:Generic:ByName indicator available from the mailing list archives at http://geniustrader.org/lists/devel/msg02362.html.

[8] committed to GT trunk r573 as of 03/18/2008.

7.3 Volatile dependencies

It is also possible for indicator dependencies to dynamically change during the computation of a series, either by the length of the dependency being computed at each iteration, or by it depending on the value of a series. Dynamically changing dependencies are referred to as *volatile dependencies*. They are defined analogously to static dependencies using the following methods

```
$self->add_volatile_indicator_dependency($indic, $p)
$self->add_volatile_arg_dependency($n, $p)
$self->add_volatile_prices_dependency($p)
```

where **`$indic`** is an indicator object, **`$n`** refers to the n-th parameter of the indicator (counting from 1), and **`$p`** is the number of periods of data this value depends on.

Before defining volatile dependencies, all volatile dependencies from the previous period must be removed through calling

```
$self->remove_volatile_dependencies()
```

Volatile dependencies are mostly useful only when indicators are calculated one period at a time (i.e. in the **`calculate()`** method).[9]

7.4 Accessing Volatile Dependency Arguments

paraphrased from gt-devel list email posted by Thomas Weigert March 2008

If an indicator wants to support volatile dependencies, it must ensure that all non-constant parameters are correctly retrieved. That can be done only via **`$self->{args}->get_arg_values`** which requires a calculator object and the period from which to retrieve that value. In particular, when using **`$calculate_interval()`**, one has to retrieve the proper values of the parameter for all periods the indicator is calculated for (this requires explicitly iterating over the series and obtaining that value).

An indicator can verify that arguments are, in fact, constants using the method **`$self->{args}->is_constant`**.

[9] Note that several indicators add volatile indicators in the **`calculate_interval()`** method. This will work only if careful attention is paid to that the dependency period is correctly obtained. In many such situations, the dependency period is established correctly only when the corresponding parameter is both constant and positive. Further, unless the dependencies are updated throughout the loop, they reduce to static dependencies (in those situations, if **`calculate()`** is desired to support volatile dependencies, it is useful to define the volatile dependencies also in **`calculate_interval()`** to avoid duplicated dependency computation in **`calculate()`** where static dependencies defined).

8 Styles of calculating indicators

The value of an indicator can be arrived at using any of three ways:

- by obtaining the value of an input data series, or
- by applying an indicator to a data series (either as an input series or as a temporary output series), or
- by performing some computation on the current or prior values of one or more available data series.

These can be combined in arbitrary ways. The example Bollinger Band indicator used each of these. It obtains the value of an input data series (I:Prices CLOSE) and applies two indicators (SMA, StandardDeviation) to these values, and then performs a calculation on the current value of these intermediate data series (indicators).

Other indicators require more complicated scenarios: For example, an indicator may require a smoothing of the calculated value (as in the Stochastic Oscillator (I:STO), the Fisher indicator (I:FISH), the Volume Oscillator (I:VOSC) or the Stochastic Momentum Indicator (I:SMI). Examining the Stochastic Momentum Indicator implementation; it first obtains values from an input series and applies an indicator to these values. It then performs some calculation to produce a temporary series, and applies smoothing to the temporary serie(s). Lastly I:SMI performs some additional computation on the smoothed temporary data, and applies a final smoothing to the result to establish SMI value(s).

These more complicated indicator calculations, as illustrated in the I:SMI description, can be constructed using the methodology outline below. Following the methodology a descriptive example based on I:VOSC is presented. First analyze the dependencies required by each step in theindicator calculation. Start the calculation at the earliest point in the chain of dependencies.

1. The current or previous value of an indicator can always be obtained as described above.
2. If an indicator application is not the final step, then calculate the value of that indicator starting from the earliest period it satisfies a dependency for subsequent computations up to the current period.

3. If a computation on current or past values of one or more series is not the final step, then calculate all subsequent values in a loop from the earliest period the computation satisfies a dependency for subsequent computations up to the current period.

For example, in the GT::Indicator::VOSC the @NAMES clause is

```
@NAMES = ("VOSC[#1]", "VOSC-volume[#1]");
```

The GT::Indicator::VOSC indicator computes the volume oscillator using the **calculate()** method listed below. The calculation is performed by first computing the value of the volume at line 23 and then smoothing that value with an SMA using a period given by the first parameter to the indicator (e.g. 'VOSC[#1]'). The smoothing is performed after the computation of the volume value (lines 10 .. 25). Thus the indicator first computes sufficient data values for the smoothing operator in the loop (lines 11 .. 25). Line 27 applies the smoothing operator and finally the I:VOSC[#1] output value is computed (line 29).

```
1  sub calculate {
2      my ($self, $calc, $i) = @_;
3      my $vosc_name = $self->get_name(0);
4      my $volume_name = $self->get_name(1);
5      my $volume = 0;
6
7      return if ($calc->indicators->is_available($vosc_name, $i));
8      return if (! $self->check_dependencies($calc, $i));
9
10     my $nb_days = $self->{'args'}->get_arg_values($calc, $i, 1);
11     for (my $n = 0; $n < $nb_days; $n++) {
12
13         next if $calc->indicators->is_available($volume_name, $i - $n);
14         if ($calc->prices->at($i - $n)->[$CLOSE] > $calc->prices->at($i - $n)->[$OPEN]) {
15            $volume = $calc->prices->at($i - $n)->[$VOLUME];
16         }
17         if ($calc->prices->at($i - $n)->[$CLOSE] < $calc->prices->at($i - $n)->[$OPEN]) {
18            $volume = -$calc->prices->at($i - $n)->[$VOLUME];
19         }
20         if ($calc->prices->at($i - $n)->[$CLOSE] eq $calc->prices->at($i - $n)->[$OPEN]) {
21            $volume = 0;
22         }
23         $calc->indicators->set($volume_name, $i - $n, $volume);
24
25     }
26
27     $self->{'sma'}->calculate($calc, $i);
28     my $vosc_value = $calc->indicators->get($self->{'sma'}->get_name, $i);
```

```
29      $calc->indicators->set($vosc_name, $i, $vosc_value);
30  }
```

The transformation to the **calculate_interval()** method is similar to that as described for the BOL example in Section 5.2 with the exception that the bounds of any loop used in **calculate()** will have to take the required data history into account.

For an example of a more complex indicator as well as for the transformation of the **calculate_interval()** method see the Stochastic Momentum Indicator GT::Indicator::SMI module.

9 Documentation

9.1 POD — plain old documentation

Adequate documentation in pod format should be provided for each indicator. In general pod should describe the indicator from the users point-of-view. Meaning the pod should define each of the arguments that are used by the indicator and the default values, if any, This discussion should be from the stand-point of a user of the indicator rather than from a programmers view. In other words the usage should be in terms of typical GT sys-sig-indic descriptions. It is up to the pod writer if a discussion of the computation is included. Also example usage, again from an end-user view is optional. This can include specific **graphic.pl** graphic directives and sys-sig-indic description for use by **scan.pl**, **display_indicator.pl**, **backtest.pl**, etc.

GT convention puts overall indicator[10] (module) pod between the end of the files' front-matter, (e.g. variable declarations) and the first subroutine in the module. So for the GT::Indicator::BOL example indicator pod would conventionally be inserted between lines 13 and 15.

```
13  @NAMES = ("BOL[#1,#3]","BOLSup[#1,#2,#3]","BOLInf[#1,#2,#3]");
14
15  <traditional location of general indicator module pod>
16
17  sub initialize {
```

[10] actually all GT module files

General indicator (module) pod format typically has this sort of general organization, but this is prior convention not policy:

```
=head2 <module name>

=head2 <indicator discussion overview etc>

=head2 <calculation>

=head2 <examples>

=cut
```

While it isn't essential that a blank line follow '=cut' it does help with some pod formatters. But this practice has not been a GT coding convention until lately.

Another GT pod convention sometimes adds pod ahead of each subroutine. This pod usually indicates the call format of the subroutine along with some context describing the purpose/use of it. This description might even go so far as describing the parameters the subroutine expects and if any are optional. In the case of the I:BOL example the following might be placed between lines 31 and 33:

```
31  }
32
33  =head2 C<< GT::Indicators::BOL::calculate($calc, $day); >>
34
35  Calculate the I:BOL values for time-period index $day using the calculator object $calc.
36
37  =cut
38
39  sub calculate {
```

These are just prior GT conventions, for the ease of module maintenance it may make more sense to put the bulk of the pod at the bottom of the module file. The most important thing is to provide pod that is useful, and that is the hard part.

9.2 Code Commentary

If you want the GT community to embrace your new GT module it is important to provide a modicum of *descriptive design intent* in the form of embedded comments. This is especially important for code where the operation is obvious but the reason to do so may be unclear, or be for an exception rather than a normal case or condition.

10 Indicator Testing

The GT toolkit currently provides very limited testing infrastructure. This section attempts to describe approaches that can be used to evaluate and prove (or more succinctly demonstrate) an indicator module is functional and functioning correctly.

10.1 Manual checking versus dedicated tests

GT application Script **display_indicator.pl** with appropriate arguments might be useful in manually checking the functionality of an 'in-development' indicator, however, there are no provisions to force selection of method `calculate()` vice the `calculate_interval()` method. ***(But refer to the GT Scripts/t directory for a preliminary set of scripts that do exercise each of these methods for result comparison).***

Therefore, it is most reasonable to write a specific test program (actually a perl test script) for a particular indicator. It is suggested these be located in a new subdirectory named 't' in the GT directory, and named after the base indicator. It is also suggested that subdirs be created to segregate test scripts by module type or by primary operational testing (e.g. indicators, signals, prices, graphic_objects, cs (closestrategies), of (orderfactory), etc). For example GT/t/indicators/BOL.t, GT/t/indicators/VOSC.t, GT/t/cs/stop-fixed.t, GT/t/cs/010_stop_sar.t, GT/t/cs/020_stop_sar.t. If muliple test scripts are needed some sort of suffix might be added, for example GT/t/indicators/BOL_1.t GT/t/indicators/BOL_2.t, or a 3 digit prefix might be devised as in the GT/t/cs/DDD_stop_sar.t examples.

10.2 Minimal coding validation

For each indicator module all basic user-application-interface operations should be verified such as:

- the module uses the built-in default arguments.
- the module accepts and uses user provided arguments.
- the module handles invalid user provided arguments in an acceptable manner

It's up to the module author to determine the meaning of *in an acceptable manner*.

Verifing indicator results is somewhat more complex and is not yet covered in the following subsections.

10.3 calculate() Method Testing

—to be supplied—

10.4 calculate_interval() Method Testing

—to be supplied—

10.5 Comparing calculate() and calculate_interval() Results

For any given indicator methods `calculate()` and `calculate_interval()` should yield identical results for each time-period with possible exceptions when the time-period is at or within a dependency time-period range at the start of the data series (usually GT::Prices), but this can happen for any intermediate dependency data series object.

The perl code in the GT Scripts/t directory is an initial attempt to develop a general methodology to perform `calculate()` and `calculate_interval()` result comparisons. ***note this code hasn't had a lot of attention and therefore it needs a bit (well more than a bit, more like 128 bits or more) of work . . .***

10.6 indicator testing infrastructure

In order to implement a (reasonably) portable indicator testing (and other GT toolkit) facility two indispensible elements are required as part of the testing methodology.

- prices data.
- gt configuration data (e.g. equivalent of $HOME/.gt/options).

The GT::DB::Text module along with various market data files can suffice for all testing needs excluding the unused GT::DB modules.

The sample market data files (http://geniustrader.org/examples/data.tar.gz) can be used or additional files can be supplied as part of a specific test package. It is suggested that test prices data be installed at GT/t/data, which is consistent with test infrastructure code that will be introduced.

The gt configuration data is only slightly more troublesome, and can be solved in a couple of ways that are reasonably efficient, practical and available now. In addition, they don't interfer (or interact) so both may be used concurrently without conflict. Implement your tests with which ever one you prefer. The default test options pathname is GT/t/options 10.6, if your testing requires changing it there are (at least) 3 options *[ordered by the revising authors' preference]*:

1. provide an add-on file (load into your tests using GT::Conf::load()),
2. create a custom version of the test options file,
3. you can request a revision of the default (`ras@acm.org`)

The first employs the GT::Test module[11] which can be used to essentially include the desired gt configuration data key-value pairs as a "here" document in the test script file. The advantage of this approach is the test script is self-contained. The disadvantage is each test script requires the duplication of all the necessary gt configuration data.

GT::Test module use example. A example GT::Test module test script file is presented here. GT indicator authors wishing to provide test scripts can choose to adopt this style of indictor module testing.

```
#!/usr/bin/env perl

use strict;
use lib '..';

use GT::Calculator;
use GT::DateTime;
use GT::Prices;
```

[11] originally developed within the CPAN'ed version of GT, an unCPANized version is attached as Appendix B

```
use GT::Test
    tests => 46,
    gt_config => sub {
        my $test_base = shift;
        my $db_path = File::Spec->catdir($test_base, 'data');
<<"EOF";
DB::module Text
DB::text::file_extension _\$timeframe.txt
DB::text::directory $db_path
EOF
    };

  my ($calc, $first, $last) = GT::Tools::find_calculator(
   GT::Test->gt_db, 'TX', $PERIOD_5MIN, 1);

<write perl Test::More test statements here>
```

The second approach simply creates an option file for a test or a group of tests. The advantage is less duplication of gt configuration data (in fact many test scripts will use the same 'common' file). The disadvantages include the need to create and maintain yet another file and the probability that some data values will be non-portable. In the example file there are a number of examples of this issue including the pathname values for keys DB::text::directory, Path::Aliases::Indicators and the various Path::Font::*. Unfortunately, the current GT toolkit fails to expand any environmental variable reference in files loaded via GT::Conf::load. Whether a key-value pathname value is relative to the current working directory or something else is not something ***known***, but could easily be evaluated by the very interested ...

Solutions to hard-paths in the default GT/t/options file:

1. create and load an auxillary options file
2. set the key DB::text::directory value appropriately in the test script itself using the GT::DB::Text::set_directory method

```
my $db = create_standard_object("DB::Text");
$db->set_directory("../GT/t/data");
```

It is suggested that a test script run correctly with the current working directory set to Scripts or GT.

Module test script example. A strawman GT Indicator test script example file is presented here. GT indicator authors wishing to provide test scripts can choose to adopt this style of indictor module testing.

```
#!/usr/bin/eval perl

use Test::More;
use Test::Differences;
use Data::Dumper;

use lib '..';
use GT::Conf;
use GT::Eval;
use GT::DateTime;

my $opt_file = '../GT/t/options';
my $timeframe = $DAY;

my $code = 13000;    # sample database first date=1993-01-04
my $mid_date = '1998-01-02';
my $object;

GT::Conf::clear();
GT::Conf::load( "$opt_file" );
my @object_types = ();

< write gt code to setup module for testing >
< write perl Test::More tests here >
```

The default GT/t/options file: The strawman default GT/t/options file is presented here. It's use it suggested for those GT indicator authors wishing to provide test scripts.

```
#
# genius trader test suite options file
#

DB::timeframes_available day,week,month,year

DB::module                      Text
DB::text::format                0 # 3 is default (for Text)
DB::text::directory             /usr/local/src/genius_trader/sample_data

Brokers::module SelfTrade

Path::Font::Arial /usr/openwin/lib/X11/fonts/TrueType/Arial.ttf
Path::Font::Courier /usr/openwin/lib/X11/fonts/TrueType/VeraMono.ttf
Path::Font::Times /usr/openwin/lib/X11/fonts/TrueType/TimesNewRoman.ttf
```

```
Path::Font::LucidaTypewriter /usr/openwin/lib/X11/fonts/TrueType/LucidaTypewriterRegular.ttf

Analysis::ReferenceTimeFrame year

Graphic::Histogram::Color lightblue
Graphic::ForegroundColor  black

Aliases::Global::TFS    SY:TFS 50 10|CS:SY:TFS
Aliases::Global::TFS[]  SY:TFS #1 #2|CS:SY:TFS #1|CS:Stop:Fixed #3

Path::Aliases::Indicators       /usr/local/src/genius_trader/ind_aliases
Aliases::Indicators::PVOL_opt   {I:Prices VOLUME}
Aliases::Global::TFS13[] SY:TFS #9 #10 | CS:SY:TFS #11 | CS:Stop:Fixed #13

GT::Conf::Test  hello, world!

GT::Options::Version    Revision: 1.1
```

10.7 Hard to Detect, Diagnose, and Isolate Bugs

A word or two about GT::Indicator gottchas. If the indicator relies on another GT::Indicator for one of its input data series its results are necessarily dependent on the other indicators' correctness in order to generate correct results. Diagnosing this type of problem can be challenging without writing tests for each intermediate step of the indicator calculation or writing tests for (all) other dependent indicators. The following paragraphs identify the source of some of the most difficult to detect, diagnose and solve bugs documented to date.

Data Series Gottchas Computing dependency time-period indices relative to an initial index (typically `$i` or sometimes `$n`) by repeated decrementing the value (i.e. stepping backwards in time). This approach can be problematic when the index value goes negative (e.g. less than zero). Perl will happily apply a negative index to an array access and thereby access the array element *from the other end of the array*, not usually what you want, certainly not usually expected in terms of programmatic operation, etc. However, because the data array access succeeds, the calculation proceeds as if nothing out-of-the-ordinary has happened, but the indicator is most likely using invalid data.

This type of error can be easy to overlook, extremely hard to detect, and even harder to diagnose especially if it isn't caused by the indicator under test or development, but is from a dependency indicator.

Examples of this sort of data series index calculation methodology are easily found in many of the GT::Indicators modules. Whether any particular one has the potential to cause adverse effects is the unanswered question.

Ways to detect when testing an indicator modules results:

- run multiple test cases where dependencies are deliberately set beyond the start of available data series.
- check indicator values at the start of the time period for consistency — a large difference between two adjacent values might indicate a input value discontinuity.
- —to be provided—

Ways to avoid:

- never ever assign from an array with a negative index. (e.g. check array indices >= 0)
- —to be provided—

A Example GT::Indicator::BOL source listing

The complete source file listing of the BOL indicator source code used as the example indicator is provide here for reference. Be aware that it has been 'sanitized' for inclusion in this document — all pod has been removed as well as all comments. In addition, note that this version ***differs significantly*** from any version at the head of any GT branch (as of q3 2011).

```
1  package GT::Indicators::BOL;
2
3  use strict;
4  use vars qw(@ISA @NAMES @DEFAULT_ARGS);
5
6  use GT::Indicators;
7  use GT::Indicators::SMA;
8  use GT::Indicators::StandardDeviation;
9  use GT::Prices;
10
11 @ISA = qw(GT::Indicators);
12 @DEFAULT_ARGS = (20, 2, "{I:Prices CLOSE}");
13 @NAMES = ("BOL[#1,#3]","BOLSup[#1,#2,#3]","BOLInf[#1,#2,#3]");
14
15 sub initialize {
16     my ($self) = @_;
17
18     $self->{'sma'} =
19      GT::Indicators::SMA->new([ $self->{'args'}->get_arg_names(1),
20                                 $self->{'args'}->get_arg_names(3) ]);
21     $self->{'sd'} =
22      GT::Indicators::StandardDeviation->new(
23       [ $self->{'args'}->get_arg_names(1),
24         $self->{'args'}->get_arg_names(3) ]);
25
26     $self->add_indicator_dependency($self->{'sma'}, 1);
27     $self->add_indicator_dependency($self->{'sd'}, 1);
28     if ($self->{'args'}->is_constant(1)) {
29         $self->add_prices_dependency($self->{'args'}->get_arg_constant(1));
30     }
31 }
32
33 sub calculate {
34     my ($self, $calc, $i) = @_;
35     my $nsd = $self->{'args'}->get_arg_values($calc, $i, 2);
36     my $sma_name = $self->{'sma'}->get_name;
37     my $sd_name = $self->{'sd'}->get_name;
38     my $bol_name = $self->get_name(0);
```

```
    my $bolsup_name = $self->get_name(1);
    my $bolinf_name = $self->get_name(2);

    return if (
        $calc->indicators->is_available($bol_name, $i)
     && $calc->indicators->is_available($bolsup_name, $i)
     && $calc->indicators->is_available($bolinf_name, $i) );
    return if ( ! $self->check_dependencies($calc, $i) );

    my $sma_value = $calc->indicators->get($sma_name, $i);
    my $sd_value = $calc->indicators->get($sd_name, $i);

    my $bolsup_value = $sma_value + ($nsd * $sd_value);
    my $bolinf_value = $sma_value - ($nsd * $sd_value);

    $calc->indicators->set($bol_name, $i, $sma_value);
    $calc->indicators->set($bolsup_name, $i, $bolsup_value);
    $calc->indicators->set($bolinf_name, $i, $bolinf_value);
}

sub calculate_interval {
    my ($self, $calc, $first, $last) = @_;
    my $nsd = $self->{'args'}->get_arg_constant(2);
    my $sma_name = $self->{'sma'}->get_name;
    my $sd_name = $self->{'sd'}->get_name;
    my $bol_name = $self->get_name(0);
    my $bolsup_name = $self->get_name(1);
    my $bolinf_name = $self->get_name(2);

    return if (
        $calc->indicators->is_available_interval($bol_name, $first, $last)
     && $calc->indicators->is_available_interval($bolsup_name, $first, $last)
     && $calc->indicators->is_available_interval($bolinf_name, $first, $last) );
    return if ( ! $self->check_dependencies_interval($calc, $first, $last) );

    for (my $i = $first; $i <= $last; $i++) {
      my $sma_value = $calc->indicators->get($sma_name, $i);
      my $sd_value = $calc->indicators->get($sd_name, $i);

      my $bolsup_value = $sma_value + ($nsd * $sd_value);
      my $bolinf_value = $sma_value - ($nsd * $sd_value);

      $calc->indicators->set($bol_name, $i, $sma_value);
      $calc->indicators->set($bolsup_name, $i, $bolsup_value);
      $calc->indicators->set($bolinf_name, $i, $bolinf_value);
    }
}

1;
```

B GT::Test module source listing

```
package GT::Test;

use strict;
use warnings;
use base 'Test::More';
use FindBin;
use File::Temp;
use GT::Conf ();
use GT::Eval ();
our @EXPORT = qw(play_prices_until);

=head1 NAME

GT::Test - Test helpers for GeniusTrader

=head1 SYNOPSIS

  use GT::Test
    tests => 2,
    gt_config => sub {
        my $test_base = shift;
        my $db_path = File::Spec->catdir($test_base, 'data');
        return "DB::module Text\nDB::text::directory $db_path\n"
    };

    my ($calc, $first, $last) = GT::Tools::find_calculator(
     GT::Test->gt_db, 'TX', $PERIOD_5MIN, 1)
    ok($calc);

    my ($q, $calc, $first, $last) = GT::Tools::calculator(
     GT::Test->gt_db, 'TX', $DAY, 1)
    ok($calc);

=head1 DESCRIPTION

This module provides helper functions for writing GT tests.

=cut

my ($gt_options, $gt_db);

sub import_extra {
    my ($class, $args) = @_;
    $class->setup(@$args);
    Test::More->export_to_level(2);
}
```

```
sub setup {
    my ($class, %args) = @_;
    my ($test_base) = $FindBin::Bin =~ m{(.*/t/*)};
    my $dir = File::Spec->catdir($test_base, 'data');
    $class->prepare_gt_conf($dir, $args{gt_config}->($test_base));
    GT::Conf::load($class->gt_options);
}

sub gt_options {
    my ($class) = @_;
    $gt_options ||= File::Temp->new;
}

sub gt_db {
    $gt_db ||= GT::Eval::create_db_object();
}

sub prepare_gt_conf {
    my ($self, $db_path, $gt_config) = @_;
    open my $fh, '>', $self->gt_options or die $!;
    print $fh $gt_config;
}

1;
```