Scheduling Subsystem Design Document

2007-06-04

Notes: Because of the limits of Word, or perhaps the incompetency of the person writing this document, there was no method of separating the solid arrow that typically represents containment in UML diagrams from the hollow arrow that typically represents inheritance. The author has instead taken the following liberties:

Inheritance: Purple Arrow

Containment: Blue Arrow

1. Major Runtime Components of the Scheduling Subsystem

The scheduling subsystem for EVLA software consists of several parts.

The central core of the subsystem is the dynamic scheduler. The external subsystems used by the dynamic scheduler include:

- A scheduling database repository containing scheduling blocks. The database is used to obtain appropriate blocks for scheduling, obtain scheduling blocks that satisfy various search criteria, and persist scheduling blocks when they have been changed or scheduled.

- Evlamon database for obtaining wind information.

- /home/miranda/cactus/weather for obtaining atmospheric phase interference information.

- A system of files for logging scheduling events, overriding default behavior of the scheduler, storing observe files, and maintaining state.

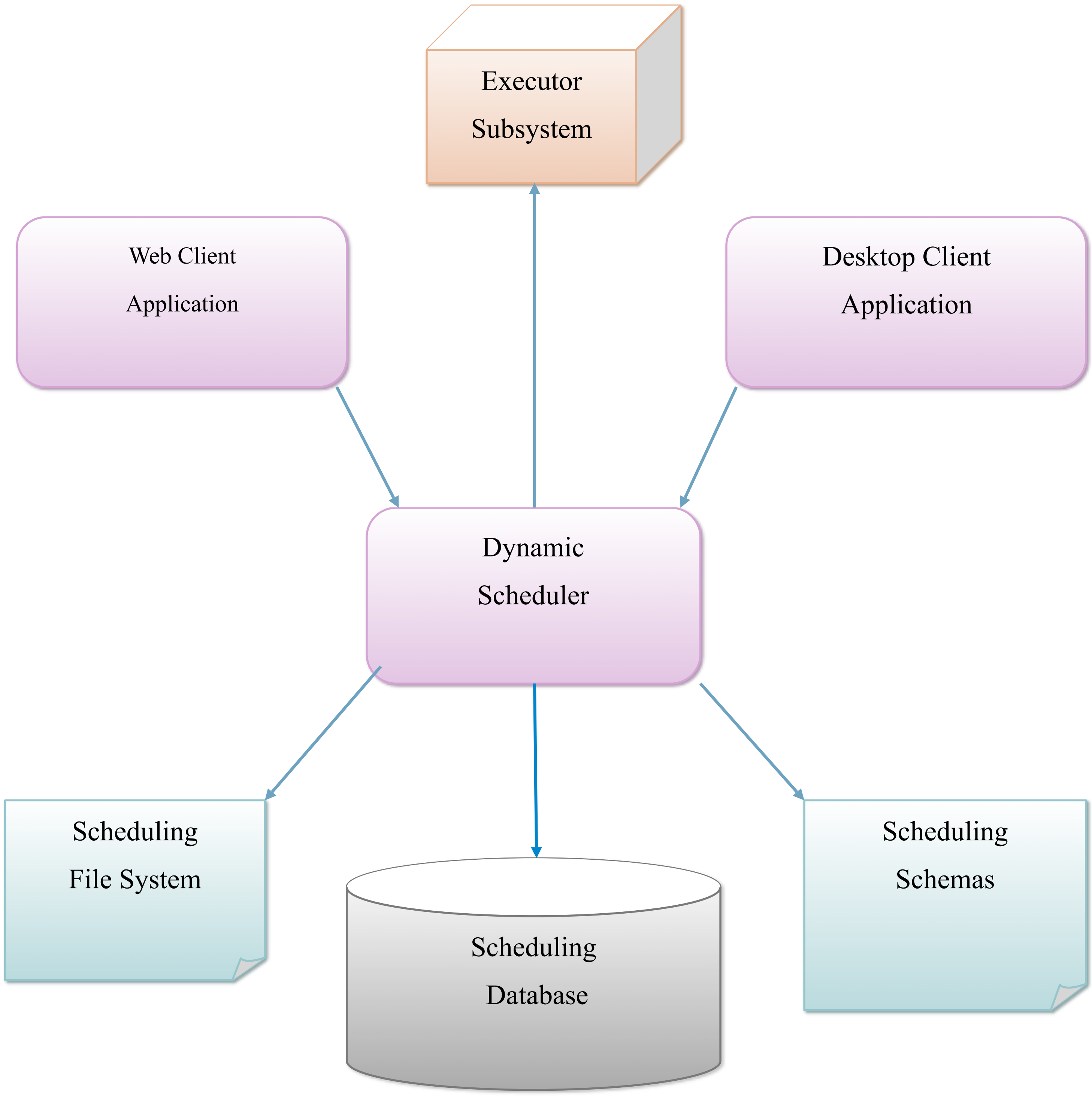
- A collection of xml schemas that define the communication between the dynamic scheduler and the desktop client application.

- A collection of web pages that display information about the dynamic scheduler.

- A desktop application that obtains information from the dynamic scheduler and issues commands to the dynamic scheduler.

- The executor subsystem which receives the observe scripts sent by the dynamic scheduler.

For the most part the dynamic scheduler is shielded from the other components within the EVLA subsystem. The only major EVLA subsystem that the dynamic scheduler communicates with is the Executor. At appropriate times, the dynamic scheduler sends the next scheduling block to the Executor for execution. This information is sent using the EvlaClient software via an http connection and conforms to a specific format.



2. Evolution of the Scheduling Subsystem

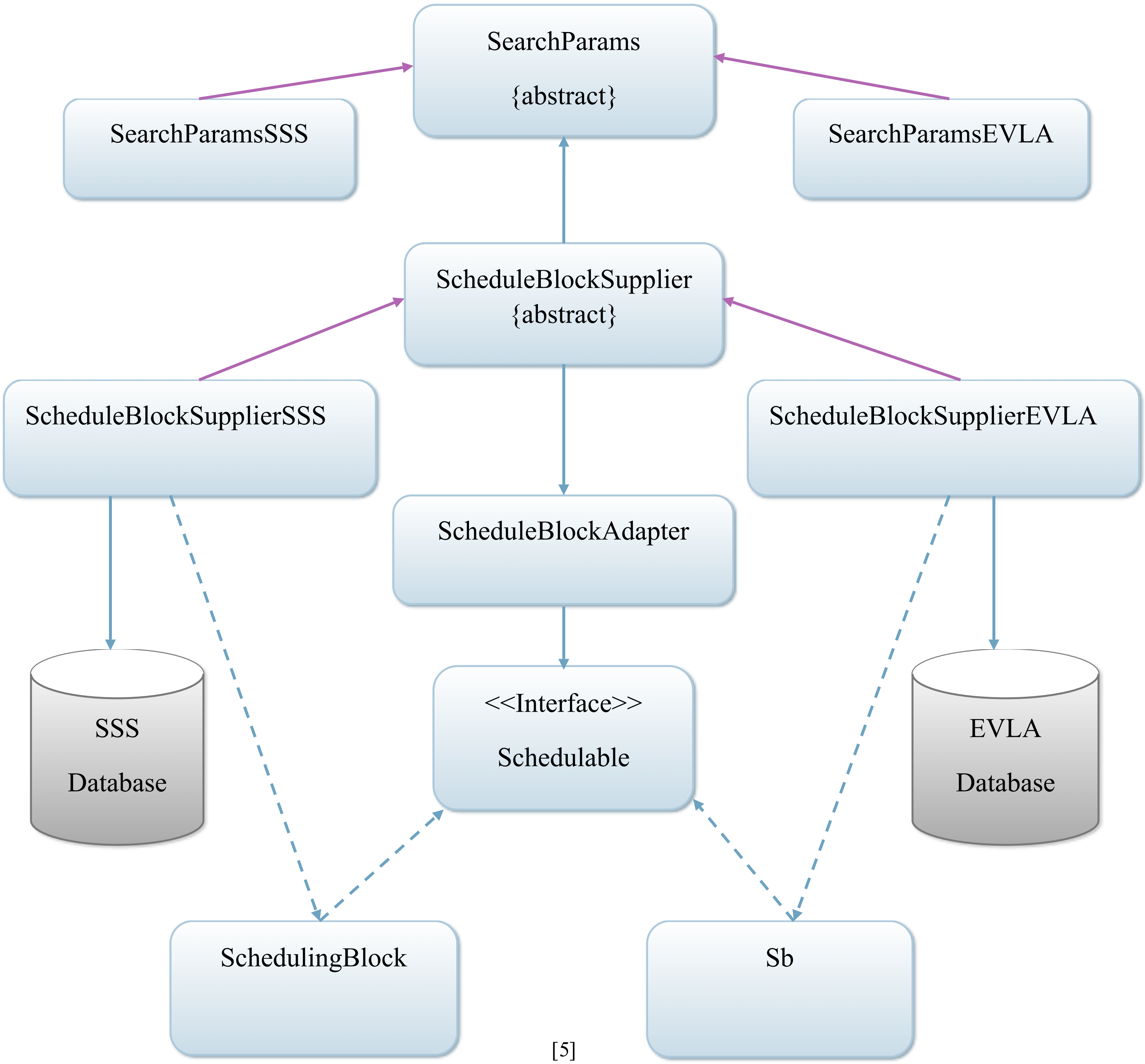
The EVLA subsystem has a dynamic scheduler in place that was designed by Barry Clark. The current dynamic scheduler codebase will be referred to as the EVLA system. At the same time, a new code base is being developed which is more extensible and powerful, and it is anticipated that at some future point in time, the dynamic scheduler will be using this new codebase rather than the legacy system. The new code base will be referred to as the SSS codebase. The dynamic scheduler described in this document is designed to be a bridge and to be able to function with scheduling blocks from either code base.

Most of the dynamic scheduling development work to date has been on developing the dynamic scheduler to run with the EVLA scheduling blocks as this is the first deployment goal for the dynamic scheduler. As the SSS codebase is developed, more scheduling functionality will be developed, and future development work will focus on using scheduling blocks from the SSS codebase.

The dynamic scheduler only deals directly with abstract and wrapper classes rather than implementation classes. Hence, it can be run with either codebase. The java code within the dynamic scheduler can be changed to instantiate either an SSS schedule block supplier or an EVLA schedule block supplier, and that determines which implementation the dynamic scheduler will use at runtime.

The eventual goal with the dynamic scheduler is to have the user interface be part of the ETS software system that is being written by Rich Moeser. With that goal in mind, the scheduler desktop application uses several of the ETS custom panels in order to mimic the look-and-feel of the ETS system as closely as possible. In addition, colors, fonts, and borders have been selected to make the integration of the two systems as seamless as possible.

Support for SSS and EVLA Codebases in the Dynamic Scheduler



3. Dynamic Scheduler Modes

There are two different operating modes for the dynamic scheduler, manual and automatic. It is anticipated that the dynamic scheduler will initially be deployed in manual mode. If a scheduling strategy can be found that appears to be satisfactory, it may be migrated to automatic mode.

3.1 Manual Mode

- Intervention of a telescope operator is required in order for scheduling blocks to be accepted and sent to the executor.

3.2 Automatic Mode

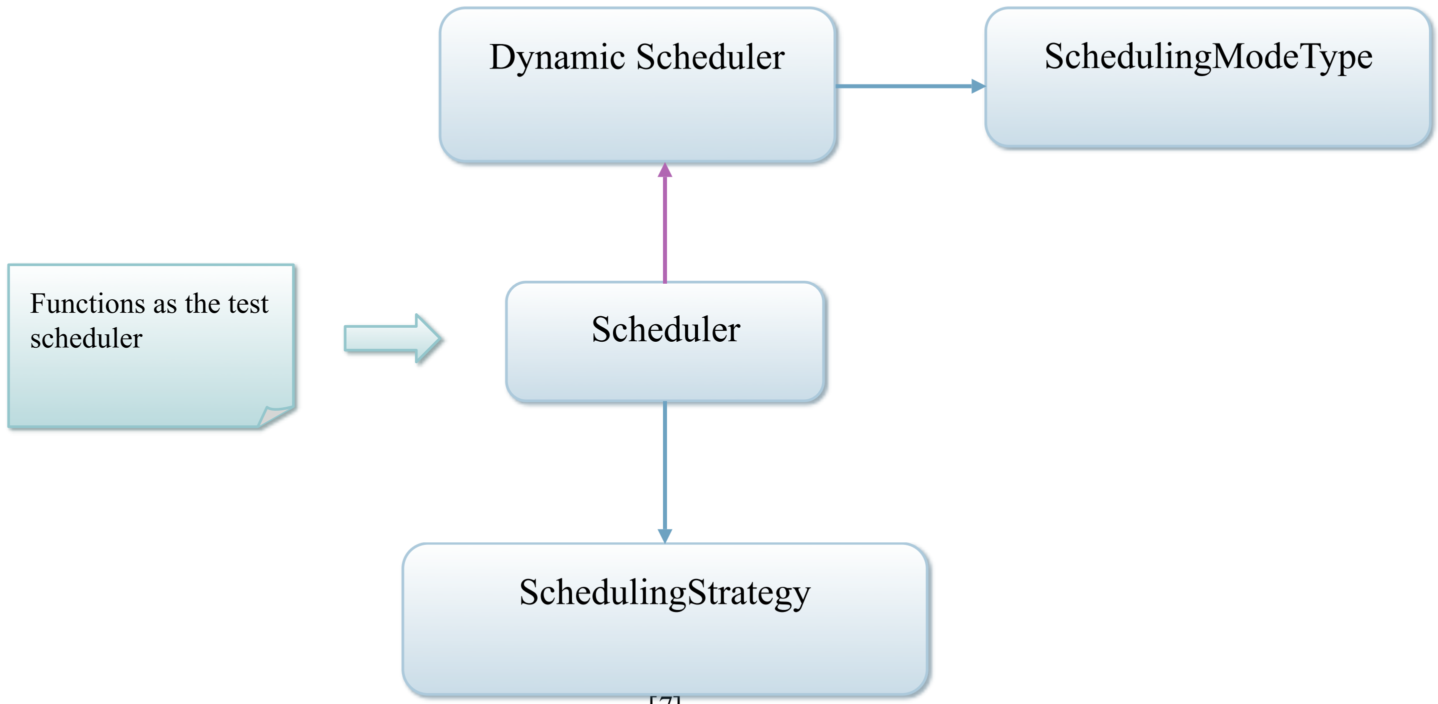
- Responsibility for sending scheduling blocks to the executor rests with the dynamic scheduler.
- Overrides by a telescope operator are still possible when the dynamic scheduler is running in automatic mode.
- Dynamic scheduler runs in a thread; just before a new scheduling block is needed by the executor, the dynamic scheduler wakes up, attempts to obtain the latest weather information, computes a new schedule, and selects the first entry in the schedule to send to the Executor.

4. Test versus Runtime Scheduler

There are two different schedulers that can be run. The first is a test scheduler that is designed for comparing algorithms and testing algorithms. The second is the dynamic scheduler which is designed to be deployed in a real time system. Both schedulers use the same codebase with minor differences described below.

- Test Scheduler
  - Code uses stubs so that experiments do not affect "real" scheduling.
  - Can use and display several scheduling algorithms at once for comparative purposes.
  - Only operating mode supported is manual mode.
- Dynamic Scheduler
  - Code performs real-time scheduling
  - Can only use and display one scheduling algorithm.
  - Supports two operating modes, manual mode and automatic mode.

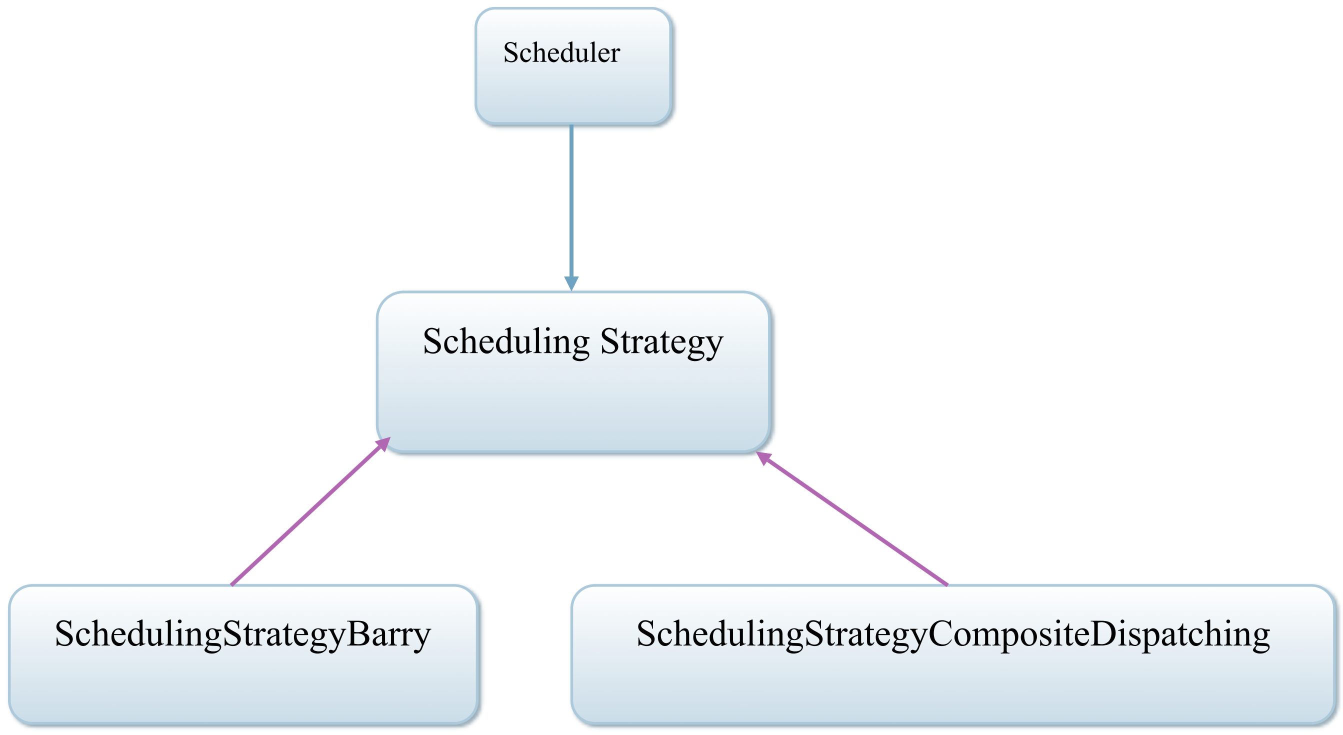
At the time of this writing, the test scheduler has not received as much work as the dynamic scheduler. It was felt that the major coding emphasis should be in getting the dynamic scheduler deployed with the existing scheduling algorithm developed by Barry rather than in experimenting with possible alternative scheduling algorithms.



5. Scheduling Strategies

There are currently two scheduling algorithms, or strategies, that have been developed. The first is implemented in the java class, SchedulingStrategyBarry.java, which is named in honor of the originator of the algorithm, Barry Clark. The second algorithm is implemented in the java class, Scheduling StrategyCompositeDispatching. It is based on a textbook example which involves ranking the scheduling blocks according to configurable criteria such as priority or duration, ranking the scheduling slots in which the blocks can fit by usage, and then trying to fit high ranking scheduling blocks into hard-to-fit slots. Further scheduling strategies can be added as they become available by inheriting from the scheduling strategy base class.

The dynamic scheduler uses only one scheduling algorithm, and that algorithm is configured in the constructor of the class. The test scheduler can potentially run with several scheduling algorithms, and the results of these algorithms can be compared. It is anticipated that this feature may be useful in testing, refining, and evolving scheduling algorithms. Below, is a diagram illustrating how the algorithms fit into the schedulers.



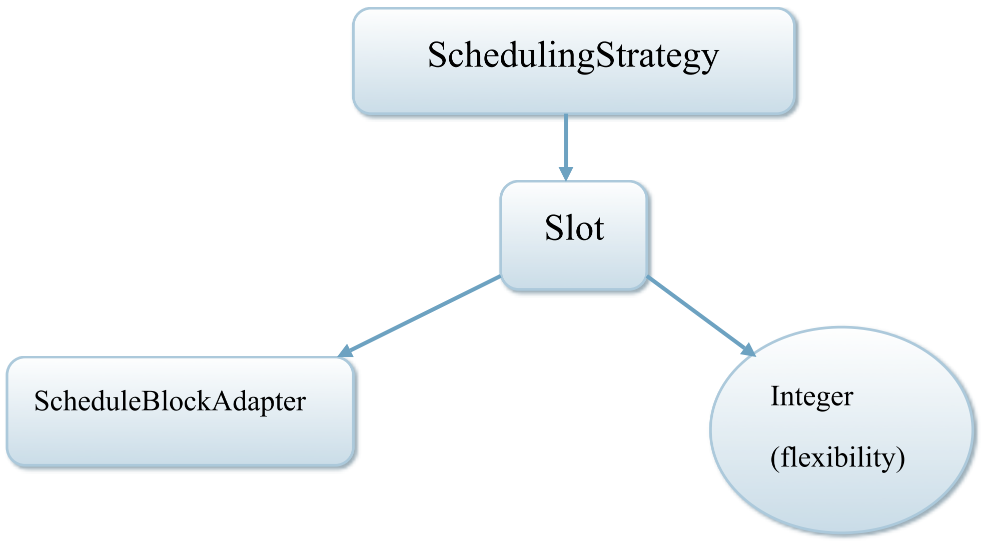
6. Slots

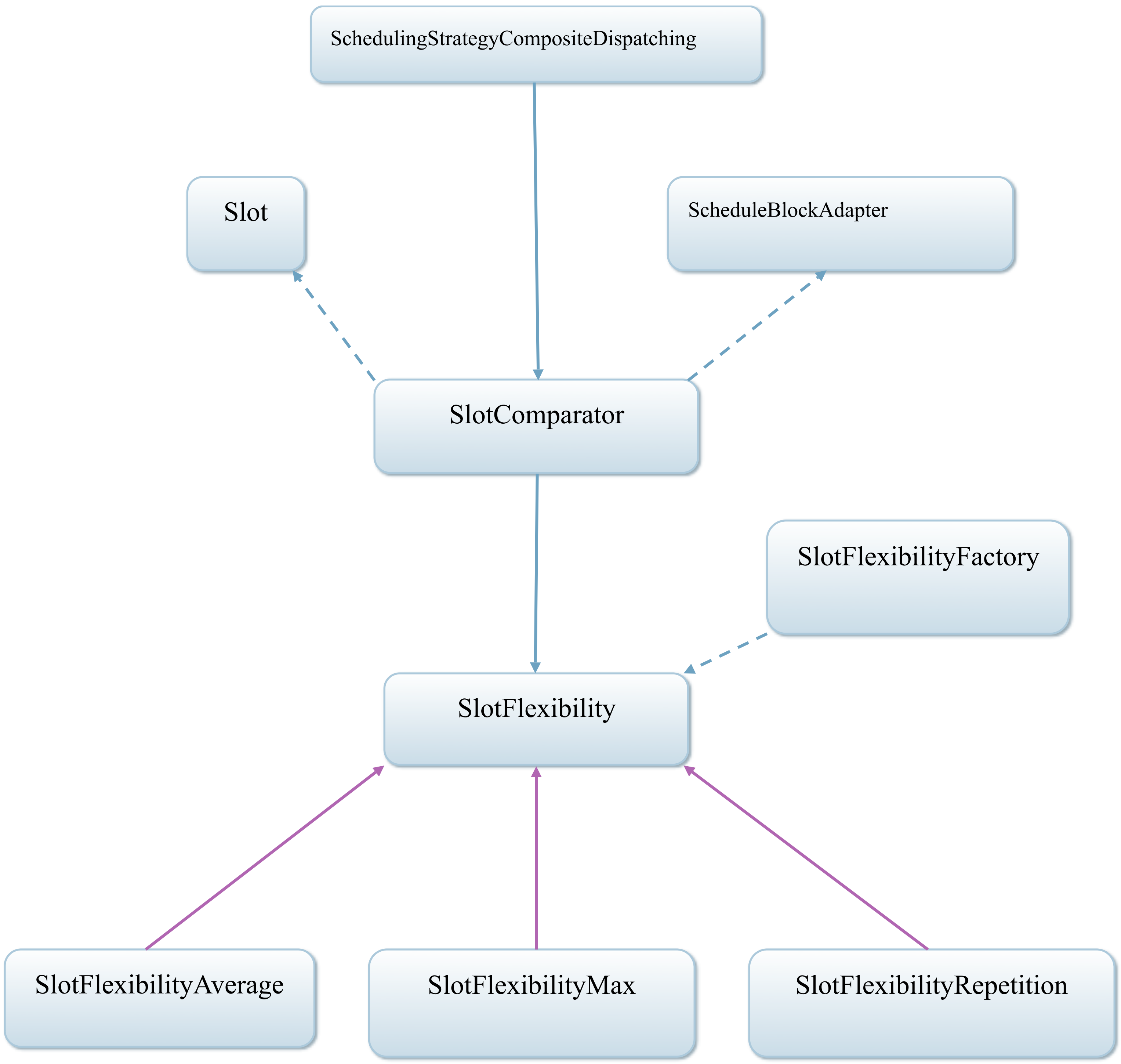
Scheduling algorithms used by the dynamic scheduler work by fitting scheduling blocks into discrete units of time modeled by Slots. The length of the time slots are configurable; however a standard time slot of one half hour is now being used by the current dynamic scheduler. As the dynamic scheduler computes a schedule, it fills each Slot with the scheduling block that will be run during that unit of time.

The slot package contains additional functionality that is specific to the composite dispatching algorithm. This algorithm takes each specific time slot and measures how many scheduling blocks can be fit into the slot in order to get a measure of its flexibility. When the algorithm decides to fit a scheduling block it looks at all of the possible slot ranges where the block might fit (taking into account things like preferred LST), and it chooses the slot range with minimum flexibility. The flexibility of a slot range can be measured in a variety of ways; subclasses of SlotFlexibility model these methods.

- SlotFlexibilityAverage
    - Measures the average of the individual slot flexibilities of the slots spanned by the scheduling block.
- SlotFlexibilityMax
    - Measures the maximum of the individual slot flexibilities of the slots spanned by the scheduling block.
- SlotFlexibilityRepetition
    - Measures the total number of repetitions of the scheduling block that will into the scheduling starting with the first slot in the range.

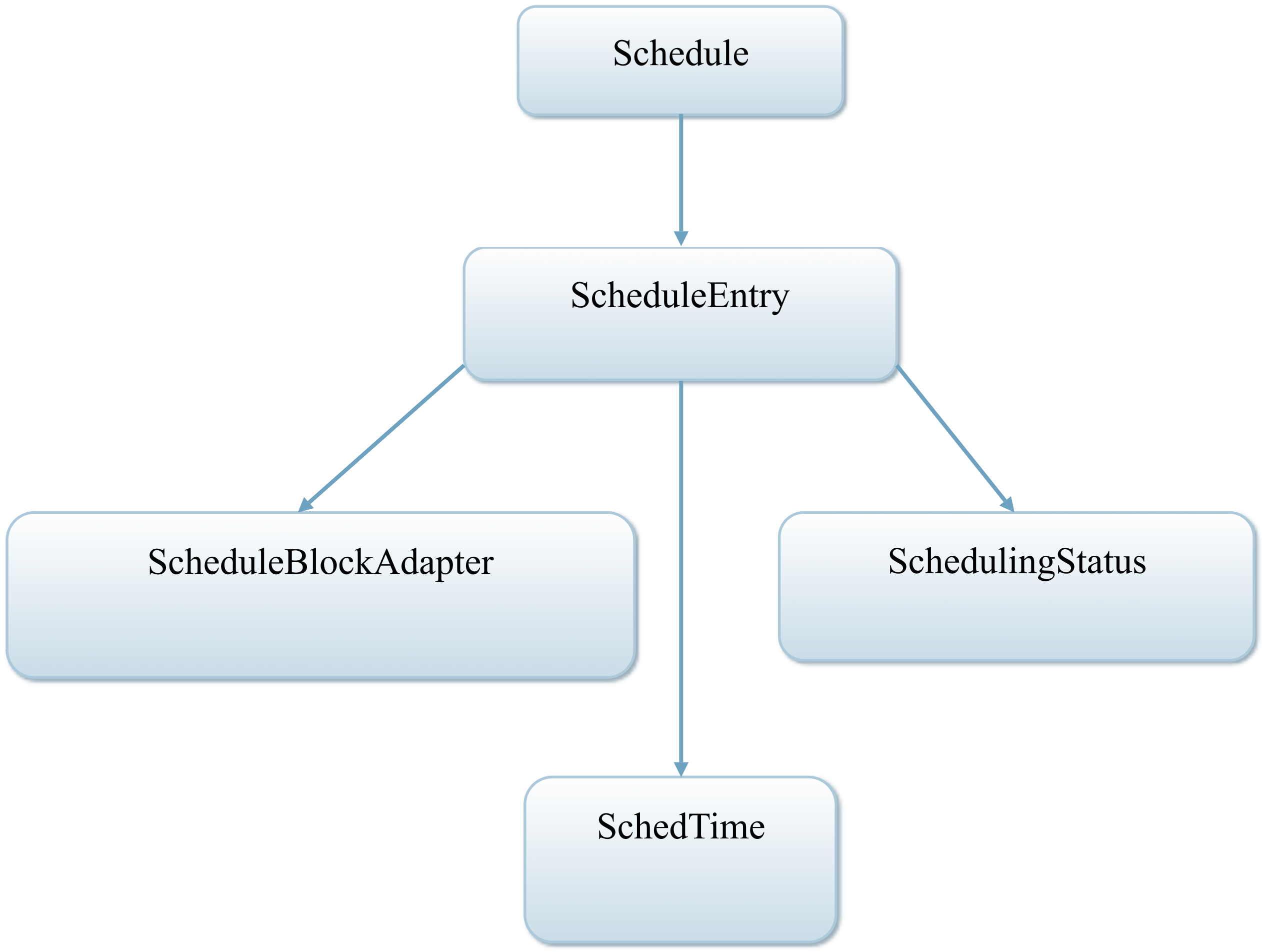
A class diagram of the slot package and how it fits into the dynamic scheduler are given below.




SchedulingStrategyCompositeDispatching
Slot
ScheduleBlockAdapter
SlotComparator
SlotFlexibilityFactory
SlotFlexibility
SlotFlexibilityAverage
SlotFlexibilityMax
SlotFlexibilityRepetition

7. Schedules

Each time a scheduling strategy is run it produces a schedule. A schedule consists of a collection of schedule entries. Each schedule entry contains a reference to the scheduling block that will be sent to the executor when it is accepted for scheduling, a time element indicating when the scheduling block will be sent to the executor, and a status element that is updated as it proceeds from an unscheduled state to a completed state. The time element, represented by the SchedTime.java class, consists of both an LST day number and hour for execution. Normally, the telescope operator will select the first schedule entry in the list to the executor, but the operator also has the option of telling the dynamic scheduler to hold the first entry in the list. If this happens, the dynamic scheduler will compute a new schedule which will no longer include the scheduling block that has been put on hold. A schedule block can later be released after being put on hold at the discretion of the telescope operator. A schedule entry can also be cancelled, which indicates that the entry will not be considered for scheduling again. The diagram illustrates the structure of a schedule.

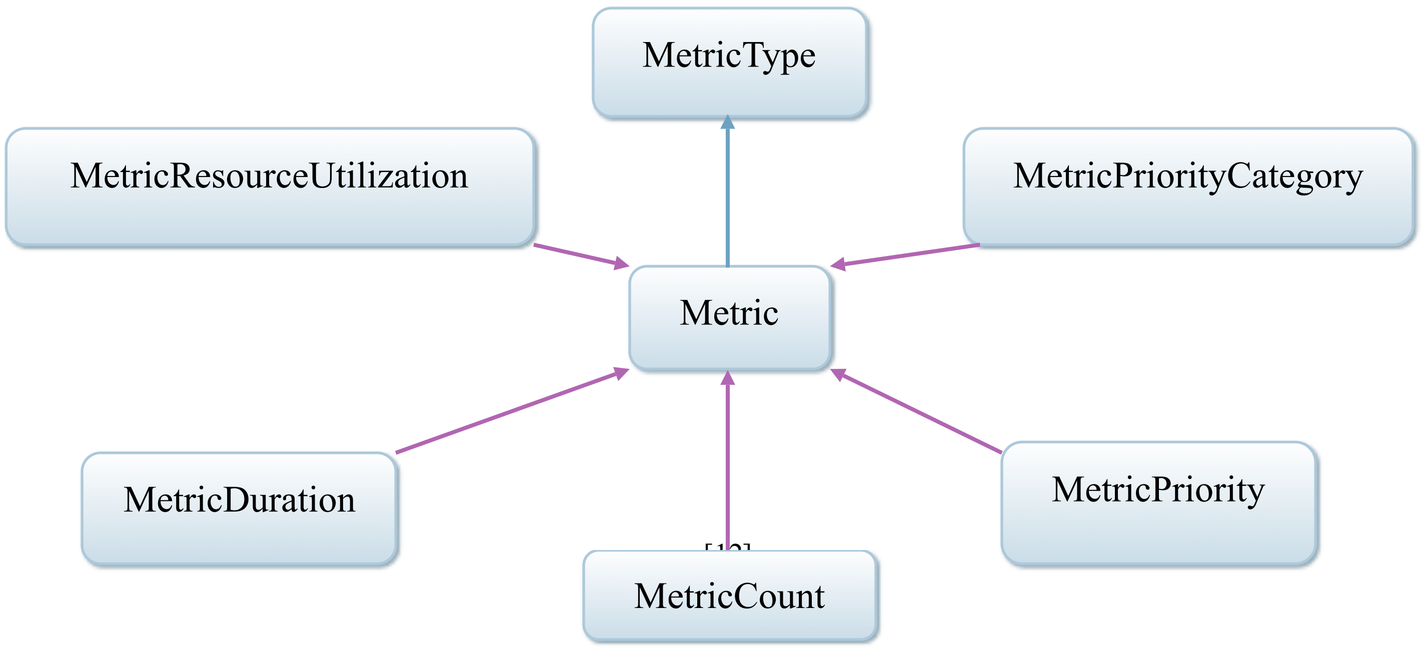


8. Measuring Schedules

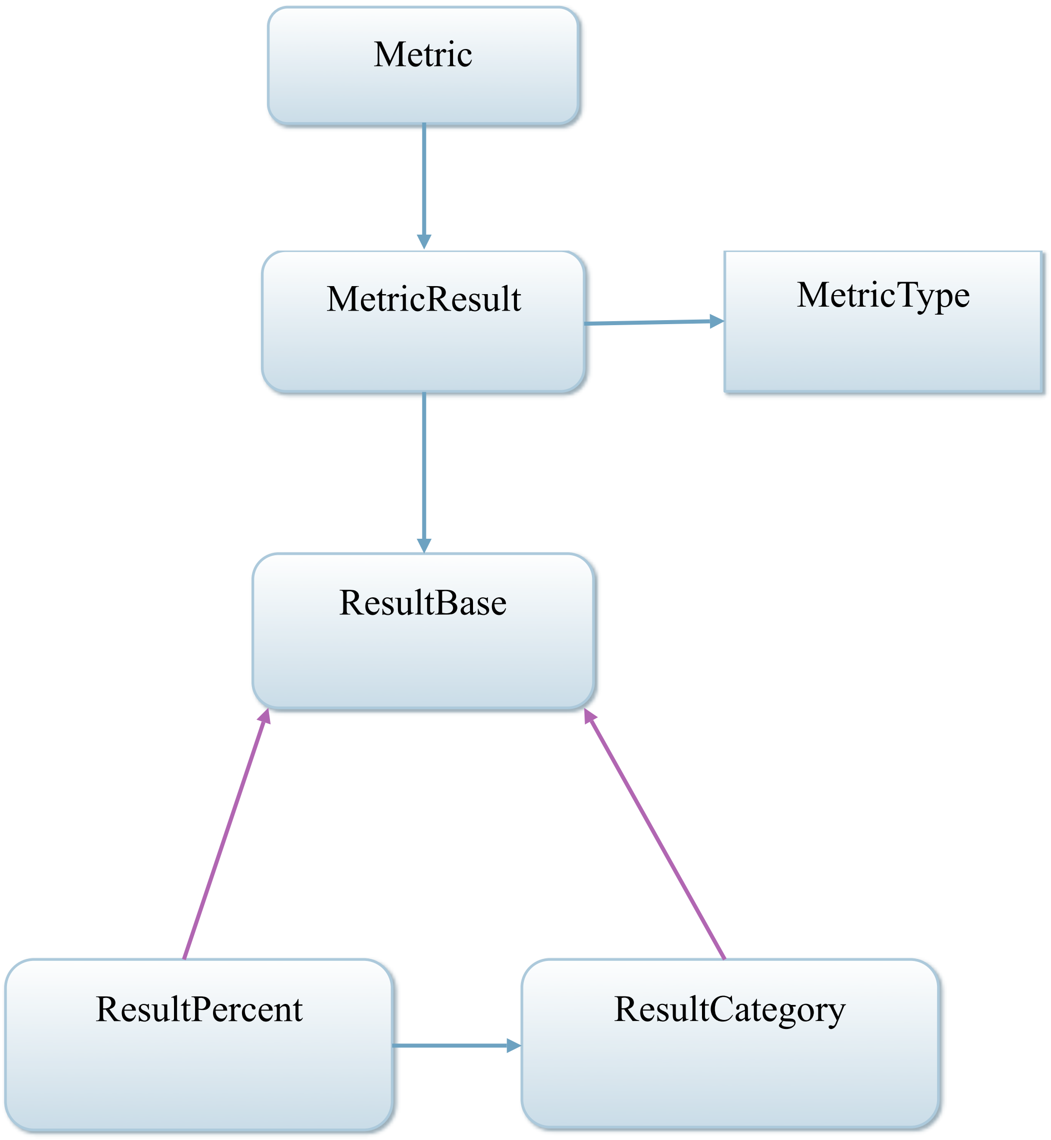
Various metrics have been written for measuring the effectiveness of schedules. They are displayed in the user interface in the form of graphs and pie charts. An infrastructure is in place for adding further measurement capabilities should the need arise. At the point of this writing the metrics that have been developed include:

- MetricCount
    - Measures how many of the available scheduling blocks were fit into the schedule.
- MetricDuration
    - Classifies the scheduling blocks according to their length and measures how many of in each length category are scheduled compared to the number in each length that are available to be scheduled.
- MetricPriority
    - Measures the total priority of the the scheduling blocks that were scheduled compared to the total priority of the scheduling blocks that were available for scheduling.
- MetricPriorityCategory
    - Classifies the scheduling blocks according to priority and measures how many in each priority are scheduled compared to the number in each priority category that are available to be scheduled.
- MetricResourceUtilization
    - Measures how much of the available time was filled by the schedule.

The structure of the metrics package is indicated below.

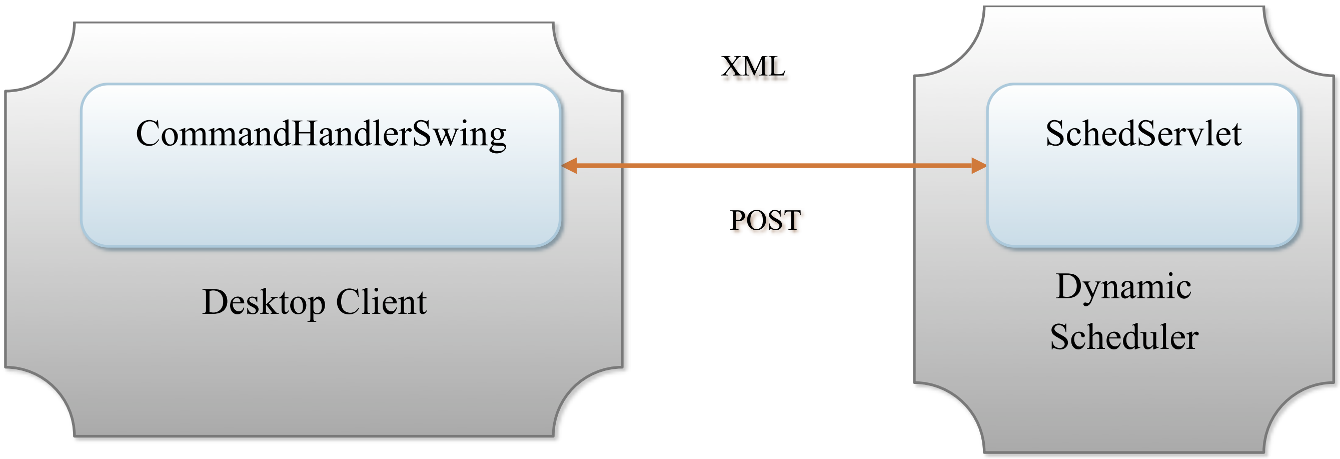


A metric produces a MetricResult, which includes information about which type of metric produced it and the result of applying the metric. There are two types of results that a metric may produce. The first is just a basic percentage, modeled with the ResultPercent class. The second type of result occurs with the scheduling blocks are categorized according to some criteria and the metric produces a percentage result for each category. This second type of result is modeled with the ResultCategory class. The structure of the metric results are indicated in the diagram below:



9. Communication between the Dynamic Scheduler and the Desktop Client

The dynamic scheduler runs on a web server. There is a main servlet in the scheduler, called SchedServlet, that listens for requests for information and directions or commands from the desktop application. Communication via the desktop application is based on xml sent back and forth in POST requests. On the client side, CommandHandlerSwing is responsible for sending requests to the servlet and receiving responses from the servlet.



The protocol for obtaining information from the dynamic scheduler or invoking commands on the dynamic scheduler from the desktop client is encapsulated in the Command package. For each message there is a corresponding Command that inherits from the base Command class. The formalized procedure for sending a command to the dynamic scheduler is as follows:

1. Add any necessarily parameters to the command that represent input parameters that the dynamic scheduler needs to invoke the command. These input parameters are stored as a list of parameters (Param.java) in the command class.

2. Use JAXB to serialize the command as XML.

3. Send the serialized XML as a POST request to the dynamic scheduler.

4. On the server side, use JAXB to reconstruct the command as a Java object from the XML.

5. Call the "execute" method of the command on the server side to obtain the result of performing the command. The results of the command are stored in a java class called CommandResult.

6. Use JAXB to serialize the CommandResult to the client side.

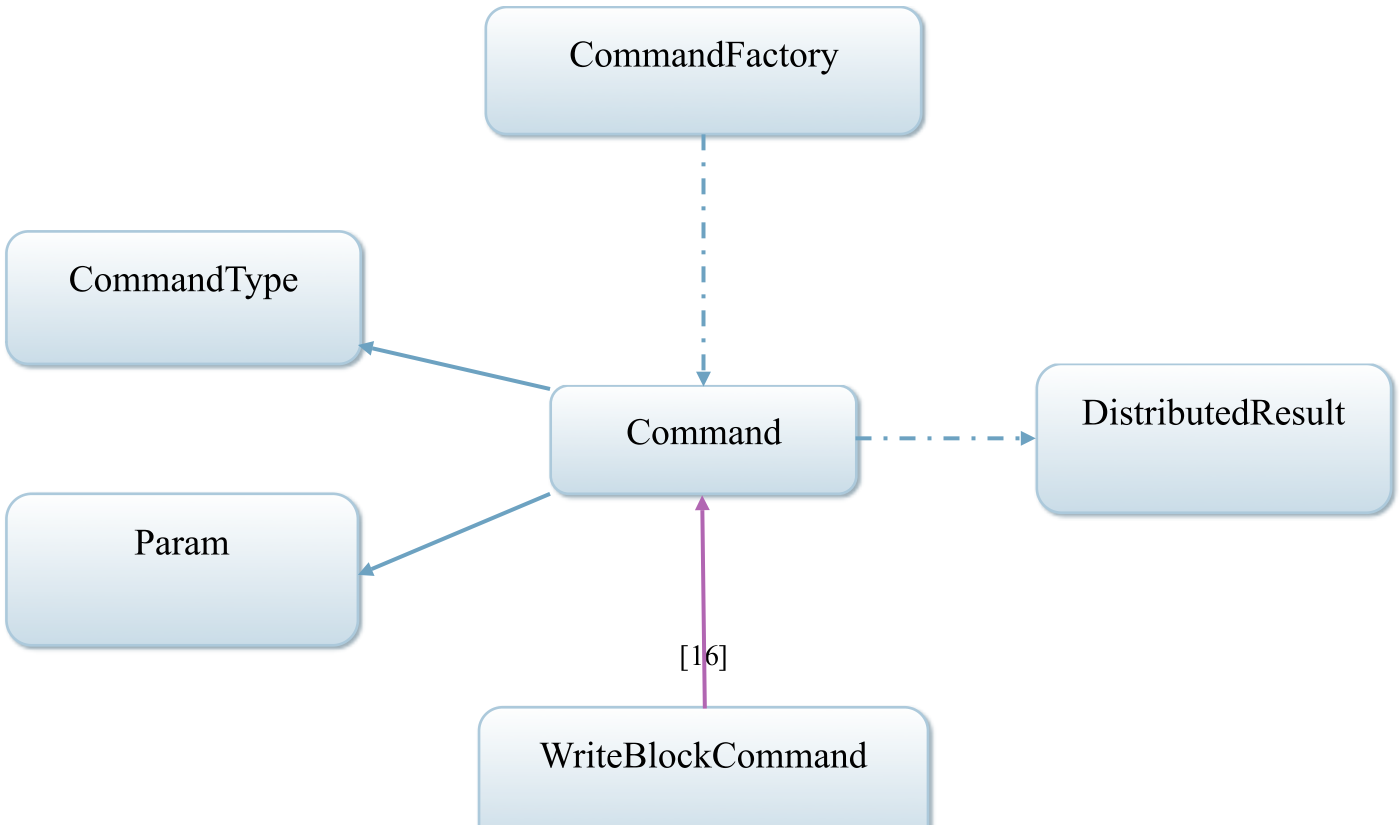
7. Reconstruct the results of the command as java objects on the client side and update the user interface with the results.

The command infrastructure is extensible so that new commands can be added as needed. The commands that are currently supported include:

- BlockInitializeCommand
    - Obtains the scheduling blocks from the database that are eligible to be scheduled.
- CancelBlockCommand
    - Marks a scheduling block in the database as cancelled so that it will never be considered for scheduling again.
- ExecuteScheduleEntryCommand
    - Sends the next scheduled entry to the Executor; computes a new schedule, selects the first entry in the new scheduler as the next block to be sent to the Executor.
- FilterValuesPersistCommand
    - Allows telescope operators to override the constraints such as the wind and atmospheric phase interference in cases where the computed values are missing or invalid.
- GetObserveScriptCommand
    - Retrieves the observe script for a specific scheduling block for display on the user interface. This allows telescope operators to check that the observe script is not missing or invalid.
- HoldBlockCommand
    - Puts a scheduling block on hold so that it will not be considered for scheduling again until it has been explicitly released through the desktop client.
- InitializeEndTimeCommand
    - Allows a telescope operator to specify the ending time for a schedule as opposed to determining it based on the starting time of the next fixed scheduling block.
- InitializeStartTimeCommand
    - Allows a telescope operator to specify the starting time for a schedule as opposed to having it calculated based on the predicted ending time of the last block that was sent to the Executor.
- RefreshSchedulerCommand
    - There is a thread in the desktop application that can be configured to retrieve the latest information from the dynamic scheduler at periodic intervals and refresh the user interface with that information. This is a useful feature when more than one desktop client may be connected to the dynamic scheduler with more than one person changing its runtime state via commands.

- ReleaseBlockCommand
  - Releases a scheduling block that has been put on hold so that it will again be considered eligible for scheduling.
- SaveAntennasCommand
  - Allows the telescope operator to update the list of EVLA antennas that should be sent the observe script command.
- SearchBlocksCommand
  - Provides a database lookup feature, selecting all scheduling blocks from the database that satisfy a specific search criterion.
- UpdateScheduleCommand
  - Directs the dynamic scheduler to calculate a new schedule and select the first entry in the schedule as the next block to be sent to the executor.
- WriteBlockCommand
  - Scheduling blocks are editable through the user interface and this command directs the dynamic scheduler to write the editing changes to the database.

The relationship among the classes in the command package is indicated in the diagram below. For simplicity only the WriteBlockCommand has been included in the diagram. The other supported commands would have identical relationships.



10. Authentication and Authorization

The login package in the desktop application contains the infrastructure for authentication and authentication.

10.1. Authentication

Authentication is required within the dynamic scheduler because the information that is sent to the Executor when someone executes a scheduling block includes the identity of the person who initiated the command.

The dynamic scheduler shares a common logon GUI with the ETS software called ETS Logon. Once a user has entered a username and a password, the SchedulerLoginService is notified. This service extends existing authentication software in swingx and implements a template method for performing authentication. This method performs the following actions:

1. Receives a username and password from the ETSLogon
2. Sends a request containing the name and password to the the AuthFilter running on the server side.
3. The AuthFilter queries the user database for a User corresponding the name and password combination.
4. The AuthFilter serializes the User as xml and sents the xml back to the client side.
5. The SimpleLoginService reconstructs the User from the corresponding xml.

10.2 Authorization

Authorization is a requirement for the dynamic scheduler because the client software provides access to sensitive information such as changing (or deleting) scheduling blocks contained in the scheduler database. Various GUI features of the desktop client should be enabled/disabled based on authorization level.

The infrastructure for authorization has been developed within the dynamic scheduler, but further work needs to be done to implement it. A class, GroupType, defines the authorization groups for the dynamic scheduler. At the time of this writing, these groups were not yet supported in the user database. Once these groups have been added, and various people assigned to them, the web desktop has the ability to enable/disable components based on the authorization level of the user. A future item of work is deciding exactly which scheduler functionality should be available at each authorization level. Once this is done, it will be possible for the developer to configure the widgets with an appropriate minimum authorization level. This will allow different functionality to be available at different authorization levels.

11. Priorities

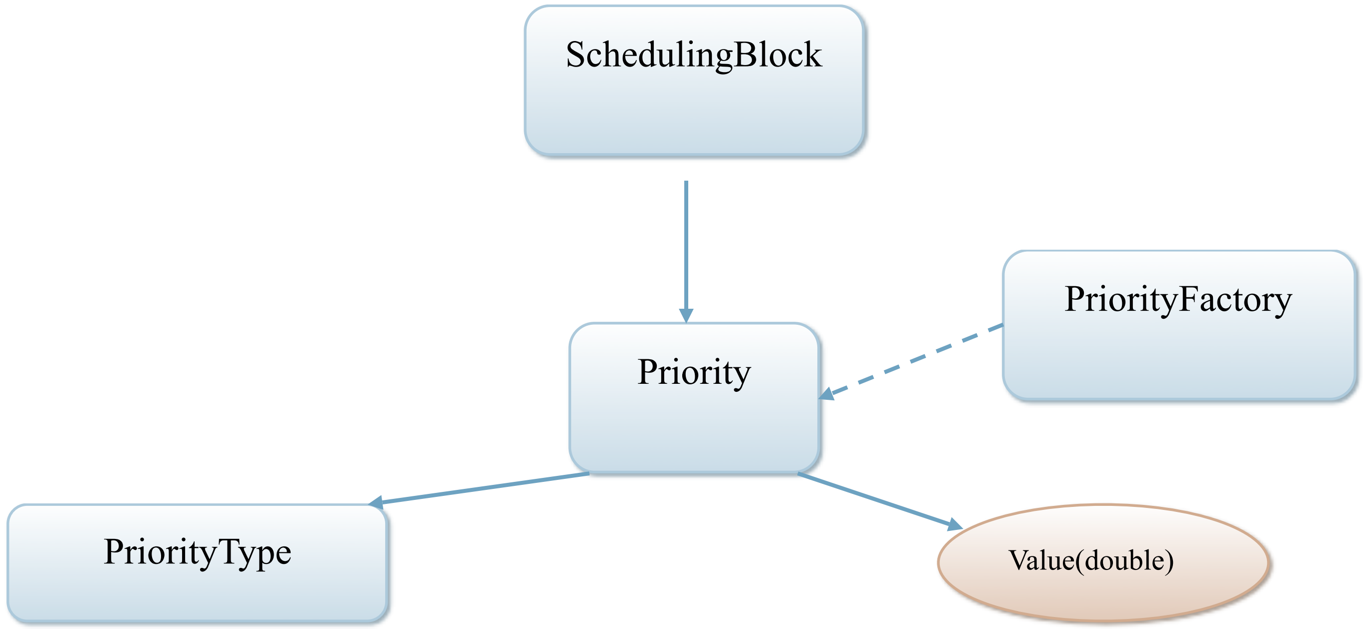
Priorities are measures of the merit assigned to a scheduling block by a person, committee, or process. Scheduling blocks have priorities of various types assigned to them by different sources. Scheduling algorithms then typically rank scheduling blocks based on a summary of their priorities. Specifically, the algorithm designed by Barry Clark uses an average of the various priorities to rank scheduling blocks for scheduling. The composite dispatching strategy is configurable with regard to how it ranks scheduling blocks, and priority is one of many criteria that can be used.

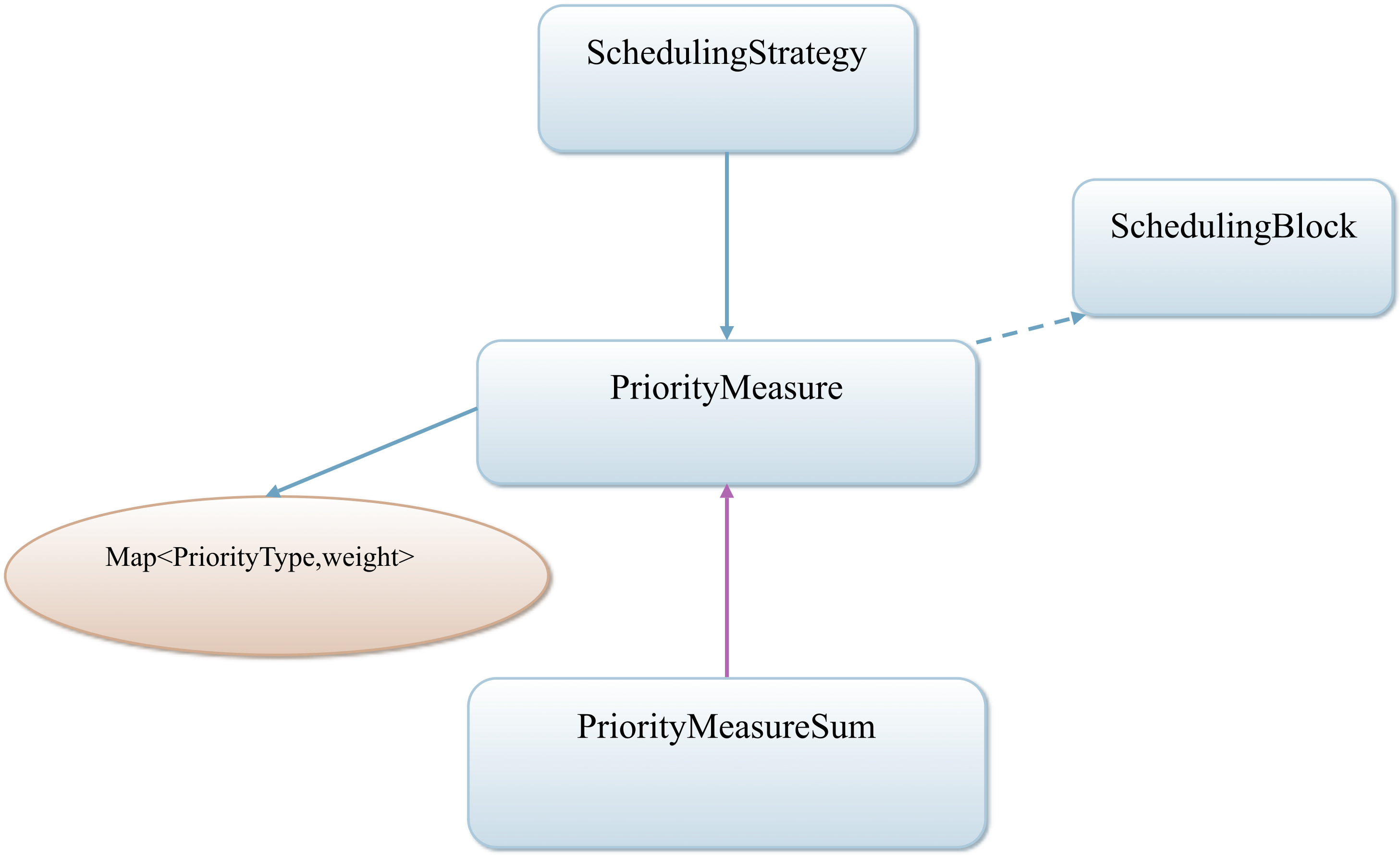
The dynamic scheduler currently supports the following scheduling block priorities:

- Mean Referee Priority
    - A normalized average of the referee rankings
- Proposal Selection Committee Priority
    - The mean proposal selection committee ranking.
- Software Priority
    - A priority that can be used by the software to favor particular types of scheduling blocks. For example, it has been used in the past to give a slight preference to longer scheduling blocks.
- Scheduler Priority
    - A priority reserved for the scheduler's use. In the past it has been used to reflect a particular urgency with regard to completing the scheduling block.
- Project Priority
    - A relative priority within a particular project that indicates the importance of completing the scheduling block compared to other scheduling blocks within the project.
    - As of this writing, a persistence method has not been written for this type of priority.

Priorities have a value and a type (one of those listed above). All of the priorities within a scheduling block need to be combined to form a single measure of the priority of the block that t scheduling algorithms can use to rank the blocks. A PriorityMeasure is a method of combining priorities to obtain a single number. Priority measures can be configured with a weight for each priority type. In this way, a priority measure can choose to ignore one of the priority types or favor one of the types over the others. Currently, only one PriorityMeasure has been developed in the dynamic scheduler, a PriorityMeasureSum, which uses a weighted average of the priorities.

The class diagrams below shows the structure of the priority package within the dynamic scheduler.


SchedulingBlock
PriorityFactory
Priority
PriorityType
Value(double)


SchedulingStrategy
SchedulingBlock
PriorityMeasure
Map<PriorityType,weight>
PriorityMeasureSum

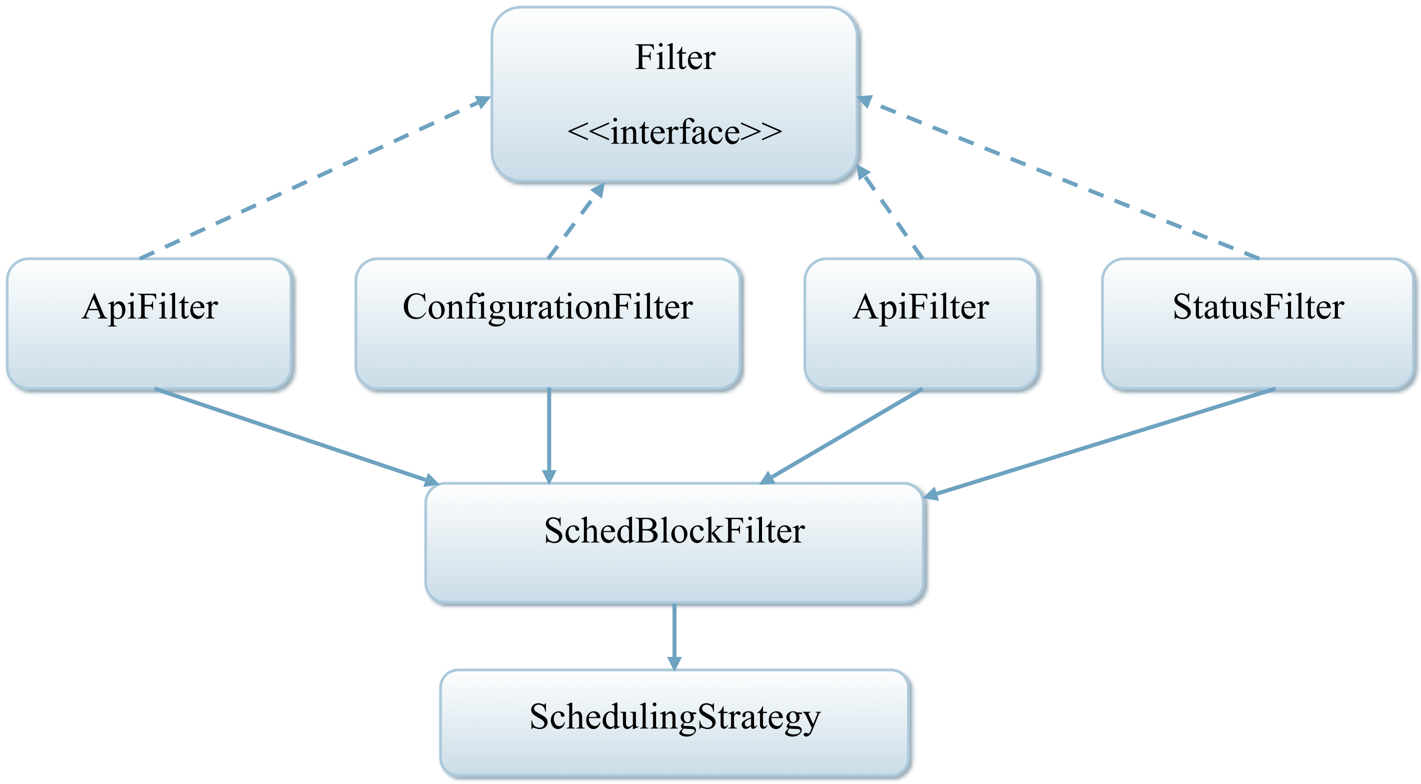
12. Filters and Constraints

Filters and constraints both represent conditions which affect the desirability of a scheduling block for inclusion in a schedule. In some instances, external condition can be represented by both a constraint and a filter, depending on the usage. We discuss the differences between constraints and filters in the subsections below.

12.1 Filters

- Represent yes/no conditions with regard to the suitability of a scheduling block for scheduling. Specifically, the output from a filter is a FilterResult which contains a Boolean variable indicating the whether the scheduling block is allowed and (in the case of rejection) an indication as to the reason for the rejection.
- Used to eliminate scheduling blocks from the list of blocks eligible for scheduling.
- Scheduling blocks do not have filters, instead scheduling algorithms are configured with filters. The filters use the constraints within the scheduling blocks to thin out the list of blocks that should be scheduled.

A class diagram indicating how the filter package fits into the dynamic scheduler is given below.



12.2 Constraints

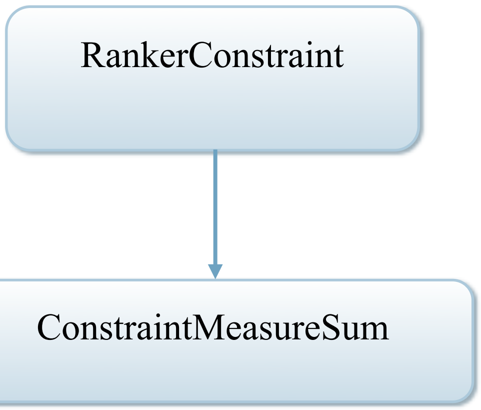
- Represents a “gray” area with regard to the suitability of a scheduling block for scheduling. More specifically, a constraint estimates the probability of successful completion of a scheduling block under the condition represented by the constraint and a ConstraintProbability is returned which represents this probability.
- Used to prioritize scheduling blocks with regard to which blocks should be considered first for scheduling.
- Scheduling blocks have constraints which indicate their limitations with regard to particular external conditions.

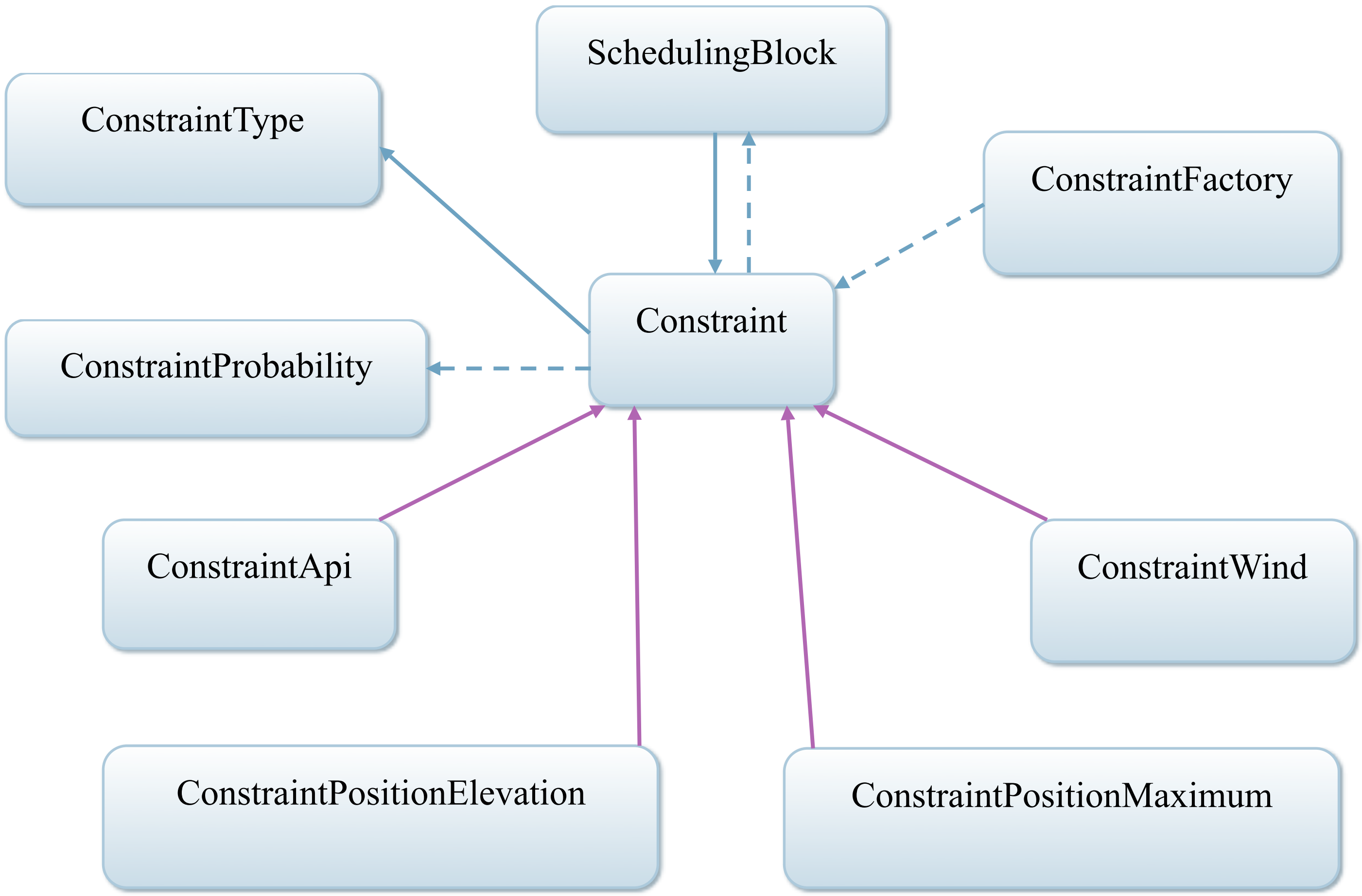
At the time of this writing, constraints are not as well developed as other areas of the dynamic scheduler. While a template method is in place for determining the probability of success of a scheduling block under a constraint, the code for determining this probability is not developed for any of the constraint objects. In addition, two of the classes, ConstraintPositionElevation and ConstraintPositionMaximum are only places holders, representing possible future development work.

Constraints work in much the same way that priorities do. A scheduling block contains a collection of constraints. The probability of success of a scheduling block under each constraint is combined into a single number using a weighted sum of the individual probabilities through the ConstraintMeasureSum class.

The scheduling algorithm developed by Barry Clark does not use constraints. The composite dispatching algorithm can be configured to rank scheduling blocks using the probability of successful completion computed from the ConstraintMeasureSum class as one of its criteria through a Ranker. Rankers will be discussed in the next section.

Class diagrams showing how constraints fit into the dynamic scheduler are given below.


RankerConstraint
ConstraintMeasureSum


SchedulingBlock
ConstraintType
ConstraintFactory
Constraint
ConstraintProbability
ConstraintApi
ConstraintWind
ConstraintPositionElevation
ConstraintPositionMaximum

12.3 Difference between Constraints and Priorities

In many ways, the usage of priorities (discussed in the last section) is similar to the usage of constraints, i.e., priorities are also used to rank scheduling blocks. In the present design of the dynamic scheduler , the decision was made to represent them as different entities due to the differences in meaning and behavior between them. These differences are summarized below:

- The desirability of a scheduling block with respect to a particular external condition is expected to vary naturally over time. Priorities tend to remain fixed. In rare instances, they can be altered by a person or process, but that takes explicit intervention on the part of a person or explicit piece of code.
- Priorities do not measure the chances of successful completion of a scheduling block under specific operating conditions as do constraints and filters. Instead priorities represent an external evaluation of the merit of the proposal.

13. Rankers

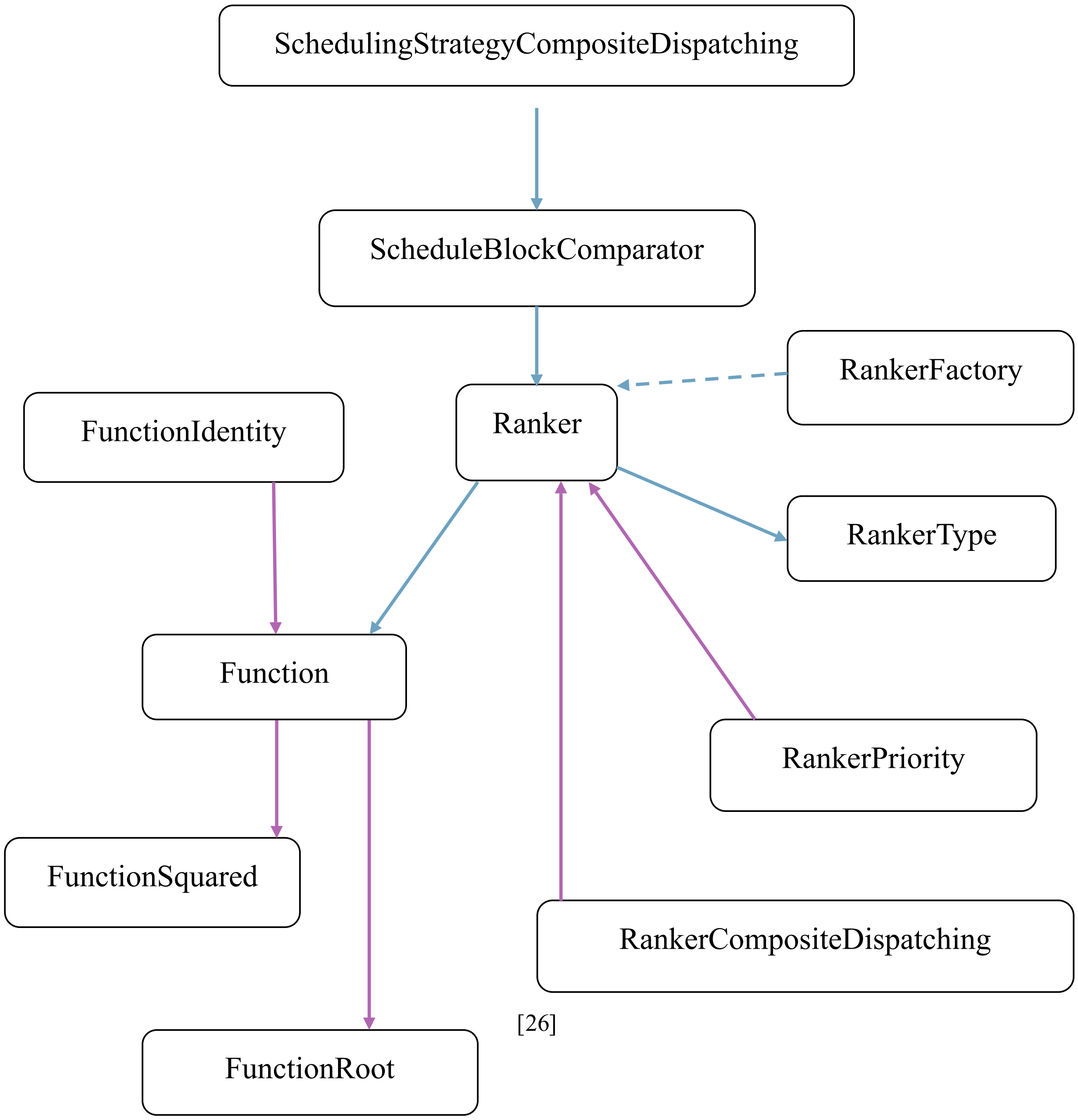
The capabilities available in the dynamic scheduler's rank package are only used with the composite dispatching scheduling algorithm. A Ranker takes a scheduling block as an input and computes a number which represents that scheduling blocks score with respect to the criteria that the ranker is using to evaluate scheduling blocks. Rankers contain weighting functions that are used when they are combined with other rankers to form a composite. Weighting functions that have been developed include a root function, a squaring function, and the default identity function. There are a variety of rankers available for use with the dynamic scheduler:

- RankerCompatibleResources
    - Ranks scheduling blocks based on the idea that resources should be conserved by scheduling "similar" blocks at the same time.
    - Stub Only – needs further development work
- RankerConstraint
    - Ranks scheduling blocks based on the probability of their successful completing based on current operating conditions
- RankerNearestToEndLST
    - Ranks scheduling blocks based on the idea that scheduling blocks which will be out of their preferred date range first should be scheduled before blocks that will remain within their preferred date range a longer period of time.
- RankerObeservationsRemaining
    - Ranks scheduling blocks based on the idea that once a scheduling block has been started it is desirable to finish all of its repetitions within some reasonable period of time and not have it drag out over many months.
- RankerPriority
    - Ranks scheduling blocks based on a measure of their overall priority.
- RankerSetupTime
    - Ranks scheduling blocks as an inverse function of their calibration and set-up requirements.
    - Stub Only – needs further development work.
- RankerSlotUsage
    - Ranks scheduling blocks based on the idea that scheduling blocks which don't fit into a lot of scheduling slots should be ranked before blocks which are very flexible and can fit into the schedule at a lot of different time slots.

- RankerCompositeMultiplication
    - A means of combining individual rankers through multiplication in order to provide a single composite ranker. This ranker uses the Function contained in the individual rankers to affect the weight of the ranker within the composite.

The composite dispatching scheduling strategy has a schedule block comparator that can be configured with one or more rankers. The comparator then sorts the scheduling block using the rankers that it contains.

A class diagram of the rank package and how it fits into the dynamic scheduler is given below. For simplicity, we have only shown RankerPriority and RankerCompositeMultiplication in the diagram.



14. Persistence

In its current implementation, configuration settings within the dynamic scheduler (such as the scheduling algorithm to use, whether to run in manual or dynamic mode, etc) will be lost if the scheduler is stopped and restarted. However, with just a few minor changes the configuration settings in the dynamic scheduler can be made persistent. The key to this is the dynamic scheduler's ability to write itself as xml and to reconstruct itself from an xml representation. All that would be needed would be for the SchedServlet to read a file containing an xml representation of the scheduler on start-up. Whenever a user saved a configuration change, the dynamic scheduler would need to be persist itself to that file as xml.

15. Weather

Before the dynamic scheduler computes a new schedule, it attempts to obtain the latest weather information through its filters. Currently, the only weather information in use by the dynamic scheduler is the wind and the atmospheric phase interference. Both of these values are persisted in a file called "sched.parm" in the scheduler file system.

15.1 Wind

The WindFilter attempts to obtain a wind reading from a table in the evlamon database. If the WindFiltter cannot find the information from the database for some reason (perhaps a network connection is down), it attempts to read the information from sched.parm. Telescope operators also have the option of writing a wind value in the user interface and saving it to the sched.parm file. This written value will be the value used if the dynamic scheduler cannot obtain a new value from the evlamon database.

15.2 Api

The ApiFilter works in much the same way as the WindFilter, first attempting to obtain a current reading of the atmospheric phase interference and if that fails, then reading the api from the sched.parm file. The dynamic scheduler attempts to connect to "/home/Miranda/phasemon/cactus/weather" in order to obtain an estimate of the atmospheric phase interference.

16. Desktop Client

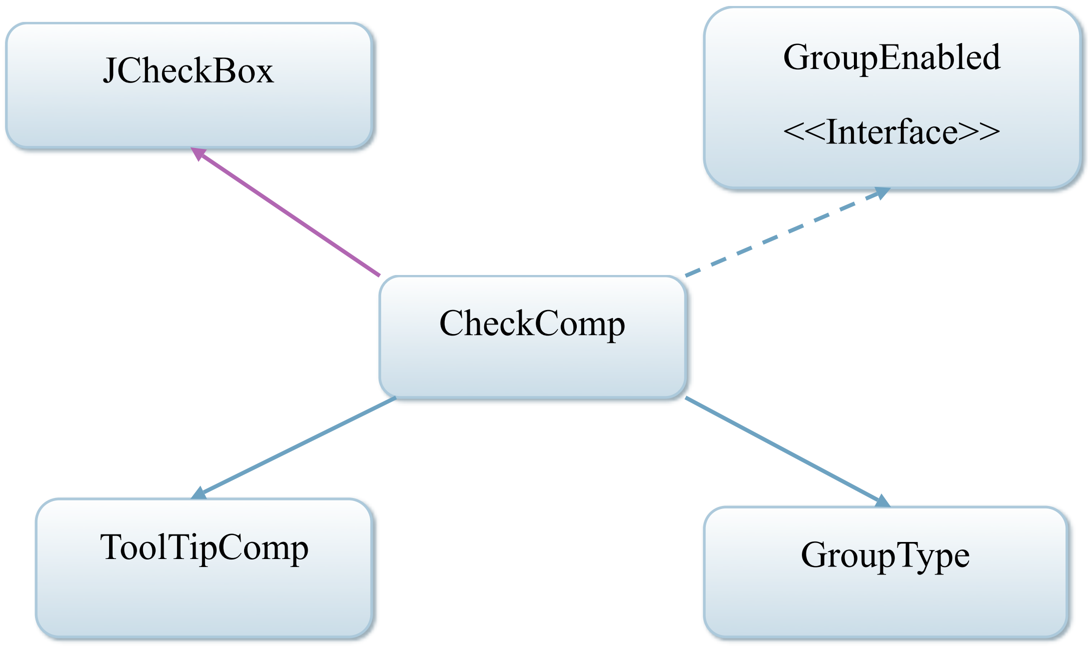
There are five packages in the desktop client project.

16.1 Comp package

The first package is the comp package. It consists of a collection of widgets, that are, in most cases, subclasses of standard swing JComponents. In a few cases, the widgets are direct subclasses of some of the widgets that are included with Sun's SwingLabs Desktop technology (http://swinglabs.org). By customizing Sun's widgets, it is possible to obtain a standard look-and-feel within the scheduler package with a minimal amount of code. For example, borders do not have to be set on each panel after it is created because the constructor of the customized widget will do it automatically. Particular features that work nicely with the custom widgets are:

- The custom widgets are configured with a specialized tool tip class that forces the tool tip to use line breaks and display on the screen with a reasonable size rather than as one very long rectangle.
- The custom widgets have a GroupType, which specifies the minimum logon level that a user of the desktop client must achieve in order for the full functionality of the widget to be available. Widgets supporting being enabled/disabled based on logon level implement the GroupEnabled interface.

Below, is a class diagram indicating the structure of one of the custom widgets in the scheduler desktop application package.



16.2 Validation package

Validation functionality within the dynamic scheduler is provided in the validation package. All validation is in the dynamic scheduler is performed on the client side. The structure for validation is provided by a base validation class called AbstractValidator. It uses a template pattern which provides a hook that subclasses can implement to provide specific validation.

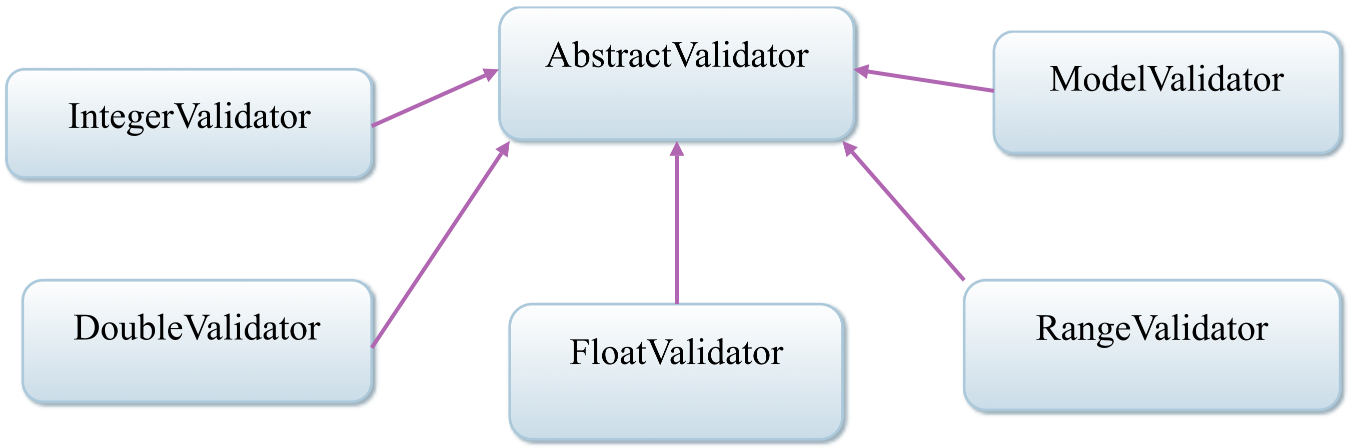
Most validation performed in the scheduler involves looking at a single field and analyzing whether the contents of that field are of the required type (integer, double, etc) and within a required range. Currently supported validators of this type are DoubleValidator FloatValidator, IntegerValidator, and RangeValidator. The validation framework is extensible so additional validators can be added as needed by extending AbstractValidator.

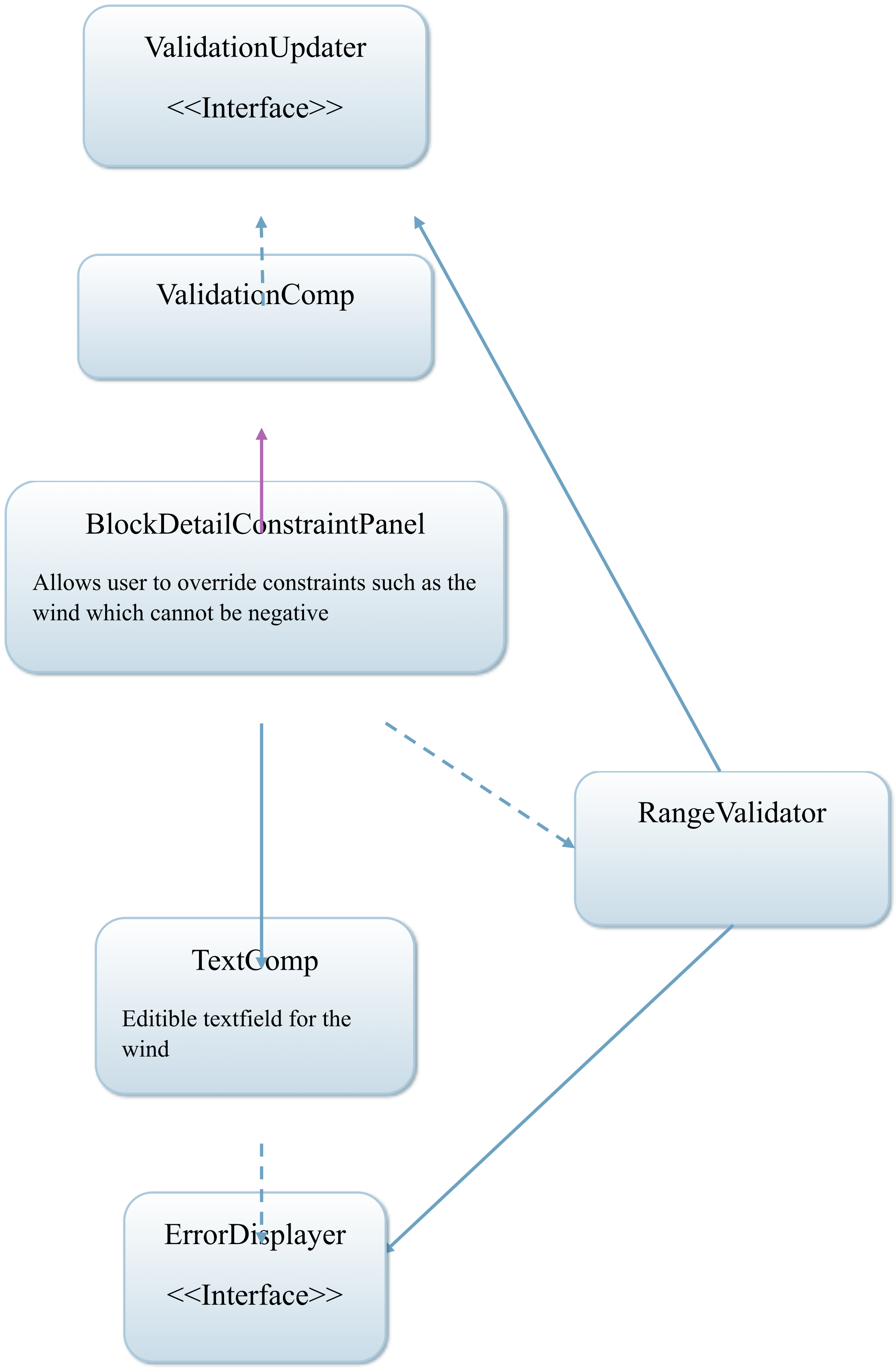
Occasionally the dynamic scheduler has the need to validate two fields within an object against one another. For example, the starting time for a schedule should always be less than the ending time of a schedule. In this case, the validation is performed by ModelValidator (also an extension of the AbstractValidator) which performs the validation by invoking the validation framework in the SSS model.

Display panels within the desktop application that have input elements that should be validated before their data is written to the model extend a widget in the comp package called ValidationComp. This widget is a customized panel for display error messages and for indicating that there is a validation problem by coloring its border red. They can also override the methods in the ValidationUpdater interface if they wish to customize the behavior provided by ValidationComp.

Certain widgets within the comp package, such as custom textfields like TextComp, implement the ErrorDisplayer interface. When their setDisplayError method is called with an argument of true, they typically use a red font and border.

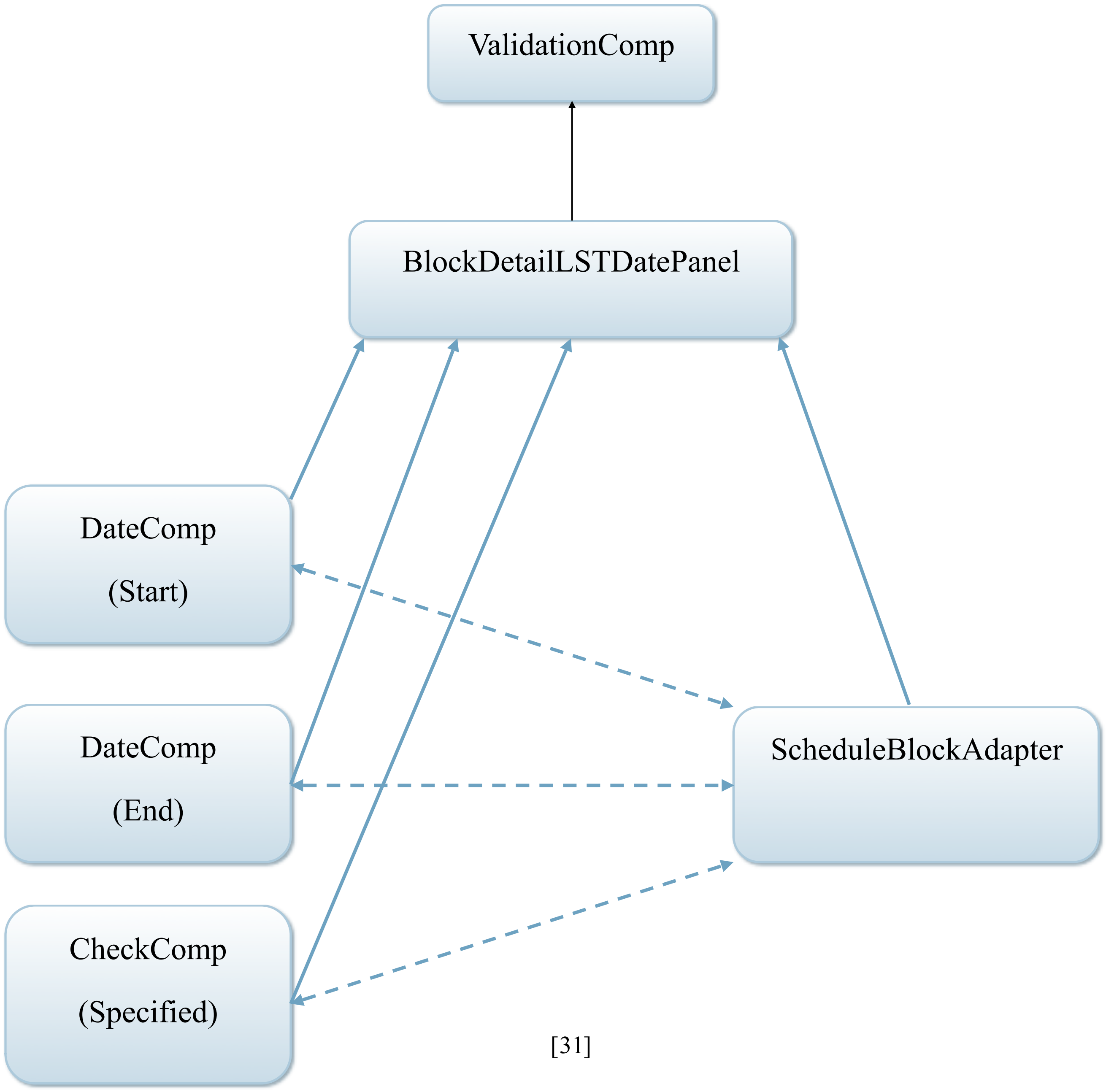
Class diagrams indicating the structure of validation within the scheduler desktop client are provided below.




ValidationUpdater
<<Interface>>
ValidationComp
BlockDetailConstraintPanel
Allows user to override constraints such as the wind which cannot be negative
RangeValidator
TextComp
Editible textfield for the wind
ErrorDisplayer
<<Interface>>

16.3 Panel Package

The panel package contains the major panels that make of the desktop client view. Each panel typically consists of custom widgets that make up the display. In addition, the panels typically extend ValidationComp in order to provide basic input validation and in order to display error/success messages resulting from validation and the operations that they support. As an example, a class diagram of one of the simpler panels, BlockDetailLSTDatePanel, is given, below. BlockDetailLSTDatePanel is responsible for displaying the preferred date range for a specific scheduling block if the range has been specified; otherwise it indicates that no preference has been indicated for the preferred date range.



16.4 Listener package

Contains a limited number of interfaces that are used by the various panels for communication purposes. Often one panel must change its display in response to a user event that occurs in another panel.

16.5 Layout package

Contains a customized layout manager to aid with laying out widgets in a common two column format.