# J

|  |
| --- |
| National Bank of Belgium |
| JDemetra+ |
| Developers' guide |
|  |
| **Jean Palate** |
| **9/7/2015** |

|  |
| --- |
|  |

This document presents some aspects of JDemetra+, following a developer's point of view. The main objective is to give a first introduction to the libraries, so that a developer or an advanced user could use them in a programming way for solving simple problems in the time series domain, and more especially in the seasonal adjustment domain.

The libraries of JDemetra+ are very large. The basic module that provides all the algorithmic part of the tool contains more than 1000 classes. Providing an exhaustive overview of them would be very daunting and - to a large extent - unproductive.

We have tried to present, using simple and if possible useful examples, some important parts of the library. The presentation mixes short explanations with class diagrams and code. We encourage the reader to discover the modules by experimenting the code and by inspecting - if need be - the source which is freely available [Github](https://github.com/jdemetra/jdemetra-core).

It is clear that a presentation based on a few examples has its limitations. A more systematic presentation of the main parts of the library, with numerical, statistical and design explanations will be progressively developed.

We only consider in this document examples of the use of the main algorithmic library (jtstoolkit.jar). The integration of new modules in the graphical interface (based on NetBeans) will be discussed in a separate document. The examples are provided in the project “JDemetra-Hello, available on [Github](https://github.com/nbbrd/jdemetra-hello). It can be compiled and tested by means of any Java development tool.

## Overview of JTsToolkit

The starting point of the library was seasonal adjustment. Tramo-Seats, more especially, has been a large source of inspiration. The concepts that are necessary for implementing such a method cover many mathematical and statistical fields. They have been developed over the actual needs of the original software. Some peripheral topics - necessary for testing, for developing new methods or new analysis tools - have also been added.

Finally, other methods that are used in the production of statistics have also been implemented. We think more especially to benchmarking and to temporal disaggregation. Those methods have been implemented using some advanced techniques, like state space models, which can also be used for completely different purposes.

So, the current state of the library is the results of an evolution started more than 10 years ago.

We provide below an overview of the main parts of the algorithmic library. A table will make the link between the diagram and the corresponding namespaces.

Matrix computation

Data handling and processing

Polynomials

Linear filters

Function optimization

Time series, calendars, regression model, time analysis...

Statistical distributions and tests

Utilities, algorithms...

Basic econometrics

Arima, Ucarima

Benchmarking, temporal disaggregation

Seats

X11

State space framework

Arima modelling

RegArima

Tramo

Seasonal adjustment

Structural models...

|  |  |
| --- | --- |
| Library blocks | Namespaces |
| Data processing | ec.tstoolkit.data |
| Matrix | ec.tstoolkit.maths.matrices... |
| Polynomials | ec.tstoolkit.polynomials |
| Linear filters | ec.tstoolkit.linearfilters |
| Optimization | ec.tstoolkit.maths.realfunctions... |
| Time series | ec.tstoolkit.timeseries...  ec.tstoolkit.timeseries.simplets... |
| Basic statistics | ec.tstoolkit.dstats  ec.tstoolkit.stats |
| Utilities | ec.tstoolkit.algorithms  ec.tstoolkit.information  ec.tstoolkit.utilities |
| Basic econometrics | ec.tstoolkit.eco... |
| Arima models | ec.tstoolkit.arima...  ec.tstoolkit.sarima...  ec.tstoolkit.ucarima... |
| State space framework | ec.tstoolkit.ssf... |
| RegArima modelling | ec.tstoolkit.modelling.arima  ec.tstoolkit.modelling.tramo  ec.tstoolkit.modelling.x13 |
| Other modelling | ec.tstoolkit.structural  ec.tstoolkit.modelling.arima.special |
| Seasonal adjustment | ec.satoolkit...  ec.satoolkit.x11  ec.satoolkit.seats |
| Benchmarking | ec.benchmarking... |

## Basic time series

### Short overview

The time series model used for computation is defined in the packages "**ec.tstoolkit.timeseries.x**". It is designed to allow easy manipulation, good performances, and to be adapted to the most frequent statistical operations on time series.

The most important classes, which describe the central concepts, belong to the namespace "ec.tstoolkit.timeseries.simplets".

|  |  |
| --- | --- |
| **Namespace** | **Short description** |
| ec.tstoolkit.timeseries | Basic concepts in the time domain |
| ec.tstoolkit.timeseries.simplets | Simple time series model   * Main concepts * Data retrieval * Simple operations |
| ec.tstoolkit.timeseries.calendars | (Gregorian) calendars (holidays...) |
| ec.tstoolkit.timeseries.regression | Regression model in the time domain |
| ec.tstoolkit.timeseries.analysis | Simple analysis tools |

## Time series Model

A time series (**TsData** object) is composed of its time domain (**TsDomain** object) and of an array of values (**Values** object), both of the same length. The domain is constant: it cannot be modified after the creation of the time series. On the other hand, if the number of values is also fixed, each of them can be modified at any time. The time domain is an array of adjacent periods (**TsPeriod** objects).

The time series treated by the model must have the following characteristics:

* Frequency: from the **TsFrequency** enumeration (number of periods by year corresponding to a divisor of 12).
* Domain: continuous; no limitation in length.
* Data: real values; the series can contain missing values (identified by Double.NaN).

### Creation

Simple time series can be created following two distinct ways. The first one prioritizes the view of a time series as an association between a continuous time domain and an array of values with the same length, the second one as a collection of couples "date/value".

The first way supposes that the domain is well known; in that case the creation process can be greatly optimized. The second way is better suited for the creation of a time series whose domain is not well known (undefined frequency or length...); it also allows the building of a time series with observations at irregular intervals.

#### Direct creation

A time series can be directly created from its domain or from its starting period and an array of data (doubles). Once a series has been created from its domain, it contains only missing values that can of course be initialized later. Direct creation should be the preferred method when the number of observations is known.

**Be aware that the periods of the year are always 0-based indexed.**

(Jan-2006 is {TsFrequency.Monthly, 2006, 0}, QIII-2006 is TsFrequency.Quarterly, 2006, 2}...).

See the example ***HelloDemetra1***

#### Creation through a **TsDataCollector** (in "ec.tstoolkit.timeseries.simplets")

A **TsDataCollector** is an object designed to collect any couple of "date/value". At any moment, it can create a time series (more than 1 observation), given a frequency (that can be "Undefined"), and an aggregation type (None, Sum, Average, Last...), that defines the way to aggregate the values, if necessary. The resulting time series will contain missing values if there are some "holes" in the array of dates. If the frequency is unknown, the object searches for the smaller frequency, if any, such that every period of the domain contains at most one observation.

Through the **TsDataCollector**, it is easy to create time series of any admissible frequency from any set, regular or not, of time observations.

See the example ***HelloDemetra2***

### Data handling

#### Data blocks

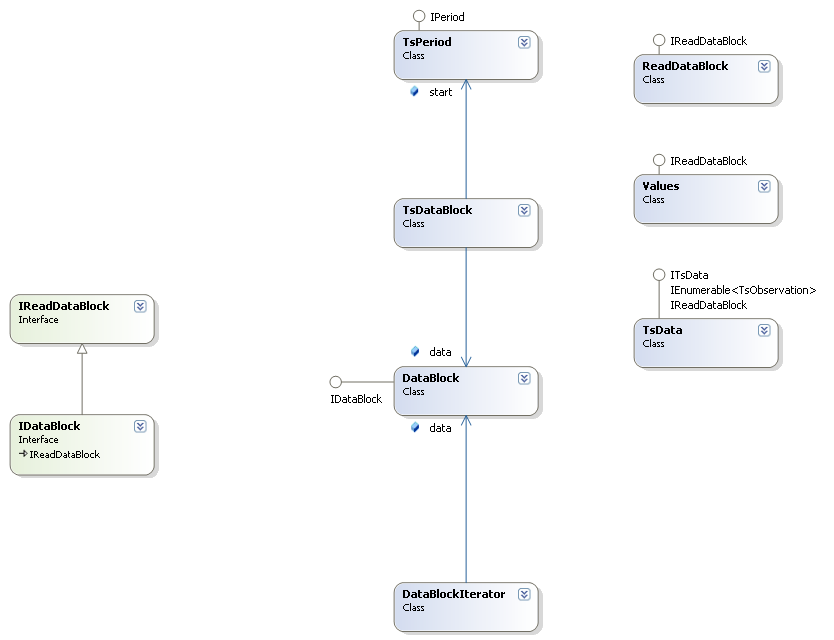
Before explaining the data handling of time series, we have to shortly explain the concept of "**DataBlock**", which is hugely used in the entire library. The main interfaces and the basic classes related to that concept are defined in the package "**ec.tstoolkit.data**".

The basic interfaces **IReadDataBlock**, **IDataBlock** formalize read/write access to an ordered set of data (doubles). The most used implementation of those interfaces is the **DataBlock** class (also widely used in matrix computation).

A **DataBlock** is simply an array of data equally spaced in some array of doubles. It is fully identified by its underlying memory block, the starting position (included) and the ending position (excluded) of the data along with the increment (possibly negative) between two elements.

Such a simple structure allows, for instance, the handling of rows, columns and diagonals of matrices or the extraction of time series elements corresponding to a specific year or to a given period of the year.

It is important to notice that a DataBlock doesn’t contain a copy of the data, but that it is a “window” on some data of the underlying memory block: changing data through a DataBlock will actually modify the original data of that memory block.

**

The library provides many facilities to handle **DataBlock** objects (and related). The class “**DescriptiveStatistics**”, for instance, offers the usual descriptive statistics on a set of data.

See the examples ***HelloDemetra3***, ***HelloDemetra4*** and ***HelloDemetra5*.**

#### *Data blocks in times series*

Beside the direct access to the observations of a TsData, through its get(), set() methods (using 0-based indexes), users can retrieve information by means of a TsDataBlock or by means of an iterator (YearIterator, PeriodIterator).

Basically, a TsDataBlock is just a DataBlock enriched by information corresponding to the time periods of its elements and YearIterator, PeriodIterator are iterators on specific TsDataBlocks of a series, i.e. years or same periods of the year.

Working on time series through TsDataBlock is always more efficient than using a direct access to the data.

### Operations

Usual operations on time series are enclosed in the definition of the TsData class. It concerns, for instance, binary operators between time series, like +, -, \* or unary transformation like exp, log, power, moving average/median...It should be noted that those operations always generate new time series objects, leaving the existing ones unchanged.

We provide in the example ***HelloDemetra7*** an implementation of chain-linking with annual overlap. The code is a good example of the use of the methods on time series provided by the **TsData** class as well as the use of iterators on time series.

# Seasonal Adjustment

We consider in this paragraph the programming use of the two leading algorithms of the Seasonal Adjustment domain, i.e. Tramo-Seats and of X13**[[1]](#footnote-1)**.

Most of the high-level algorithms provided by the "jtstoolkit.jar" library may be used in different ways. For people that don't need to get advanced information on a processing, the use of the "generic" interface defined in the package "ec.tstoolkit.algorithms" should be the preferred solution. We discuss it shortly in the first paragraph. A more complete discussion on generic algorithm interfaces will be provided later.

People who want to access all information or to interact in a fine way with the routines will have to learn the specific design of the considered algorithms. We present in the next points the main parts of Tramo-Seats and of X12. The concepts handled by Tramo-Seats and X13 and the design of the Java implementation will be discussed in more detailed in specific paragraphs.

## Generic interface

The generic protocol for algorithms can be summarized as follows: each processing is defined by a specification (parameters); it can be applied on a set of data (input) to produce a results set.

For example, a "Tramo-Seats" processing is defined by a set of parameters that are applied on a time series to produce the final decomposition. In practice, calling an algorithm will involve several objects:

* a "specification" object that defines the parameters of the processing
* a "processing factory", which will generate the actual algorithm (processing) for a given specification
* an input (for instance a time series) that will be passed to the processing for generating the results set

In its generic form, the results will be inspected by means of a "dictionary", which describes the available information.

To a large extent, the splitting between specification and input is arbitrary. For example, in the case of Tramo-Seats, the current implementation considers that the input is the time series and that the other parameters are defined in the specification; however, we could also consider as input the pre-specified regression model; the specification would then contain the automatic estimation part and the parameters to control the processing.

It should also be noted that the same processing may be re-used with different inputs and that, for the most used algorithms, the library provides some short cuts.

An example of the use of the generic interface for processing Tramo-Seats and X12 can be found in HelloDemetra10 and HelloDemetra11

## Tramo-Seats

A Tramo-Seats processing consists in applying a TramoSeats specification to a given time series (**TsData**). The result -as often in jtstoolkit for complex processing - is a "**CompositeResults**", which is just the juxtaposition of several "sub-results".

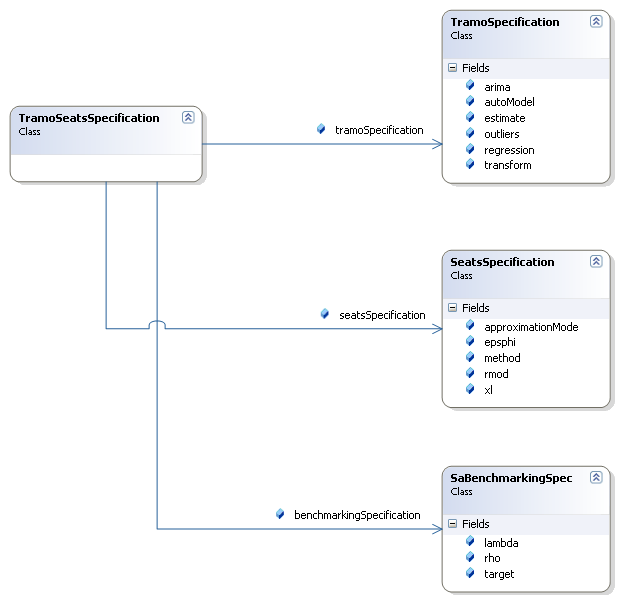
We describe below the structure of a **TramoSeatsSpecification** and of the different parts of the results.

### Tramo-Seats specifications

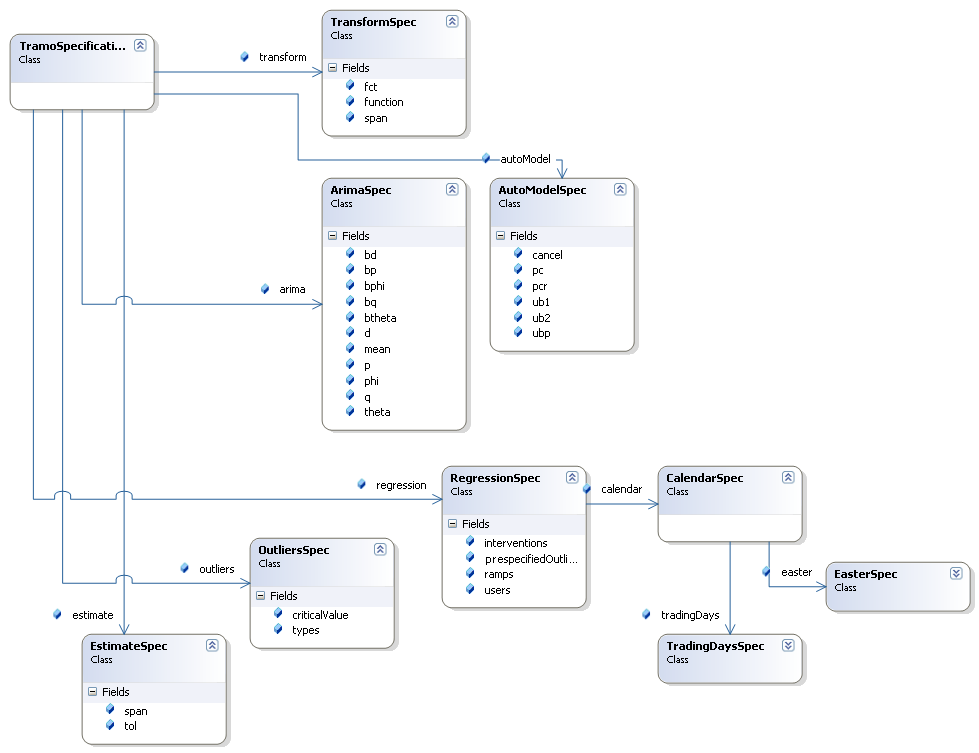
To a large extent, the **TramoSeatsSpecification** object corresponds to an input file of the original program (without the series itself).

It contains specifications for Tramo (pre-processing), for Seats (decomposition) and for univariate benchmarking (based on the Cholette's method). Strictly speaking, benchmarking is not a part of seasonal adjustment. By default, it is not enabled.

The different components of a **TramoSeatsSpecification** are displayed on the next diagrams. The diagrams don't provide all the available parameters. More information is available in the description of the API.



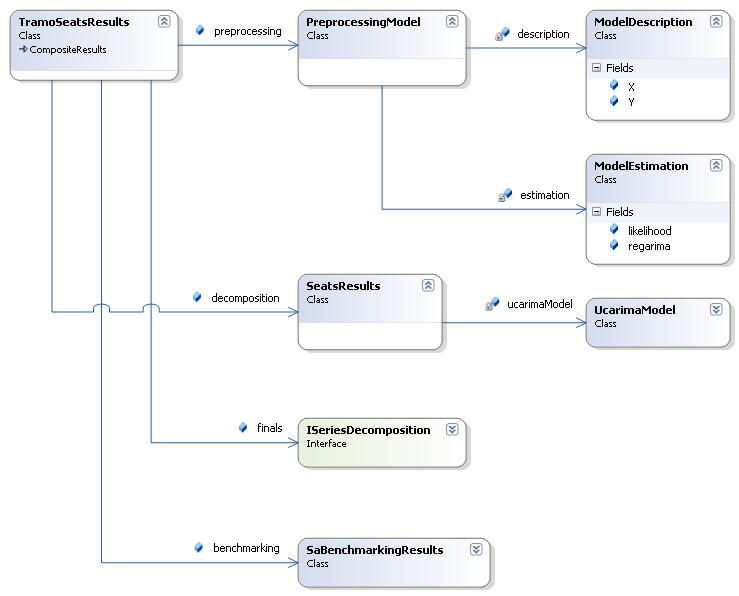
Details on **TramoSpecification**



### Tramo-Seats results

The results for a Tramo-Seats processing are stored in a **CompositeResults** object that contains the following parts:

|  |  |  |
| --- | --- | --- |
| **Key** | **Class** | **Description** |
| preprocessing | ec.tstoolkit.modelling.arima.PreprocessingModel | RegArima model. See below for a detailed discussion |
| decomposition | ec.satoolkit.seats.SeatsResults | Canonical decomposition |
| finals | ec.satoolkit.ISeriesDecomposition | Final components |
| benchmarking | ec.satoolkit.benchmarking.SaBenchmarkingResults | Benchmarking of the sa series (optional) |



## X13

The X13 model is very similar to the Tramo-Seats one. An X13 processing consists in applying an X13 specification to a given time series (**TsData**). The result is also a "**CompositeResults**", with very similar sub-parts.

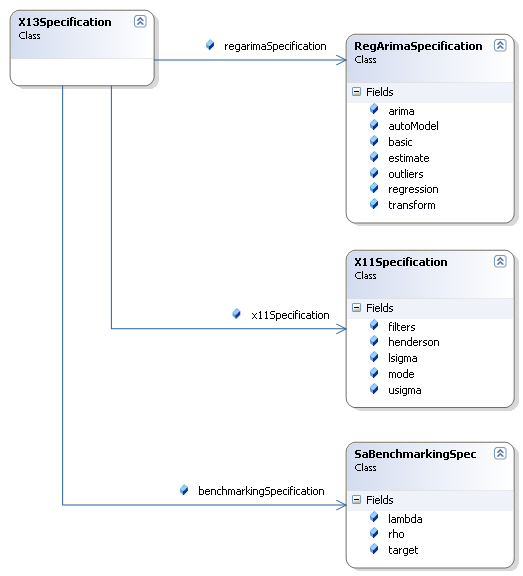
We describe below the structure of an **X13Specification** and of the different parts of the results.

### X13 specifications

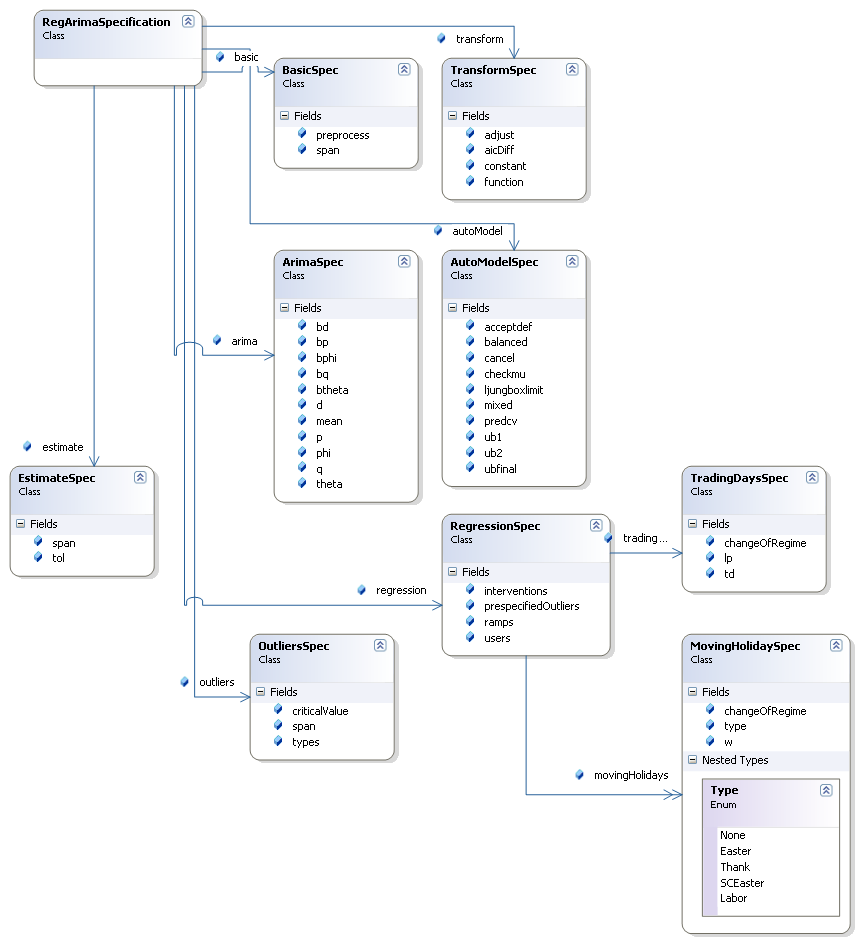
To a large extent, the **X13Specification** object has a structure similar to the different "specs" of the original program.

It contains specifications for RegArima (pre-processing), for X11 (decomposition) and for univariate benchmarking (based on the Cholette's method, "force" spec).

The different components of an **X13Specification** are displayed on the next diagrams. The diagrams don't provide all the available parameters. More information is available in the description of the API.



Details on **RegArimaSpecification**

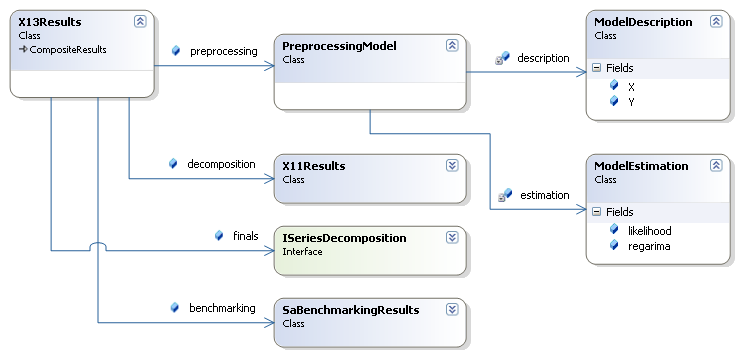


### X13 results

The results of an X13 processing are stored in a **CompositeResults** object that contains the following parts:

|  |  |  |
| --- | --- | --- |
| **Key** | **Class** | **Description** |
| preprocessing | ec.tstoolkit.modelling.arima.PreprocessingModel | RegArima model. See below for a detailed discussion |
| decomposition | ec.satoolkit.x11.X11Results | X11 decomposition |
| finals | ec.satoolkit.ISeriesDecomposition | Final components |
| benchmarking | ec.satoolkit.benchmarking.SaBenchmarkingResults | Benchmarking of the sa series (optional) |

Apart from the decomposition itself (X11 instead of Seats), all the different sub-components of the X13 results are strictly identical to the Tramo-Seats ones.



We provide in HelloDemetra12 and HelloDemetra13 some examples that show the use of the detailed output.

## Tramo-Seats and X13 pre-processing: common framework

Before going into the details of the regression models estimated by Tramo and by X13 and into their automatic identification methods, it seems useful to clarify a few points. When we are talking about the Arima modelling features of Tramo and of X13, we consider several things:

* The definition of the regression model itself. It includes:
  + Regression variables
  + Arima component
* The estimation of the model.
  + When a model contains parameters, we usually try to estimate them following a criterion that we want to maximize.
  + In the case of RegArima models, the criterion is generally the maximum likelihood (or a variant of it). There exist many methods to estimate the likelihood of a model; Tramo and X13 use the same definition for the likelihood, but they estimate it following different approaches.
  + The maximization of the likelihood depends on an optimization procedure. Tramo and X13 use different strategies.
* The identification of the model
  + Despite the differences mentioned above, Tramo and X13 get (in most cases) the same results (or very similar ones) when a model is entirely specified.
  + Tramo and X13 should be considered as expert systems that identify the "best" model following rather similar methods (firstly developed in Tramo); however, they differ in many details, so that their final models often differ.

Concerning JDemetra+, the library "jtstoolkit.jar" provides a definition of regression model that encompasses the Tramo and the X13 ones (as far as the usual SARIMA models are considered). The different types of variables are defined in the package "ec.tstoolkit.timeseries.regression" and the models themselves are defined in the package "ec.tstoolkit.modelling.arima". See below for further details on the regression variables. It should be noted that the regression variables are not linked to Tramo or to X13 (they can be used for other purposes, like temporal disaggregation).

"jtstoolkit.jar" contains numerous estimation procedures of ARIMA (regression) models. Most of them use the same likelihood definition as Tramo/X13, i.e. the likelihood of the differenced model or - equivalently - the likelihood conditional to the first observations**[[2]](#footnote-2)**. See the classes ArmaKF (Tramo-like), AnsleyFilter, KalmanFilter, LjungBoxFilter, ModifiedLjungBoxFilter (X13-like) in the package "ec.tstoolkit.arima.estimation". The different methods yield the same likelihood evaluation; however, they usually give different "residuals", so that diagnostics may differ**[[3]](#footnote-3)**. The different estimation procedures are low-level routines (independent of Tramo, of X13 and of the time series model itself) that can be used to evaluate the likelihood of other parametric forms of ARIMA models (for example generalized airline models) or models corresponding to unusual frequencies (for example daily models for hourly observations).

The low-level optimization procedures developed in JDemetra+ belong to the same family (Levenberg-Marquardt) as Tramo and X13[[4]](#footnote-4). However the Java implementation differs in some details. For Arima models, JDemetra+ proposes the concentrated likelihood optimization procedure (based on a QR decomposition) similar to the Tramo ones (see the classes "**ec.tstoolkit.arima.estimation.GlsArimaMonitor**" and "**ec.tstoolkit.sarima.estimation.GlsSarimaMonitor**"). It also contains the so-called "iterative GLS" estimation procedure used in X13 (see the classes "**ec.tstoolkit.arima.estimation.IGlsArimaMonitor**" and "**ec.tstoolkit.sarima.estimation.IGlsSarimaMonitor**").

Jtstoolkit.jar tries to reproduce the experts systems defined in Tramo and in X13, in a much more modular way. Due to the complexity of the original procedures and to the completely different design of jtstoolkit.jar, some unintentional incoherencies are still present.

In this paragraph, we will first describe the regression models used in jtstoolkit.jar. We will then present the automatic model identification strategies of the core engines and the way they have been implemented in jtstoolkit.jar

### Regression models, definition and estimation

The regression models identified by Tramo and by X13 have the same form. We describe first the variables that can be used in the model. We consider in a second point the PreprocessingModel objects, which contain both the description and the estimation of a model. That class is the main output of the pre-processing step of TramoSeats and of X13.

#### Regression variables

All the regression variables that are used in Tramo or in Seats are defined in the package "**ec.tstoolkit.timeseries.regression**". The module provides a large set of variables. Their object-oriented design will allow easy and powerful extensions (see the examples below for a first idea).

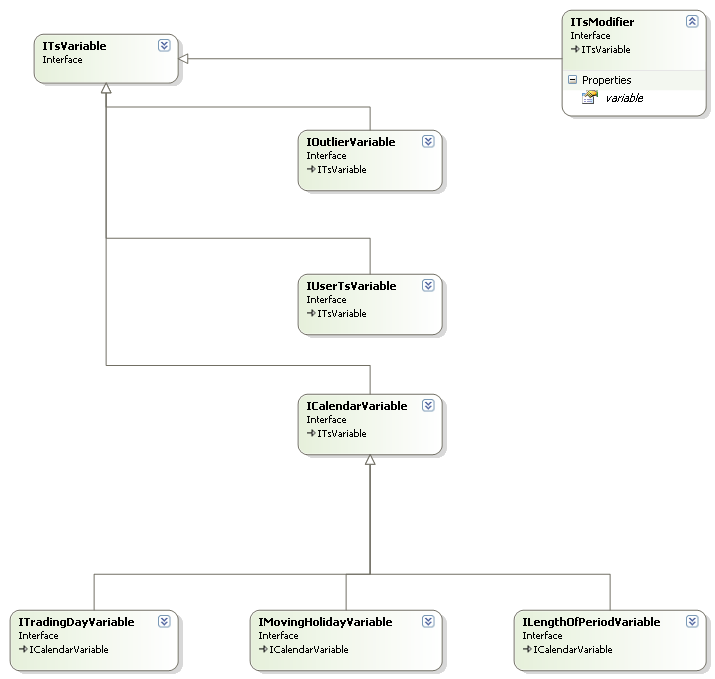
Most of the regression variables defined in jtstoolkit are self-generating, which means that they are able to provide figures for any time domain. This is the case, for instance, for the outliers and for the calendar variables. Obvious exceptions are the variables that are defined by a specific time series.

The regression variables are classified by means of a hierarchy of interfaces that derived from the basic **ITsVariable** interface. The goal of that hierarchy is mainly to allow a flexible way to select some types of variables (including variables that are defined outside the core library) in a complete regression model.

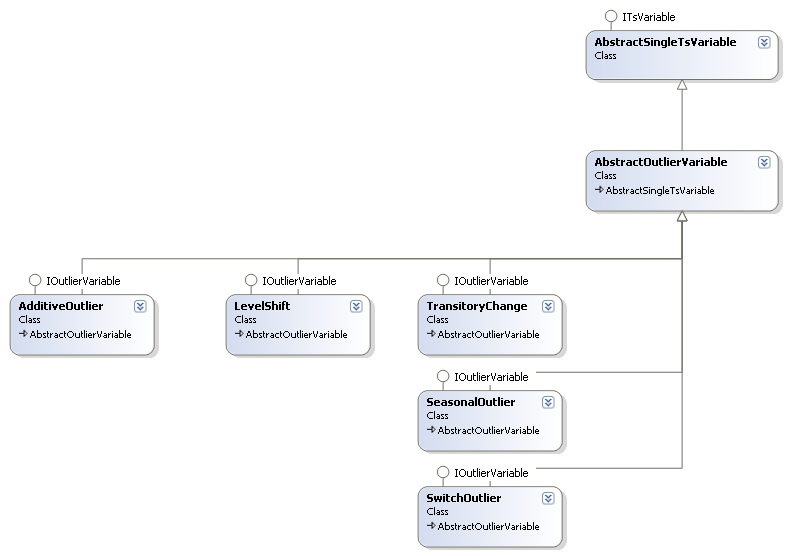
Beside the "atomic" variables, the library also defines "modifiers" (implementing the **ITsModifier** interface), which can modify the behavior of other variables. Lagged variables or change of regimes (as defined in X13) are examples of modifiers.

The following diagrams provide an overview of the hierarchy of the main interface, of the outliers variables, of the calendar variables and of the other kind of regression variables defined in the library. We outline in the code provided below the main principles for the use of the regression variables and we show how the current model can be extended.

*Main regression variables interfaces*

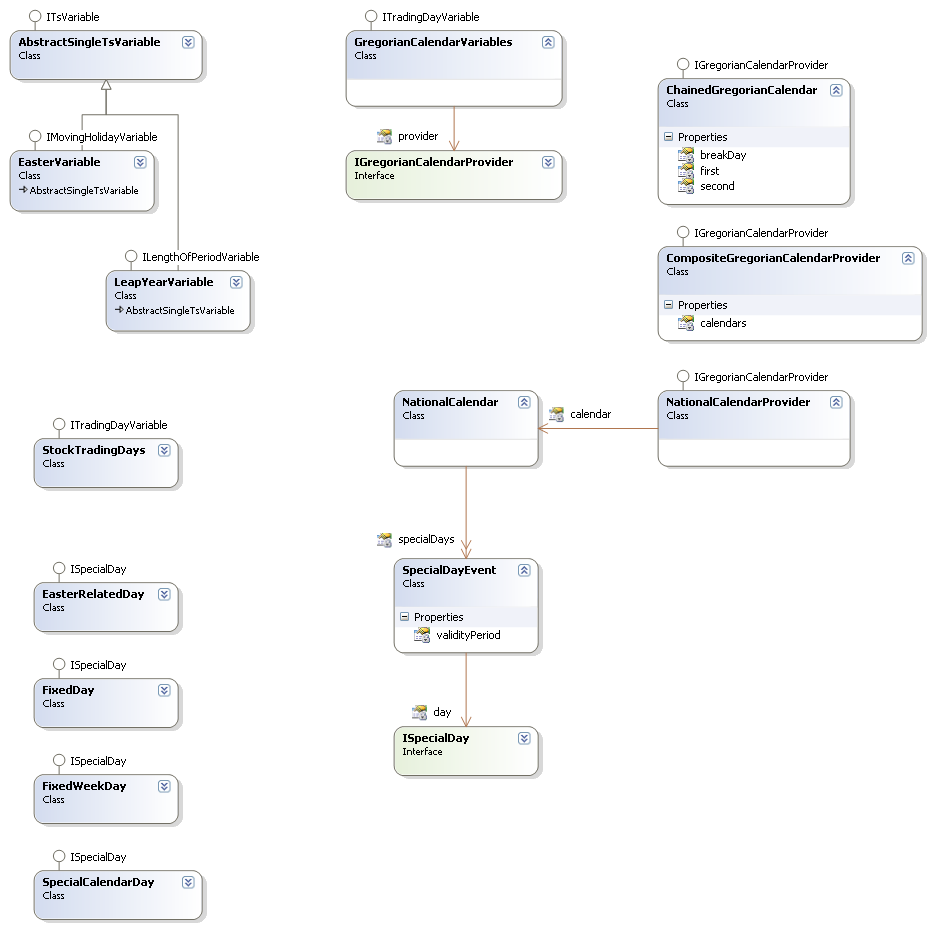


*Implementations of* ***IOutlierVariable***

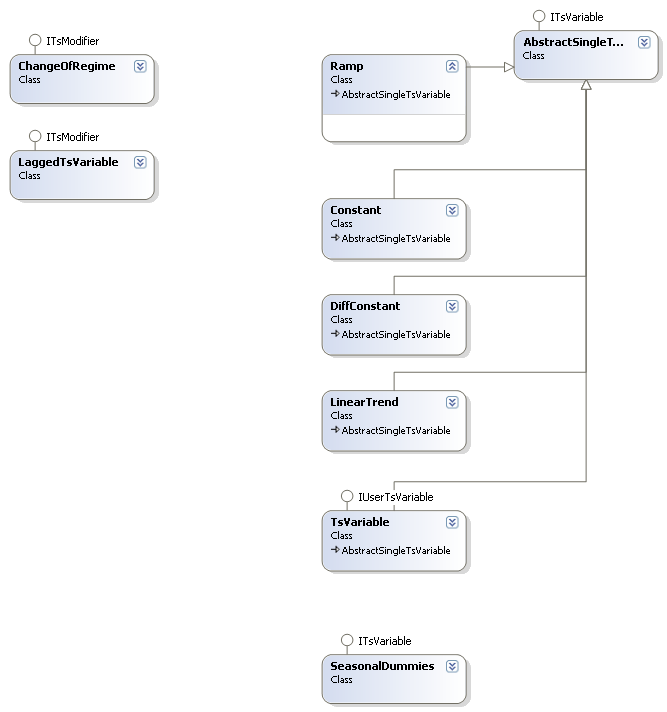


The next diagram shows the current calendar variables, which are based on different implementation classes located in the package "**ec.tstoolkit.timeseries.calendars**". The way calendars are implemented in the library will be discussed in a specific paragraph.

***Calendars and Calendar variables***



*Other regression variable and modifiers*



The code below focuses on the handling of the regression variables.

##### Example 1: basic use of regression variables

The principal aim of regression variables is to provide data for a given time domain. They also are able to provide short descriptions. Variables may be single (dim = 1) or grouped (trading days...). They are put together in a "**TsVariableList**".

In the following example, a set of usual trading days (without and with a change of regime) is generated. Seasonal dummies are also added to the list.

It is then possible to retrieve information for any selection in that list, using the "**select**", "**selectCompatible**" or "**all**”. See ***HelloDemetra14*** for an example

##### Example 2: defining a new regression variable and a new modifier

The library has been designed to permit the creation of new regression variables and of new modifiers.

Adding new variables/modifiers in the graphical interface of JDemetra+ requires of course more steps:

* To define the algorithm for generating the data.
* To define the parameters that will identify the variable and to integrate them in the current specifications
* To create an editor for the parameters

However, for research or for batch processing through the programming interface, only the first step is needed. We present below two small examples. The first one is a raw implementation of a variable representing the Julian Easters (rather inefficient code, no long term correction) and the second one is a modifier which is able to extend other variables by forecasts/backcasts generated by Tramo.

See the files "HelloDemetra15" and "HelloDemetra16".

#### Preprocessing model

All information that makes up a "RegArima" model is put together in a **ModelDescription** object (in the package "**ec.tstoolkit.modelling.arima**"). Such an object contains:

* The explained series
* Its preliminary transformations (log, correction for the length of periods...)
* The variables of the regression
* The Arima model that describes the residuals (structure and - if available - parameters) (Box-Jenkins seasonal Arima model, described in "**ec.tstoolkit.sarima.SarimaModel**")

The main task of that object is to generate the low-level data of the RegArima model ("**ec.tstoolkit.arima.estimation.RegArimaModel<M>**") for estimation.

This point is illustrated by the code in ***HelloDemetra17***

The actual estimation of a **RegArimaModel<SarimaModel>** may be processed through different algorithms. The code below, which estimates the model described in the previous paragraph, use implementations that correspond (more or less) to Tramo (Kalman filter + QR decomposition) and to X13 (Iterative Gls). It should be noted that, using low-level objects, advanced developers are able to change much more details (optimization routines, initial estimates, filters...); they also are able to retrieve more information (number of iterations, hessian/gradient of the likelihood function at the solution...).

See ***HelloDemetra18***.

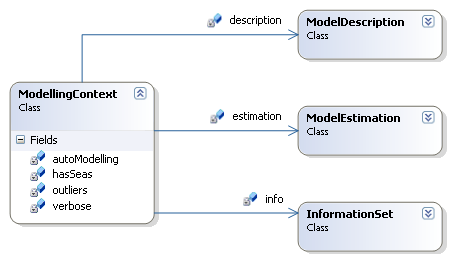
### Automatic model identification

The implementations of the automatic model identification (AMI) of Tramo and of X13 are based on an identical framework.

Basically, it can be described as an iterative processing which progressively specifies and improves a model that was initially defined by a specification object.

All information on the current state of the model - its description, its estimation and some additional information - is stored in a unique object, called a **ModellingContext**.

The **ModellingContext** is described in the next diagram. It is very similar to a **PreprocessingModel**, which is just the "stabilized" view of a **ModellingContext** (in the package "**ec.tstoolkit.modelling.Arima**".



The main function of the different modules that make up the full algorithm is to modify the modelling context. We consider in the current implementation the following modules (identified by the **IPreprocessingModule, IModelEstimator** and **IModelController** interfaces).

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Tramo classes**  (in **ec.tstoolkit.modelling**  **.arima.tramo**) | **X13 classes**  (in **ec.tstoolkit.modelling**  **.arima.x13**) | **Description,**  **Remarks** |
| Initialization | DefaultModelBuilder  TramoModelBuider | X13ModelBuider | Initializes the modelling context, using the (Tramo/Regarima) specifications |
| Log/level test | RangeMeanTest  LogLevelTest | LogLevelTest | Choose between log and level. Tramo and X13 are very similar. |
| Regression tests | RegressionVariablesTest  RegressionVariables-  Controller  RegressionTestTD  RegressionTestTD2 | CalendarEffectDetection  EasterDetection RegressionVariablesTest  RegressionVariablesTest2 | Tramo uses T-Stats[[5]](#footnote-5), X13 uses AIC tests or T-Stats[[6]](#footnote-6). The used modules will depend on the step of the AMI |
| Outliers detection | OutliersDetector | OutliersDetector | The detection of outliers is the most sensitive module for good performances. |
| Differencing orders | DifferencingModule | DifferencingModule | Search the differencing orders (regular/seasonal) |
| Arma identification | ArmaModule | ArmaModule | Search the best ar/ma orders |
| Estimation | ModelEstimator  FinalEstimator | RegArimaEstimator  FinalEstimator,  IGlsFinalEstimator | Estimation of the model. May slightly differ following the step. |
| Controllers | ModelBenchmarking  RegularUnderDifferencing-  Test  SeasonalUnderDifferencing-Test  SeasonalOverDifferencing-  Test  SeasonalityController | MeanController  ModelController | Check the validity of a model (comparison with an airline, over/under differencing, residual seasonality... |

The different modules are put in monitoring objects - **TramoProcessor** and **X13Preprocessor** - which have to control the AMI path following the state of the current **ModellingContext**.

We put below a very synthetic schema that describe the flow of a complete AMI in Tramo. X13 is very similar. Following the options, the iteration round and the state of the current model, some steps may be skipped (which explains some apparent redundancies).

Changed

Failed

Initialization

Log/level

Regression (TD, Easter...)

Differencing

Arma identification

Outliers detection

Check model

Check regression

Final controls

Final

estimation

Check regression

Changed

It is interesting to note that the different modules are independent of their core engines, so that they can be used for other purposes. The next paragraph is a simple example of that feature.

#### Modifying the core engines

The goal of the library is certainly not to modify the current core engines. However, it should simplify their maintenance as well as their more systematic testing.

The modular approach of the Java implementation avoids the pitfalls of the Fortran global variables that make developers' life hell. The code below shows the flexibility of the current approach by replacing the outliers detection module of X13 by the Tramo's one. So, it also shows the relationships between the specifications and the processors: specifications should be considered as scripts that are used to put together and to initialize the sub-modules that constitute the algorithm itself. Developers could define new specifications that could lead to new algorithms using the same approach or they could directly modify, as in the example provided in ***HelloDemetra19***, some parts of an existing algorithm.

## Applications based on the RegArima modelling

The API simplifies in a dramatic way the use of the modelling facilities offered by Tramo-Seats or X13. We consider below a few examples, in the research domain or in the operational domain.

#### Outliers detection

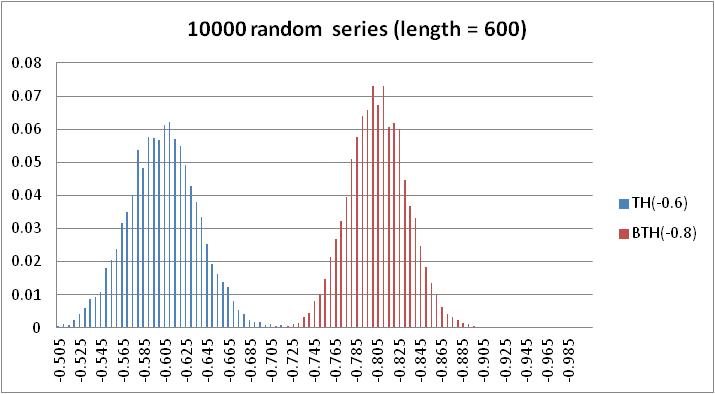
Outliers detection may be a very useful tool in the production process of statistics. The example ***HelloDemetra20*** shows how to call it from jtstoolkit.

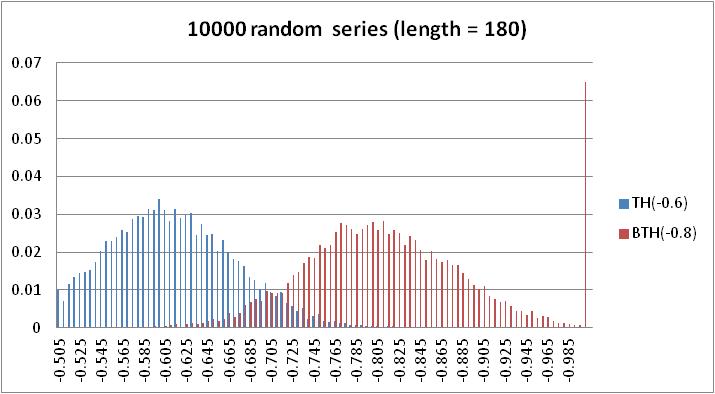
Users are able to define new types of outliers and to add them in the outliers detection modules of Tramo or of X13. We consider below a "switch outlier", identified by the sequence "0...0 1 -1 0...0" (that corresponds to a switch between two adjacent periods).

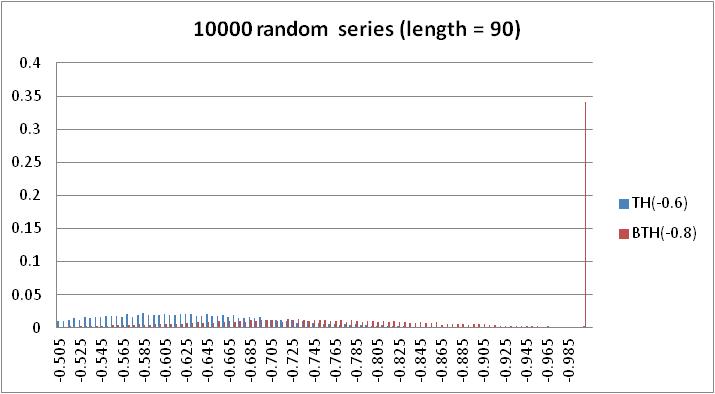
Defining a new kind of outliers will need a new implementation of the "IOutlierVariable" interface and of the corresponding "factory". The code is implemented in ***HelloDemetra21***.

#### Simulation

In the example provided in ***HelloDemetra22***, we consider the estimation of Airline models with random series. It puts a well-known problem forward: the estimation of moving average parameters may be biased towards -1, especially in the case of short series. The results we get should arouse further investigations on the quality of the random series (try other parameters), the robustness of the estimation procedure (try alternative methods)... before concluding.







#### "Terror"-like tool

"Terror" is a popular tool for detecting anomalies in the last observation(s). The tool simply compares the out of sample forecasts of the series (shortened by the corresponding number of observations) with the actual figures. The differences are expressed in function of the standard error of the forecasts.

The class "**ec.tstoolkit.modelling.arima.CheckLast**" is a straightforward implementation of Terror.

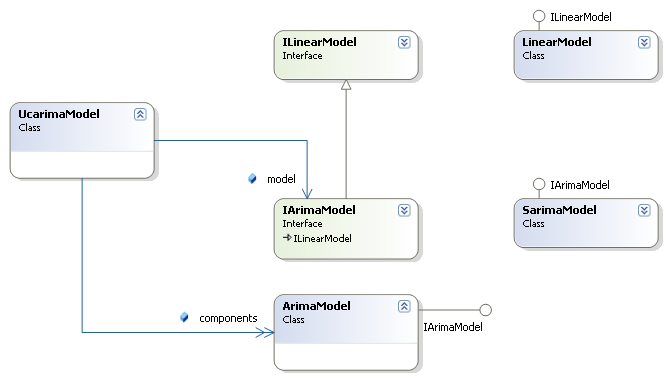
See ***HelloDemetra9*** for some examples.

## Model-based decomposition

The model-based decomposition of jtstoolkit is largely based on the concepts of "**IArimaModel**" ("**ec.tstoolkit. arima**" package) and of "**UcarimaModel**" ("**ec.tstoolkit.ucarima**" package).

The main implementations of the generic interface of Arima model are the general **ArimaModel** class ("**ec.tstoolkit. arima**" package) and the seasonal Box-Jenkins **SarimaModel** ("**ec.tstoolkit. sarima**" package).

An **UcarimaModel** (unobserved components Arima models) is composed of a list of **ArimaModel**s with independent innovations.



The decompositions algorithms will estimate such a model for a given series. The library provides the "Wiener-Kolmogorov" approach, based on the Burman's algorithm (Seats-like), a Kalman smoother solution and the direct matrix computation derived by McElroy.

The low-level implementations are specified in the next table.

|  |  |  |
| --- | --- | --- |
| **Method** | **Packages** | **Classes** |
| Burman | ec.tstoolkit.ucarima.estimation | BurmanEstimatesC |
| Kalman smoother | ec.ssf.ucarima  ec.ssf | SsfUcarima  SsfUcarimaWithMean  Smoother  DistrubanceSmoother |
| Matrix computation | ec.tstoolkit.ucarima.estimation | McElroyEstimates |

### Using Arima models

The library provides the usual properties on Arima models: auto-covariance functions, (pseudo-)spectrum... It also allows straightforward operations on them (under the assumption that they have independent innovations): addition, subtraction...

Some of them are illustrated in ***HelloDemetra23 and HelloDemetra24***

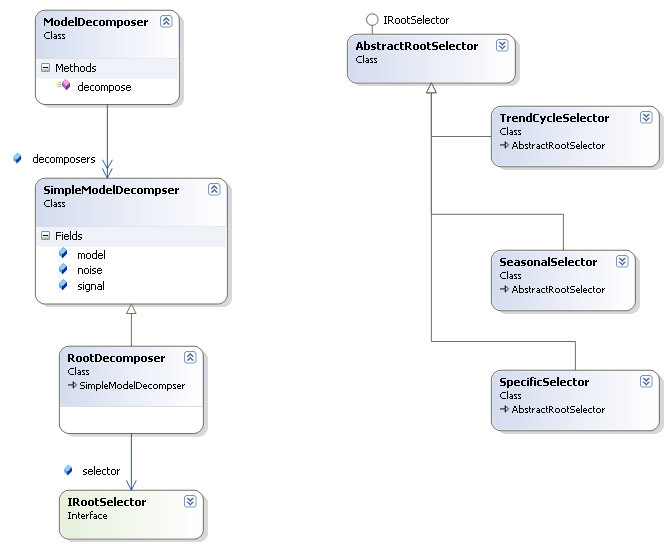
### Estimation of Ucarima models

Starting from a list of Arima models, we can build an **UcarimaModel**, which is the basis for the estimation of the components of a series.

Using the example ***HelloDemetra24***, we compute in ***HelloDemetra25*** the usual Hosrick-Presoctt decomposition of a series.

### Canonical decomposition

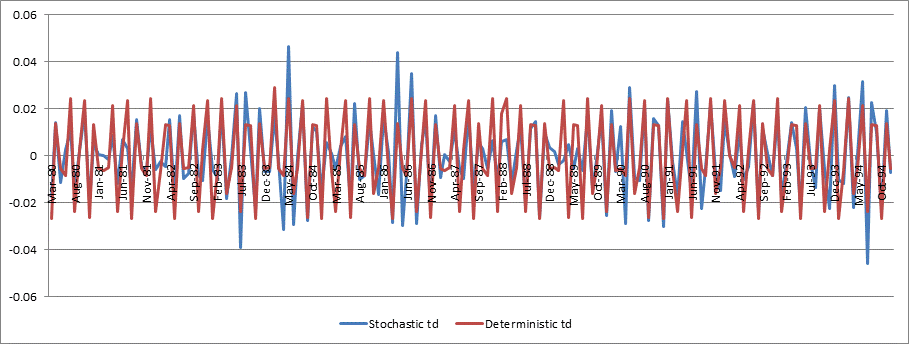
The decomposition of an Arima model in its components is based on the solution developed in Seats: the model is factorized following the roots of its auto-regressive polynomial.



In jtstoolkit, we use a very flexible design that could allow alternative decompositions. The **ModelDecomposer** is a list of **SimpleModelDecomposer**s, which split a model in a signal and in a noise. The default implementation is based - as Seats - on the splitting of the roots of the auto-regressive polynomial. Several strategies for such a splitting are proposed; they can be extended.

We provide in the test file ***HelloDemetra26*** a decomposition that contains the new stochastic trading days component of Seats (not yet available in the graphical interface). For the current implementation, the user will refer to the class "**ec.satoolkits.seats.DefaultModelDecomposer**".

See below for the output.



1. Be aware that the current implementation is not fully compatible with the original programs: the Java implementation doesn't provide all the options of the Fortran programs. Moreover, the results may slightly differ, because of different estimation routines. Those points will be discussed in more details further. [↑](#footnote-ref-1)
2. In the case of non-stationary series, the usual likelihood is not properly defined. Usually, the likelihood of the differenced series is then computed. Using the state-space framework avaialable in "ec.tstoolkit.ssf", it is also possible to compute the "diffuse likelihood", as defined for instance in "De Jong" or in "Durbin-Koopman". [↑](#footnote-ref-2)
3. ArmaKF is usually the fastest. When the model contains many regression variables, AnsleyFilter may become the most efficient one. The others are provided for test purposes or for compatibility with existing software. ArmaKF, AnsleyFilter and Kalman Filter lead to the same residuals (one-step ahead forecast errors). [↑](#footnote-ref-3)
4. JDemetra+ also contains an implementation of BFGS. All the low level routines are defined in the packages "**ec.tstoolkit.math.realfunctions...**". They constitute one of the most tricky parts of the entire library. [↑](#footnote-ref-4)
5. The new version of Tramo will use F-Tests for some variables. It is quite similar to the X13 solution [↑](#footnote-ref-5)
6. For TD, X13 computes in some steps the T-Stats, including on the derived regression variable. It is then very similar to a F-Test. [↑](#footnote-ref-6)