Design document

DaCoPAn2

Helsinki, 7th April 2005

Software Engineering Project

UNIVERSITY OF HELSINKI
Department of Computer Science

Course

581260-4 Software Engineering Project (6 cr)

Project Group

Mikko Airaksinen
Tomi Korkki
Pauli Miettinen
Timo Tuominen
Mikko Väänänen

Customer

Markku Kojo

Project Masters

Juha Taina (Supervisor)
Marianne Korpela (Instructor)

Homepage

`http://www.cs.helsinki.fi/group/dacopan2`

Change Log

Version	Date	Modifications
1.0	March 17th 2005	First version
2.0	March 29th 2005	Finished version

Contents

1 Introduction

The DaCoPAn Animator module enables the visualization of the packet trace information provided by the DaCoPAn Analyzer module through the Protocol Events File (PEF). This DaCoPAn2 Animator Design Document is an updated document based on the existing DaCoPAn implementation and the original DaCoPAn Animator Design Document [1]. It is a stand-alone document, which describes the entire Animator software along with the changes that will be made to it by the DaCoPAn2 project.

Section 2 presents the general architecture of the Animator module. An overview diagram will introduce us to the different elements of the Animator, explaining how they are inter-related. A brief description of the elements can be found after the diagram. However, more precision on the different components follows in the next sections.

Section 3 contains a description of the different data structures that hold the packet trace information given by the Analyzer module. These are accesible to the other components in a suitable way. Data related to the scenario is also included in the data structures (e.g. the Notes framework, see below).

Section 4 introduces the main user interface design. The different areas that the DaCoPAn Animator main window shows, their different features and the toolbar needed to control them are presented. Java Swing package `javax.swing` is used for implementation.

Section 5 presents the four different animation panels that the Animator shows. These are the Message Sequence Chart, Unit Flow Orchestration view, Encapsulation view, and Time Sequence Chart. It contains both diagrams and text descriptions on their disposition and behaviour.

Section 6 contains an explanation about the Animation setting classes.

Section 7 explains how the Control Signal framework is in charge of receiving commands from the user and timing events, and calling consequently the appropriate DaCoPAn Animator components, through its different classes and interfaces.

Section 8 contains the necessary information about the Animation Sequence framework, which is how the DaCoPAn Animator manages to present different presentations sequentially.

Section 9 briefly presents how notes to be shown during different animation types are handled by the Animator module. Notes are not a separate module, but are included in the network data model package as a part of the core data structures.

Section 10 illustrates how the Animator parses the packet trace information present in the PEF (Protocol Event File) using existing XML libraries. Still, the PEF reader won't be tied to XML format for further developments of the DaCoPAn software.

Section 11 presents a way to save and load scenario data to and from a file, by representing different settings and configurations made by the user and related to a specific set of packet trace data (to a PEF, basically).

Section 12 describes the localization architecture of the Animator.

2 Architecture

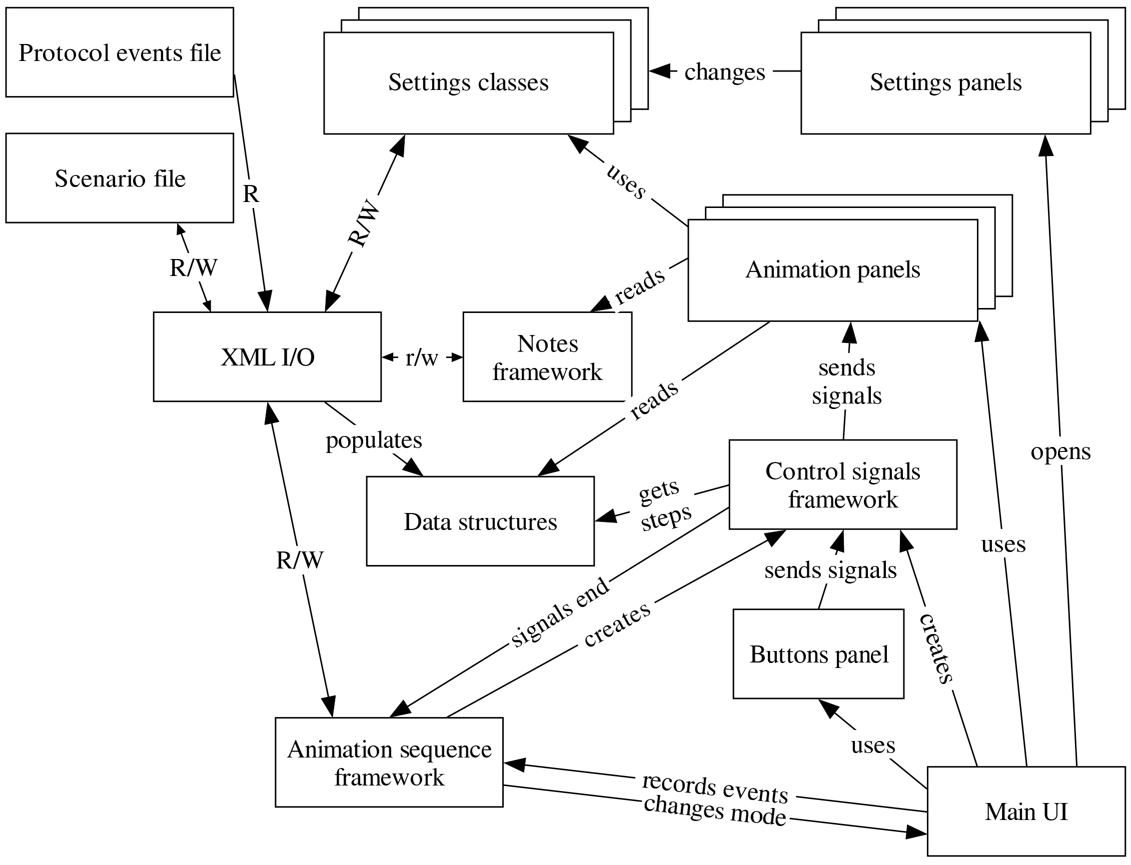


Figure 1: Architecture diagram

2.1 Data structure classes

Store all animation data internally and provide a convenient view to this data for all other components. After the data has been read from the Protocol Events File, the data does not change, so there is no need for update or delete operations, or concurrency problems, at least for network data.

There is also a small amount of data specific to the animation, for example textual notes. This data is mapped to the network data and is accessible also through the view to the data.

2.2 Main UI

Means first of all the classes for the main application frame. The classes are responsible for the layout of the animation components and making most commands available to the user, for example through a menu bar.

This set of classes need to provide methods for changing the view mode between the ENC, MSC and TSC animation types.

The Main UI also provides signals about user actions (mode change, start, stop, etc.) that are needed for recording animation sequence information.

2.3 Animation panels

Individual panels, derived from `javax.swing.JPanel`, for presenting different types of animation. Each panel acts on signals from the Control Signals framework. Each panel queries the necessary data from the data structures. Thus, these panels are independent of each other, and rather passive in terms of user interaction. The individual panels are the MSC, ENC, UFO and TSC. The `NotePanel` is also an animation panel.

2.4 Settings classes for animation

The settings objects are mostly record-like collections of relevant settings for a particular animation type (or mode, e.g. MSC+UFO). The classes also provide methods for validating user input. The objects are clonable to facilitate storing of an animation sequence.

2.5 Panels for changing settings

A set of `JPanels` that can be used by the user to change settings in the settings objects for animation panels. There are individual classes for the MSC and TSC animation types and for general program wide settings.

2.6 Control Signals framework

Provides a modular and uniform way to map control signals from the user (selecting buttons like Play, Pause, To Beginning, To End) and from an animation timer to the animation panels. One or many animation panels may be simultaneously controlled by the framework.

2.7 Buttons panel

An instance of `javax.swing.JToolBar`, that contains the player control buttons for (any) animation, and sends signals (method calls) about user actions to the Control Signals framework.

2.8 Animation Sequence framework

Is able to record and store a sequence of different presentation types (e.g. MSC animation, ENC animation, TSC animation). Recording is done by collecting relevant user actions from the main UI and storing the necessary settings objects. The Control Signals framework signals the end of one presentation type from the **AnimationTimeState** class so that the Animation Sequence framework can make actions to start the next presentation type. The format for storing the sequence is a playlist-type list of different presentations, that can also be presented to the user as a "playlist". The user can jump to any item in the playlist and thus allow using the list as a crude "table of contents" for a scenario.

2.9 Notes framework

The notes framework is part of the network data model classes. It includes the **Note** and **NoteManager** classes, and is described in more detail in section 9.

2.10 XML input/output

Takes care of reading in the Protocol Events File from the Analyzer and populating the data structures with network data as well as saving user edited scenarios. Consists of the **ProtocolEventsReader** interface which handles the input from PEF files, and the Scenario package which handles the serialization of the Scenario data.

2.11 Localization

Means a way storing all localizable resources (mostly strings) in one place where they can all be edited in one place. The mechanism makes it possible to retrieve localizable resources according to the locale in use. The mechanism also allows placing placeholders (for variables) inside strings. The basic mechanism for this is already provided by a standard Java class, `java.util.ResourceBundle`, which is used by **Localization**, an intermediate class for accessing the localized texts.

3 Data structures

3.1 Protocol data

The central data structure class for the protocol data is called **TransferUnit**. A transfer unit is a generalization of a unit transferred on any network layer: for example, an IP packet, a TCP segment, or a HTTP request. An instance of this class encapsulates all the essential data on a unit: source and destination hosts, timestamps (send/receive), variable data (protocol-specific header fields, host variables),

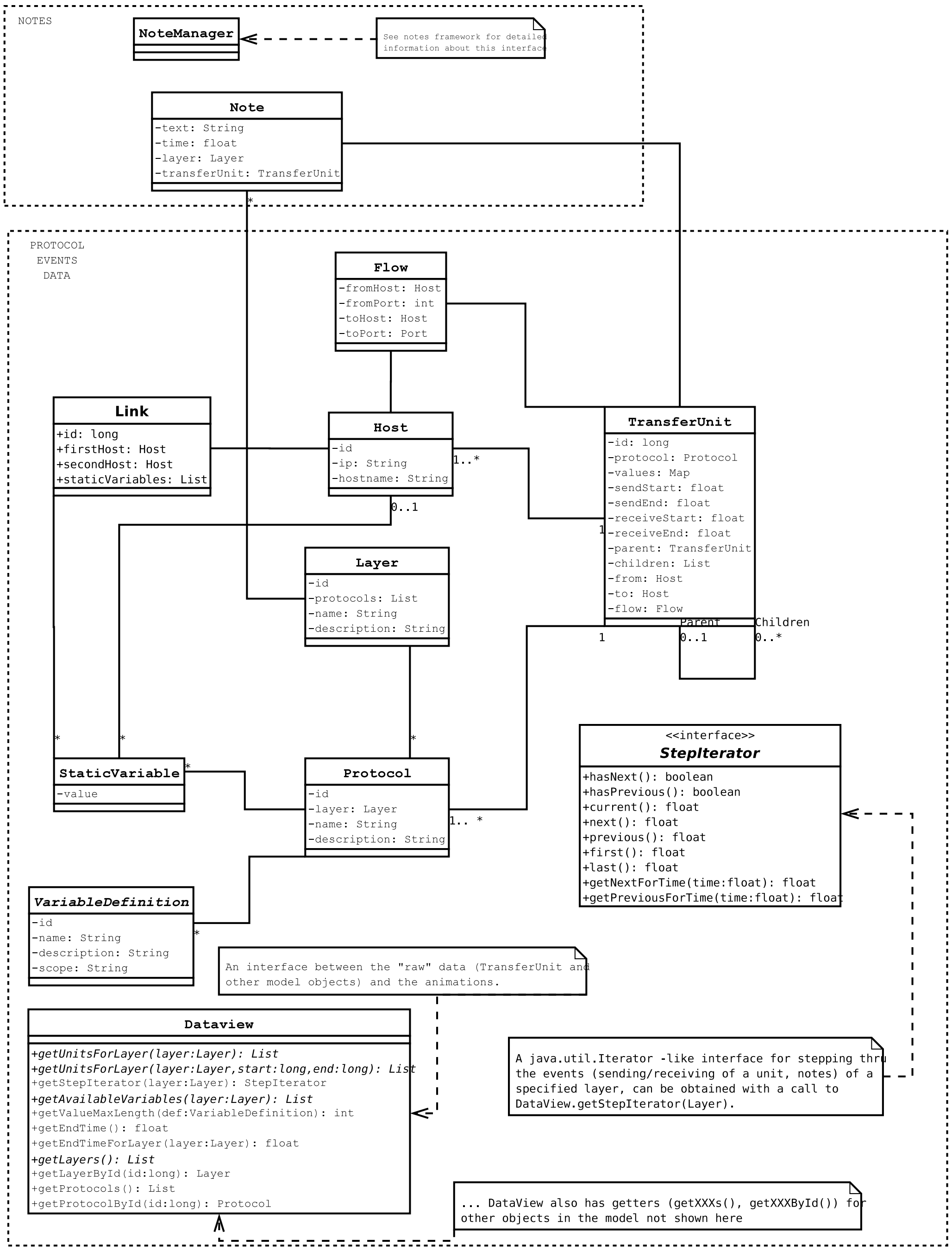


Figure 2: Class diagram for the data structures

etc. The units on different layers form a tree structure (parent-child relationship) that's used to represent the encapsulation that occurs between different layers. For example, if the data of an HTTP response is contained in three separate TCP segments, the TransferUnit representing the HTTP response will have three children (the TCP segments) and each of these children will have the HTTP response as their parent (see Figure 3).

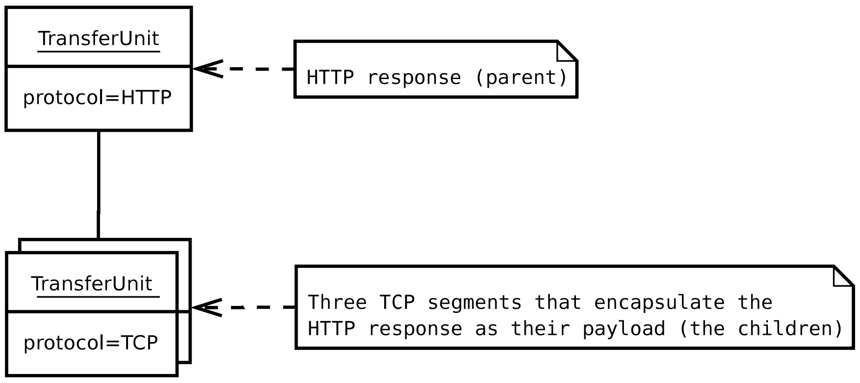

TransferUnit
protocol=HTTP
HTTP response (parent)
TransferUnit
protocol=TCP
Three TCP segments that encapsulate the HTTP response as their payload (the children)

Figure 3: Encapsulation in `TransferUnit` objects

In general, the data structures used to represent the protocol data in the Animator are immutable, that is they will not (or can't) be modified after they have been created. This in turn implies that they are thread-safe, and can be concurrently accessed by multiple threads without using synchronized collection classes, for example.

3.2 Host class

The **Host** class holds information about a single host, containing the hostname (for instance, "A") and the IP number in String format. A host is the network entity which interchanges packets (transfer units) with other hosts. References to hosts are maintained from the `TransferUnit` and the `Connection` classes.

3.3 Flow class

The **Flow** class contains information about a protocol-specific connection between two hosts, for example a TCP connection. Therefore an instance of it makes references to both source and destination hosts, and as well stores information about the port numbers used for this connection to be established.

3.4 Link class

The **Link** class models a physical link between two hosts. It can have constants such as MTU (maximum transfer unit size) associated with it.

3.5 Protocol class

The **Protocol** class is intended for maintaining data about the name and the network layer of a given protocol. One protocol can only belong to one layer. There must be at least one `TransferUnit` for that protocol, and it can have none or many `StaticVariables` and `VariableDefinitions`.

3.6 Layer class

The **Layer** class represents a network stack layer. It is described by the name commonly given to the layer. Many protocols can belong to a same layer.

3.7 VariableDefinition class

This class contains all useful information needed to describe a network variable, meaning that it contains a variable's id (for Animator and Analyzer internal representation), the name of the variable, a short description and the variable scope. Depending on the scope of a variable, this one will be shown in different places during visualization. The name and description attributes are determined by the Localization feature to adapt them to particular languages. A `Protocol` can have many or no variables. The **VariableDefinition** class is be used for those variables that can change throughout the interchange of transfer units between the hosts, for instance representing the value of the congestion window. Those dynamic variables must be mapped to `TransferUnits` as their value changes in relation to the packet interchange. For static variables that are constant all through the packet interchange sequence, the value is represented with a `StaticVariable` instance.

3.8 StaticVariable class

A **StaticVariable** is a particular kind of `VariableDefinition` that is specific to a `Host` and a `Protocol`. This class contains a value attribute that will store the actual value of the network static variable. It is called static as it doesn't change during the packet interchange. It extends from `VariableDefinition`.

3.9 Notes

Notes are separately discussed in, section 9, the Notes framework.

3.10 DataView interface

While a `TransferUnit` object contains all the necessary information that needs to be visualized of a single protocol unit, different animation types need a way of accessing the protocol data as a whole. The `TransferUnit` class has a way of representing the encapsulation between units on different layers, but has no mechanism for returning all the units on a specific layer in a sequential list, for example. This mechanism is needed by the MSC animation type, for example, as it is only interested in units on one specific layer at a time. Thus, there is a need for an interface that can provide different views to the "raw" unit data (`TransferUnit` objects) for the rest of the system.

This interface is called **DataView**. While the actual `TransferUnit` objects are created by the XML I/O classes when reading in the Protocol Events File, the `DataView` interface processes the units further to provide meaningful and efficient ways of viewing the data. For example, it indexes the units by their network layers so that retrieving an ordered list of units on the transport layer is a constant-time operation (see Figure 4). Furthermore, the `DataView` interface has methods that provide a way of accessing only a subset of data on a certain layer by allowing a time interval to be specified. For example, an MSC animation is not interested in drawing any units that would not be visible on the user's screen at a given moment, so it can ask the `DataView` to return only the visible units (the visibility of a unit is determined using its timestamps by the `DataView`).

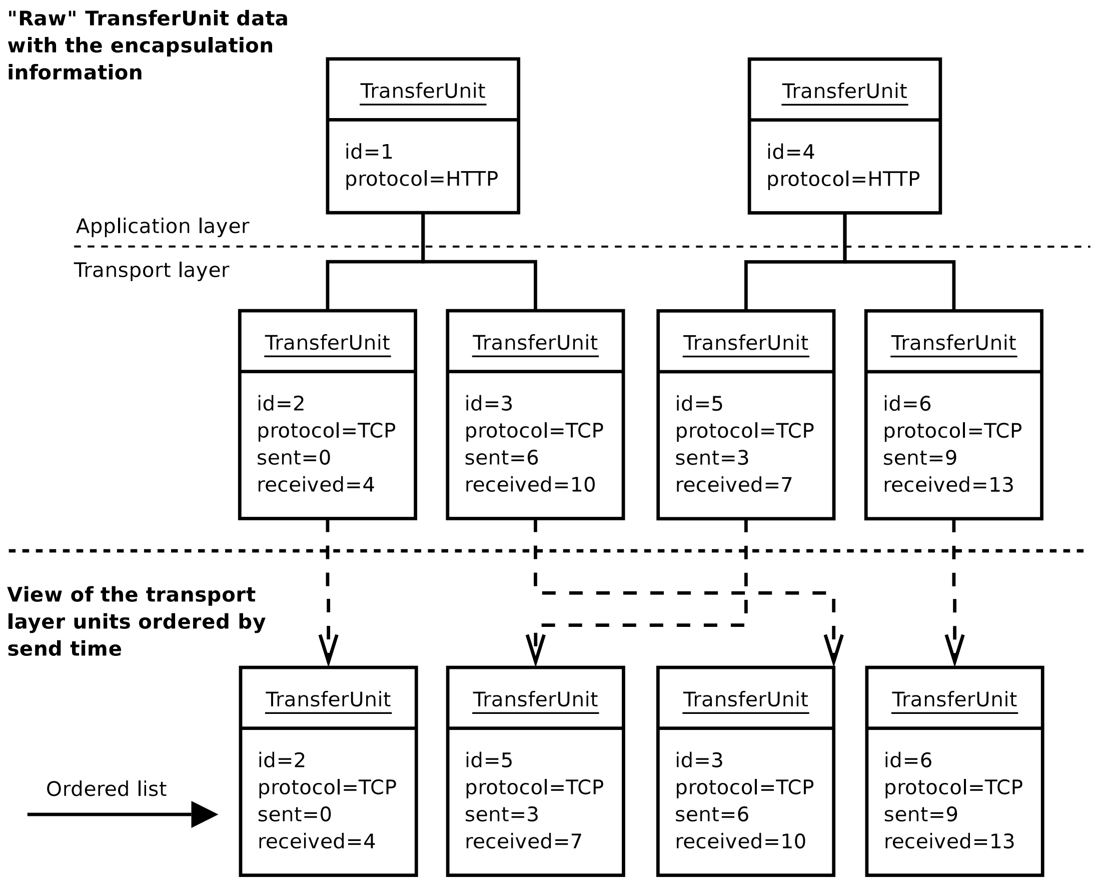


Figure 4: An example view on protocol data provided by the `DataView` interface

3.11 StepIterator interface

For stepping thru the animations (MSC in particular) in a meaningful way, certain steps need to be calculated from the protocol data. A special interface, **StepIterator**, provides the animations with a means of stepping thru the events of a certain layer: sending and receiving units as well as any notes on the layer.

4 Main User Interface design

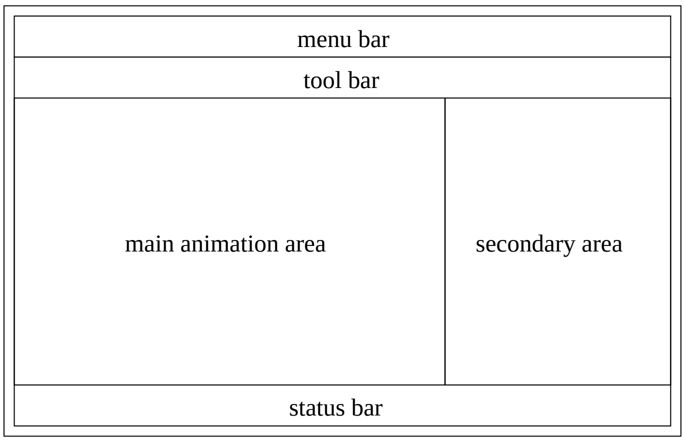


Figure 5: General UI layout for the Animator

4.1 General User Interface design

The main window of the DaCoPAn Animator, see Figure 5, is divided into the following parts:

- **main animation area:** Basically contains one of the following panels: ENC, MSC or TSC.
- **menu bar:** Contains the program specific menu items.
- **secondary area:**. In the MSC, the area is divided between a Unit Flow Orchestration animation panel and a note area. In the TSC, it is occupied by a unit info panel and a shared panel for notes and the chart legend.
- **status bar:** Shows information about the state of the Animator.
- **tool bar:** Enables the user to control the flow of animation.

Most of the animation occurs in the main animation area. This animation is a sequence of Message Sequence Chart (MSC) type animation that is occasionally interrupted with some Encapsulation (ENC) animation. Alternatively, the main animation area shows a Time Sequence Chart (TSC) animation. The animation can be watched in two modes: scenario mode and explore mode. In scenario mode the animation follows a predefined script and in explore mode the user is in charge of controlling the animation flow.

The main animation area takes up most of the screen width and is positioned on the left side of the program frame. The right side is used by the secondary area and is divided vertically in two parts. The areas are divided using Swing `Splitpanes`, so that the user can easily drag them to a preferred size. The animation panels then know how to draw themselves according to the space given to them.

When starting the Animator, no animation panels are shown as there is no file loaded to the memory. Instead a welcome screen is shown in the main animation area.

4.2 The functionality of the main UI classes

The DaCoPAn animator is based on the Swing framework, which means that the control over the software runs always through Swing, which in turn gives the control over the other parts of the program to `MainFrame` (i.e. the main UI classes). When the Animator is started, an instance of `MainFrame` is created. It then creates the user interface using the class `UserInterface` and instances of all other classes needed by the different animation modes: `DataView` for accessing the data structures, `NoteManager` for using notes, `AnimationSequence` for storing scenario information. An instance of `AnimationTimeState` is created when the animation is started.

After this the Animator is in a state called DACOPAN_STATE_NO_FILE and in a mode DACOPAN_NO_MODE, which indicates that the animator doesn't do anything before the user uses the menu or tool bar to open a file.

`MainFrame` can be in the following modes:

- DACOPAN_NO_MODE: Used when no animation file has been loaded.
- DACOPAN_EXPLORE_MODE
- DACOPAN_SCENARIO_MODE

`MainFrame` also has three distinct states:

- DACOPAN_STATE_NO_FILE: Used when no animation has been loaded.
- DACOPAN_STATE_PLAYING_ENC: Used when the animator is showing Encapsulation animation.
- DACOPAN_STATE_PLAYING_MSC: Used when the animator is showing MSC animation.

- DACOPAN_STATE_PLAYING_TSC: Used when the animator is showing TSC animation.

Distinction between play and pause mode is done by `AnimationTimeState` and therefore those are not distinct states from the point of view of `MainFrame`.

To show the relations between `MainFrame`, other user interface classes and the rest of the Animator, a class diagram is presented in Figure 6.

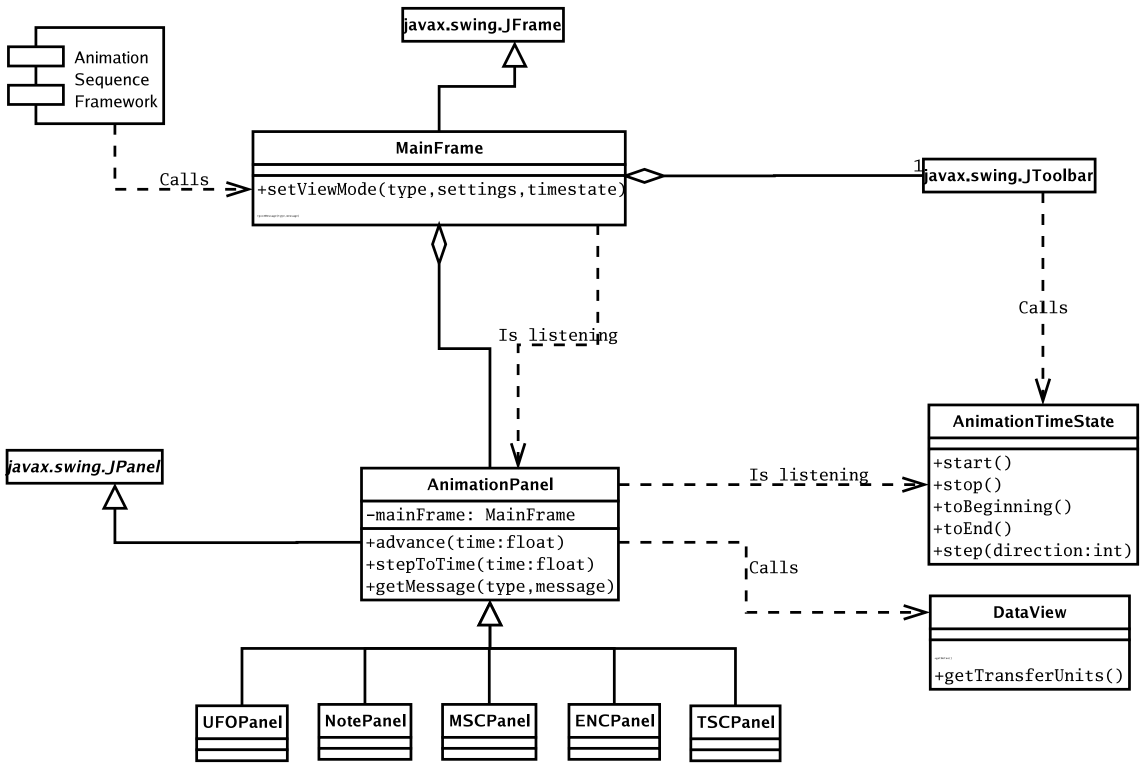


Figure 6: Main UI Class Diagram

4.3 User actions

The menu bar contains the following commands for letting the user control the Animator. Most of them are also present in a tool bar on the top of the main frame.

Open file This command opens a dialog for selecting a Protocol Events File or a Scenario file to be loaded to the Animator.

Save file Clicking on this button saves the file in memory along with any note and scenario data included. If the file has not been previously saved as a Scenario file, the user is prompted for the file to save.

Settings This command opens a settings dialog that is described in more depth in, section 6, the Animation settings.

Save all settings This command is available in the Settings menu and saves all user specific settings globally.

Rewind to the beginning This buttons rewinds the animation to the beginning of the current `AnimationTimeState`. This command can only be called when the animation is paused.

Play Calls the current `AnimationTimeState` to start playing the animation it is assosiated with. If the current sequence is part of a bigger scenario, the `AnimationTimeState` notifies the animation sequence about reaching the end of the animation.

Pause Pressing this button changes the animation to paused mode.

Fast forward to the end This button steps to the end of the animation data assosiated with the current `AnimationTimeState`.

Step forward When in stop/scroll mode, this button can be used to step forward through events one by one in the animation.

Step backward When in stop/scroll mode, this button can be used to step backward through events one by one in the animation.

Animator mode selection The Animator can be in either explore or scenario mode. The user can do this selection in the menu. In the status bar the currently selected mode is highlighted and in scenario mode a small window for controlling the scenario and recording a play list is presented. When this window is closed, `MainFrame` minimizes it to the status bar as a button for restoring the window.

Layer selection The user can select the layer that is currently shown by clicking the button of the layer he wants to see either in the menu or on the tool bar. When such a button is pushed, all animation panels are recreated with correct settings.

Language selection When the user selects a new language, the whole user interface is recreated using the strings from the selected language. This is done by calling `UserInterface` method `recreateUI`.

Help The help command opens a new window with an HTML help document shown in it.

About The about command opens a window with tabs for general information, licence information and information on the team behind the product.

5 Animation panels

All the animation panels that can be placed on the four animation panel spaces in the Animator have to extend the abstract class **AbstractAnimationPanel**. This class handles some aspects of the communication between the panels and the `MainFrame`. It also implements the interface `ControlSignalListener` that allows the panels to receive control signals from the Control Signal framework.

In this version of the Animator five animation panels are created:

- Message Sequence Chart
- Time Sequence Chart
- Encapsulation
- Unit Flow Orchestration
- Notes

The animation panels are described in more detail in the following subsections.

5.1 Message Sequence Chart

The Message Sequence Chart (MSC) animation type, see Figure 7, is one of the two primary animation types for the DaCoPAn2 Animator. It shows unit transfers in "text-book" style, that is, by drawing lines from one host to another. The y-axis represents time and the gradient of a line is proportional to the transfer delay of a unit. The central column is the drawing area, where the lines are drawn. On both sides of the drawing area there are up to two optional columns, where timestamps and other information is displayed. The third column, the notes column, is always visible. Above the columns there is a host field, which lists the names for hosts A and B.

The blue line is the progress line, which scrolls down the display area as the animation proceeds in play mode. The display area is vertically scrollable. Also, when the progress line is moved away from the drawing area with the mouse, via click and drag, past the top or bottom panel border, the MSC panel will scroll.

The MSC animation style is created by class **MSCPanel**, which implements the interface **AbstractAnimationPanel**. Because of this `MSCPanel` also implements those methods that are listed in section 7.2.

The properties and functionality of the MSC animation type are explained in the further subsections.

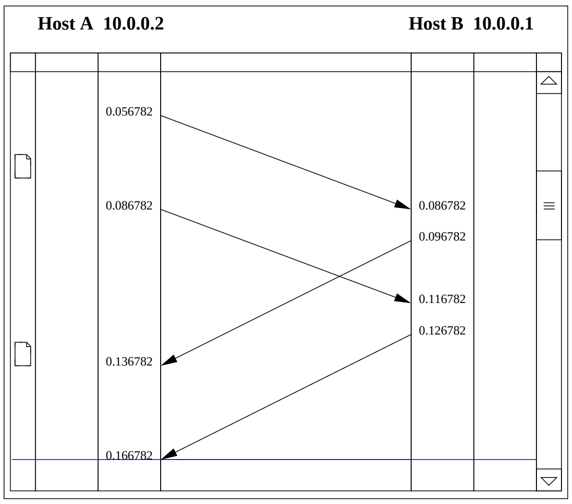


Figure 7: General layout of the MSC animation type

5.1.1 Columns

The MSC presents symbolic information (text and numbers) in vertical columns. The most important column is the timestamp column. That column shows the animation time for the adjacent event (i.e. the sending or receival of a packet). The user can also choose to view other data in columns. This is done by showing many variables in different rows of one column and identifying those variables by a caption at the beginning of the row. The selection and number of columns is configurable through the MSC settings dialog. The order of columns is not configurable and the timestamp column will always be drawn nearest to the center.

The columns are mirrored for both of the hosts, so that exactly the same columns and variables are shown for both hosts. The showing order of columns is also mirrored, so that the column nearest to the drawing area always presents values of the same variables on both sides of the drawing area.

The columns are generated by the **MSCColumnModel** class, which is an inner class of `MSCPanel`. The width of the columns is determined by the maximum length (in characters) of the values in that column. This information is queried from the `DataView` class using its method `getValueMaxLength()`.

5.1.2 Drawing area

The drawing area column is created by the **MSCDrawingPanel** class, a private innerclass of the `MSCPanel`. It is where the actual animation of the application, network and transport layer message sequences is shown. This is accomplished by drawing a straight arrow headed line from a point on one side of the drawing area column to another point on the opposing side of the column, which represents the unit being transmitted through the network from one host to another. The arrow head distinguished the direction of transmission between the hosts. Dropped packets are marked with a red cross, which is drawn close to the opposing end of the column from the sending host's side. In addition to this, via the settings, the user can select the unit's variables to be drawn above the line. The font size for this text can also be chosen from the settings.

See subsection 5.1.4 for more detail on the implementation of the drawing area.

5.1.3 Notes

The presence of a note is signaled by showing a small icon in a notes column that is always present as the left-most column next to the left edge of the component. (That column is never mirrored, as there is only one notes column.) The top edge of the icon corresponds to the exact location of the note on the y-axis. The actual note text is displayed in the Note panel, situated on the bottom half of the secondary area of the main window.

See subsection 9.1 for more information about MSC notes.

5.1.4 Improving the readability of the MSC

One of the main problems with the previous DaCoPAn version was in representing a "clear" visualization of data for the MSC animation type. The problem arises from the fact that, usually, there are lots of packets sent within a very short period of time. The scaling for the y-axis (time) would have to be in the order of microseconds (μs), to distinguish separate events from each other in the drawing area. Additionally, the separation of events ranges from atleast 2 μs to about 700 μs. This leads to overlapping lines if the scale is too large, as seen in Figure 8, or in lines that have nearly vertical gradients. In the latter case the scale is simply too small, which causes a network transmission of 2 seconds to stretch over several hundred A4 lengths in the drawing area. Clearly neither is acceptable.

The constraints imposed on the improvement of the MSC animation view are hence:

1) The scaling, separation of adjacent packets, for the vertical time coordinates for the animation view is limited.

2) For packets with very lengthy transfer delays, the gradient for drawn lines is excessively large (almost vertical), depending upon the scaling used.

As both constraints make it very difficult to read the MSC diagram and distinguish separate events, the DaCoPAn2 projects version of the Animator allows the user to force the program to leave some space between consecutive events. It also allows the user to force a maximum length for the separation of both points of a single drawn line. This, of course, removes the strict linearity of time for the y-axis. This modified animation mode is referred to as the "non-linear time" mode, whereas the previous version is called the "true time linear mode". In Figure 9, a view of the non-linear mode for the MSC animation type which distinguishes packets is presented.

As mentioned above, the DaCoPAn2 project must be able to improve the readability

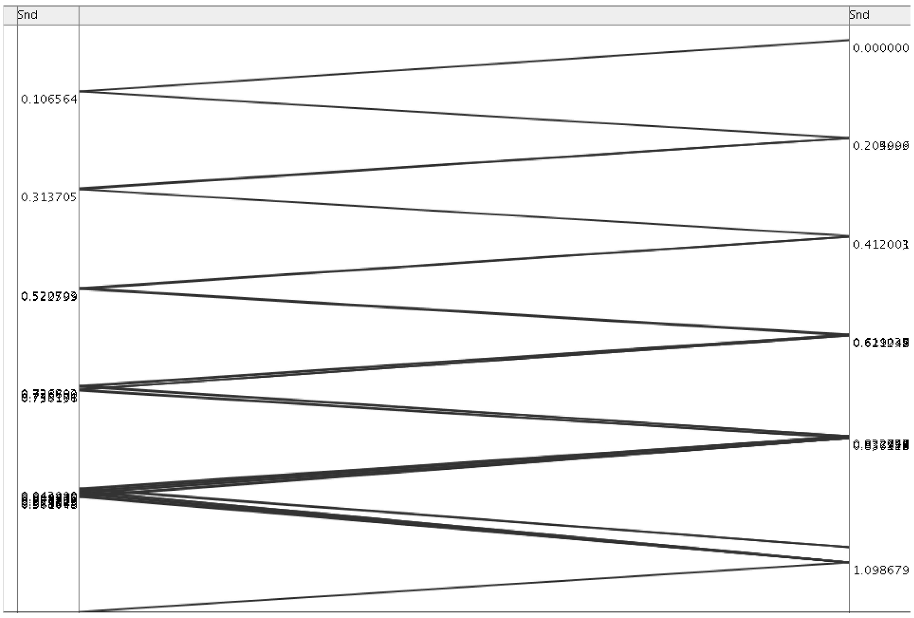


Figure 8: MSC animation type with a "true" linear time scale

of MSC animation view. This is accomplished by adding a configurable minimum stepping between any two events. That is, no two events can occur on the y-axis within a space called "minimum stepping".

Another improvement for readability is that packets that have excessively lengthy transfer times are forced to arrive within a vertical length called "maximum stepping". If the packet is received more than "maximum stepping" pixels after it was sent, its receiving point is transfered to a "maximum stepping" distance from its sending point. However, should this cause collision with the "minimum stepping" then the receiving point is pushed further. In other words, "maximum stepping" is always inferior to "minimum stepping". All events happening before the packet was originally received, but after its new receiving point, are transfered upwards so that the order between events is preserved.

The third improvement to the readability of the MSC is so called "gap cutting". That is, if two following packets are sent too far from each other (measured in pixels), the empty gap is cut away. Thus this cutting is done only for empty space (all sent packets are either dropped or received) between two packets if this empty space is larger than "maximum gap". The empty space is calculated from events.

From the mentioned variables "minimum stepping" and "maximum stepping" are user configurable, but "maximum gap" is not. There are also a few constraints for each. "Minimum stepping" cannot be smaller than the font size multiplied by the number of rows in the multi-row column (if it is chosen to be shown). This constraint is to prevent the text of two packets to overlap each other and thus making it impossible to read them. "Minimum stepping" cannot either be larger than "maximum stepping" (and vice versa) for obvious reasons. "Maximum gap", on the other hand, must be greater than "maximum stepping". This constraint is because of the implementation

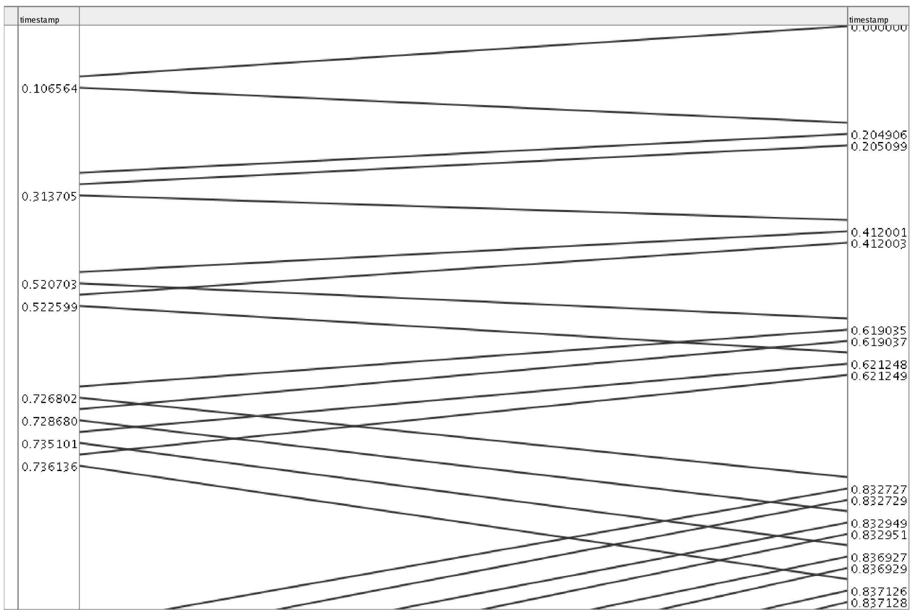


Figure 9: MSC animation type with a non-linear time scale

of the algorithm. This is all facilitated by the **CalcYCoord** class described in the next subsection (5.1.5).

Additionally, the progress line must progress with constant speed which the user can configure. This is because when using a non-linear time scale, the progress line has nothing to do with "real " animation time and an alternative method is required to control the progression of animation.

If the user wants to see the linear time scale, the "minimum stepping" can be set to zero and "maximum stepping" (and thus also "maximum gap") to infinity (which is implemented as some number which is greater than any possible stepping).

5.1.5 The CalcYCoord class

The algorithm that calculates y-coordinates, which satisfy the requirements listed in section 5.1.4, is implemented as class **CalcYCoord**. Events from one layer are given to this class with a maximum and minimum stepping and it calculates new y-coordinates for events.

In addition to the constructor, this class has few public methods:

CalcYCoord(units:List, settings:SettingsMSC) The constructor reads events for a specific layer in `units`. It also queries `settings` for settings that are important (i.e. maximum stepping, minumum stepping and visual scale). Calls `calculateYCoord()` in order to actually calculate y-coordinates.

calculateYCoord() Calculates y-coordinates for events. The algorithm is shown as a pseudo-code in Table **??**. This method must be called every time when something interacting with y-coordinate changes.

getYCoordForTime(time:float):int Returns the y-coordinate for given `time`. If some event has the same time as the argument, it returns the exact y-coordinate. Otherwise, it calculates linear interpolation based on y-coordinates of the nearest events.

getLinearYCoordForTime(time:float):int Returns the y-coordinate, such that the ratio between it and the last y-coordinate of events is equal to the ratio between the given `time` and the time of the last event.

getTimeForYCoord(yCoord:int):float Returns the time for a given y-coordinate. Acts as inverse function to `getYCoordForTime`, that is

$$t = \texttt{getTimeForYCoord}(\texttt{getYCoordForTime}(t)).$$

getLinearTimeForYCoord(yCoord:int):float As with `getTimeForYCoord`, but for `getLinearYCoordForTime`.

```
CALCULATEYCOORD
Let $L$ be a sorted (by time) list of events.
for each $e \in L$ do
  $e.yCoord \leftarrow e.time * visualScale$
done
for each $e \in L$ do
  if $e.transferTime > maxStepping$ then
    $e.transferTime \leftarrow maxStepping$
    for each event $f$ that happens while $e$ is transmitted do
      if $f.time > e.time$ then
        $f.time \leftarrow e.time$
      end
    done
  end
done
for each $e \in L$ do
  $f \leftarrow e.nextEvent$
  if $|f.yCoord - e.yCoord| \leq minStepping$
   and $f$ and $e$ are not events for same transfer unit then
    $f.yCoord \leftarrow e.yCoord + minStepping$
  else if $f.yCoord < e.yCoord$ and $f$ and $e$ are events for same transfer unit then
    $f.yCoord \leftarrow e.yCoord$
  end
done
for each $e \in L$ do
  $f \leftarrow e.next$
  if $f.yCoord - e.yCoord > maxGap$ then
    $gap \leftarrow f.yCoord - e.yCoord - maxGap$
    for each $g \in L$ and $g.yCoord > e.yCoord$ do
      $g.yCoord \leftarrow g.yCoord - gap$
    done
  end
done
```

Table 1: An algorithm to calculate proper y-coordinates for the MSC

5.2 Encapsulation

The Encapsulation animation occupies the main frame when it is activated. The UFO panel is cleared and disabled, and the Note panel displays the note of the specific ENC animation.

The ENC animation is a static diagram which shows a snapshot at a specific time. The user actions for controlling the animation via the Animation menu and the tool bar buttons are disabled.

5.2.1 Note framework for ENC

Each ENC animation can contain a single note. The note is bound to the `TransferUnit` from which the ENC diagram is constructed. The note can be retrieved from **NoteManager** by calling the method `getEncNote(TransferUnit)` with the active ENC unit as the parameter.

5.2.2 The ENC tree model

The generation of the encapsulation tree for an ENC animation is the responsibility of a class called **ENCTreeModel**. This class contains all the logic needed to select the most interesting units in the encapsulation tree of a selected unit for display in the animation. By default the tree model produces ENC trees with a maximum of three units on the same level.

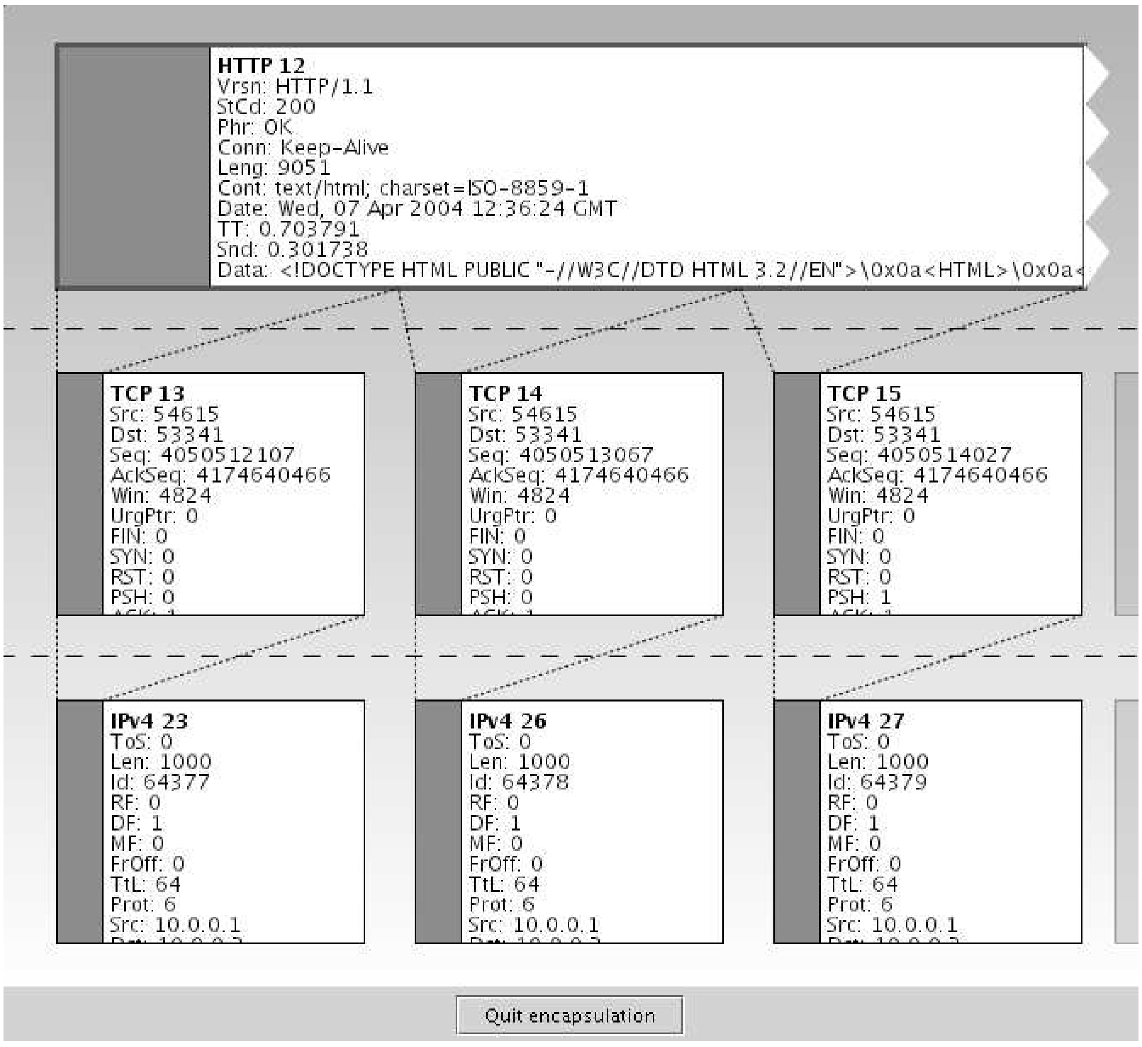
5.2.3 General layout

The ENC panel consists of three areas, corresponding to the different layers in the communication. The application layer can contain up to one unit, and the transport and network layers can be populated with up to three units. If the application layer contains no unit (this corresponds to showing protocol exchanges where the application level protocol is not recognized/supported) then the transport layer, being the highest layer, contains exactly one unit. However, the design of the encapsulation animation is flexible enough to support layers other than the three usually present in scenarios.

As an example of a three-layer encapsulation diagram, see figure 10.

A unit contains a fixed width field variable area for displaying unit variable values. The rest of the available width is used for a payload area. The physical widths of the areas are not in relative proportion. On lower layers lines are drawn to the right side of the encapsulating unit on the next layer to designate the composition of the unit.

At the bottom of the ENC panel there is a Quit encapsulation button for returning to the MSC mode and continuing with the scenario.



HTTP 12
Vrsn: HTTP/1.1
StCd: 200
Phr: OK
Conn: Keep-Alive
Leng: 9051
Cont: text/html; charset=ISO-8859-1
Date: Wed, 07 Apr 2004 12:36:24 GMT
TT: 0.703791
Snd: 0.301738
Data: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">\0x0a<HTML>\0x0a

TCP 13
Src: 54615
Dst: 53341
Seq: 4050512107
AckSeq: 4174640466
Win: 4824
UrgPtr: 0
FIN: 0
SYN: 0
RST: 0
PSH: 0

TCP 14
Src: 54615
Dst: 53341
Seq: 4050513067
AckSeq: 4174640466
Win: 4824
UrgPtr: 0
FIN: 0
SYN: 0
RST: 0
PSH: 0

TCP 15
Src: 54615
Dst: 53341
Seq: 4050514027
AckSeq: 4174640466
Win: 4824
UrgPtr: 0
FIN: 0
SYN: 0
RST: 0
PSH: 1

IPv4 23
ToS: 0
Len: 1000
Id: 64377
RF: 0
DF: 1
MF: 0
FrOff: 0
TtL: 64
Prot: 6
Src: 10.0.0.1

IPv4 26
ToS: 0
Len: 1000
Id: 64378
RF: 0
DF: 1
MF: 0
FrOff: 0
TtL: 64
Prot: 6
Src: 10.0.0.1

IPv4 27
ToS: 0
Len: 1000
Id: 64379
RF: 0
DF: 1
MF: 0
FrOff: 0
TtL: 64
Prot: 6
Src: 10.0.0.1

Figure 10: The Encapsulation diagram

5.3 Unit Flow Orchestration

The UFO panel is designed to visualize the message sequence data as a continuous animated flow of units moving between hosts. It has two channel areas for animating the movement of the units and two unit variable fields for showing the data that is contained by the selected transfered units, one for each direction. See figure 11, for the main layout of the UFO panel.

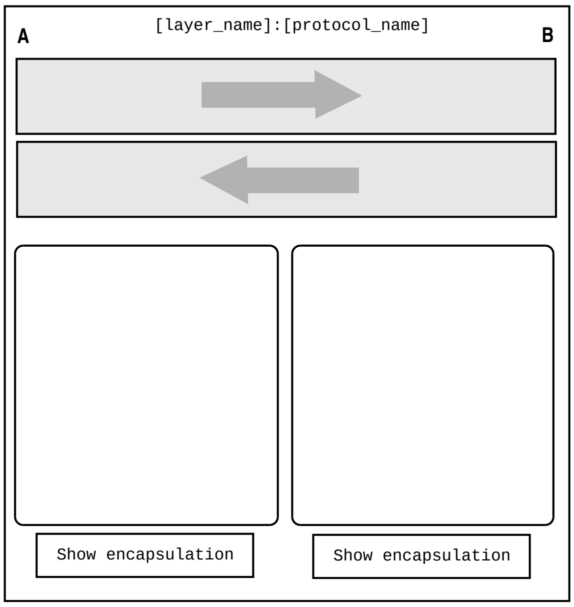


Figure 11: Empty UFO panel displayed in the ENC view

The UFO panel is a subclass of **AbstractAnimationPanel** which in turn implements the **ControlSignalListener** interface. This means that it has the method `stepTo()` in which it is asked to update to a new animation state. When the UFO panel is drawn, it calculates the position for each unit and draws them to the correct place on the appropriate channel. To synchronize the UFO animation with the MSC animation, the UFO uses the `CalcYCoord` class to get the Y coordinates of the send and receive events of units in the MSC panels, and uses them to calculate the units' X coordinate on the channel. Because the UFO panel is synchronised with the MSC panel, the amount of units drawn on top of each other is minimized, but when unit lines intersect in the MSC animation, the corresponding units are also drawn on the exact same place in the UFO animation.

The data of the active units is shown in the data areas. When a new unit appears to the visible area, it is selected as the active unit. The selected unit stays selected until it is received. Units are activated automatically when they are created, so that the latest unit is always the default active unit.

Another important use for the UFO panel is that it can be used to move into encapsulation mode. There are two buttons under the data fields that can be used to show the encapsulation for the selected unit. When a unit is active, this button can be clicked. The main frame switches to showing encapsulation instead of MSC animation.

The two channels (pipes) contain only one color of units each, which have a darker border when they are active. Dropped packets are marked with a red cross, and disappear when they arrive at the center of the channel.

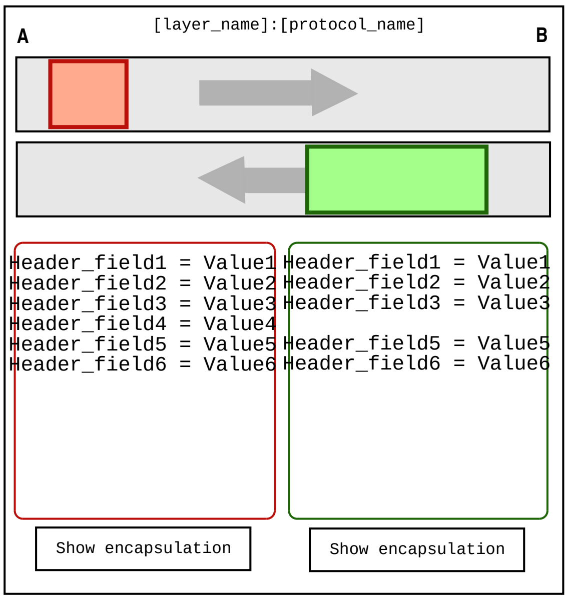


Figure 12: UFO panel in play mode

5.4 Time Sequence Chart

The Time Sequence Chart (TSC) is an animation type, which shows detailed information about the status of transport layer communication between two hosts. Its main function is to illustrate TCP layer communication. The TSC animation doesn't use the same Control Signal framework (**ControlSignalsListener** interface) that the other animation panels use, since the other animation types are time based, which the TSC is not. The TSC component has its own thread to animate the display. The animation is "user friendly", which means that the Animator draws graphical elements using delays between draws. For example, when a notice is attached to a drawn transfer unit, the delay amount is determined from a "event/second" slider in the TSC settings.

In Figure 13, a sketch of the animation panel is presented. The animation panel is divided into four components: the drawing area and tool bar, tabbed notes and legend panel, notice bar, and unit info panel. The animation panel and its components are explained in the further subsections.

The horizontal dimension in the figure is the time axis, with the positive direcction being from left to right. The vertical dimension is the sequence number of the packet, which increases from bottom to top.

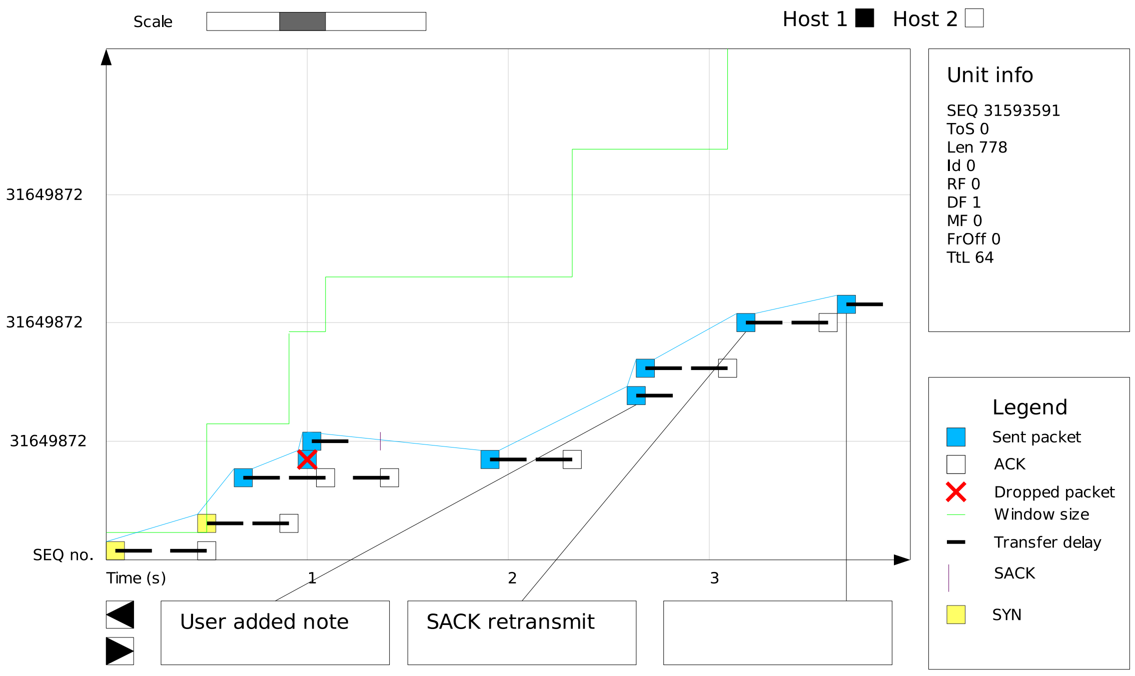


Figure 13: User interface of the TSC animation type

A class diagram is presented in figure 14. The `MainPanel` class is the main TSC component, which extends `javax.swing.JPanel`. It contains the public methods to control the TSC animation thread. The `DrawingArea` class is responsible for drawing the transfer units and other graphical elements. It also has references to `PacketInfoPanel` and `NoteBar`, for updating them as the animation proceeds. The classes `Xaxis` and `Yaxis` draw the axis labels to positions given by `DrawingArea`. All data is read from the `DataView` object.

5.4.1 Drawing area

The drawing area component, class **DrawingArea**, is where all the actual drawing of animation data from the PEF takes place. The drawing area is scrollable. The horizontal dimension (x-axis) is time, with the positive direction being from left to right. The vertical dimension (y-axis) is the sequence number of the packet, which increases from bottom to top. The scaling for the axes is adjusted using scale sliders. The scale sliders are located in a separate toolbar area, which can be minimized to give more space for the drawing area.

The toolbar is part of the main animation area, which is split by a Swing `SplitPane`. It contains two sliders and a host chooser radio button. The sliders adjust the x- and y-axis scales. The host chooser button determines the direction of data flow between the hosts, which is then animated in the drawing area.

When using scale 100% the drawing area contains all events of the data file (PEF/SCE),

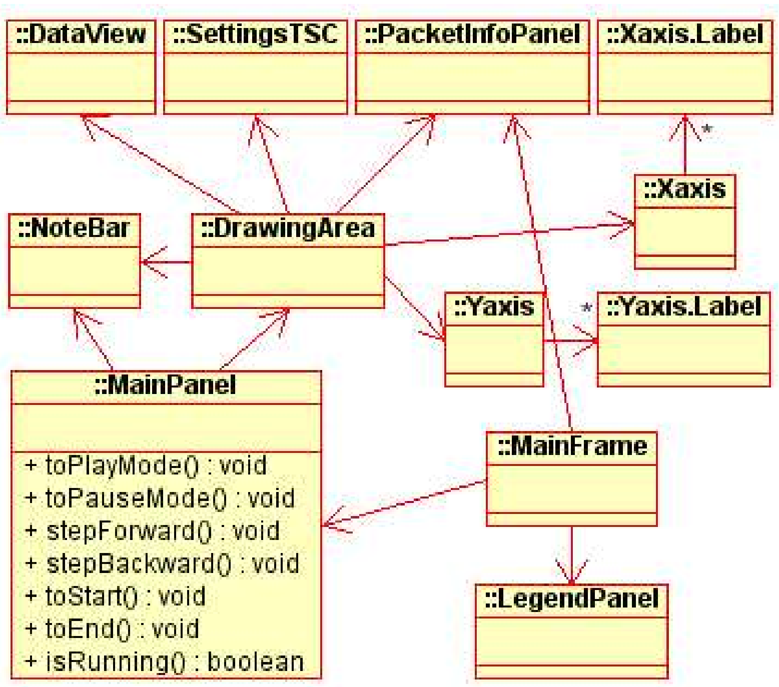


Figure 14: TSC class diagram

for the given axis. With the smallest possible scale, 0%, only a few elements (events) are displayed. Additionally, the time scale is automatically adjusted as follows. If the window's size for the DrawingArea is more than one second, a seconds scale (s) is used. If time scale is below one second, a millisecond scale (ms) is used. Below one millisecond a microsecond scale (μs) is used.

The drawing area contains the following elements:

- **Transfer units:** The TCP transfer units are symbolized by filled-in blue squares and dropped packets are marked by crossing over the unit in question with a red cross ("x")..
- **Connecting lines of units:** Consecutive units are connected together by a straight blue line, which is drawn from the upper left vertice of the unit (rectangle) to adjacent units.
- **Unit transfer delay line:** The transfer delay of each unit is visualized by a dark horizontal line, which is drawn from the middle of the corresponding unit to the point in time where it has been fully transmitted.
- **ACK:** ACK packets are shown as empty squares. For ACK packets, the transfer delay line starts when the ACK is sent from the remote host and ends in the middle of the ACK unit rectangle which depicts the receival of the ACK.

- **SACK:** The SACK blocks are drawn as vertical violet lines directly above the corresponding ACK packet. The lines' vertical coordinates represent the sequence numbers of the SACK blocks.
- **SYN:** The packets with SYN flags are highlighted using a diffrent color from other units, chosen by the user from the settings. The default color of the units is yellow.
- **Window size:** The window size of the receiver is represented by a continuous green line which grows in discrete (ladder-like) steps. The distance from the ACK packet to the horizontal level of the line directly above it represents the actual window size.
- **Grid:** The grid is drawn using horizontal and vertical lines drawn at regular intervals, which are dependent on the scaling used for the axes.
- **Crosshair:** The crosshair consists of a dashed horizontal and vertical line, which intersects over the highlighted unit. During animation, the latest drawn packet is highlighted. The user can also choose the highlighted packet with the mouse, by clicking on it.

5.4.2 Legend

The Legend panel is located in the same tabbed panel as the Notes panel. It contains the description for the used graphical symbols. The legend contains only those symbols, which are selected for display from the settings.

5.4.3 Notes

The Notes panel is located in the same tabbed panel as the Legend panel. Notes use the same Notes framework as in the MSC view. The text in the Notes panel, if any, is specific to the active unit. TSC notes are independent from MSC/ENC notes, meaning that separate texts are shown in each view.

See subsection 9.3 for more information about TSC notes.

5.4.4 Notices

The notice bar component, generated by class **NoteBar**, displays the automatically generated or user added notices. Notice bar is scrollable, since it may contain more notice elements than fits into the area. The notice bar only contains notice elements corresponding to current events, which are visible in the drawing area.

Below the drawing area is a notice bar component (NoteBar class), which displays the automatically generated or user added notices.

Notices are attached to transfer units using connecting lines. Since the notice bar and drawing area are separate Swing components, the line drawing is done in the parent component `MainPanel`.

Notices (aka. the balloons) are clearly distinguishable from notes, which are in the same tabbed panel with the legend table. The active unit's notice text, if any, is duplicated in the Notes panel. Notices are edited through the Notes panel in the tabbed panel.

See subsection 9.3 for more information about TSC notes/notices.

5.4.5 Unit Info panel

The Unit Info panel, generated by the **UnitInfoPanel** class, contains the variables of the active transfer unit. In addition to transport layer data, it can display network layer data. The displayed data variables are configured from the TSC settings. The user may select the unit to be displayed by clicking on it in the drawing area. Additionally, the active unit changes during animation by using the step-forwards and step-backwards buttons. The active unit is highlighted in the drawing area using a red border.

6 Animation settings

The animator has to keep track of the settings that the user has chosen to be used to visualize the different events contained in the Protocol Events File. In this section, the classes implementing the settings are described, and an explanation of the settings user interface panels is given.

6.1 Settings class structure

The user can access the settings using the dialog `SettingsPanel` in which the other settings panel classes are embedded. All the settings panels are `javax.swing.JPanels`.

The settings are stored in the settings data classes. See Figure 15, for a class diagram of all the relevant classes. In the diagram, the classes above the dashed line are part of the user interface, and the classes below it are the settings data classes.

The settings dialog is implemented as the class **SettingsPanel**, which is responsible for setting up all the other settings panels, where the actual settings are made. It includes a selector where the user can browse between settings for the Message Sequence Chart and the Time Sequence Chart animation types and general settings. It also includes an Apply-button, which immediately saves all settings made in the currently open settings panel, and updates the animation settings using `MainFrame.refreshSettings()`.

The settings dialog sets and queries the fields of the settings data classes through

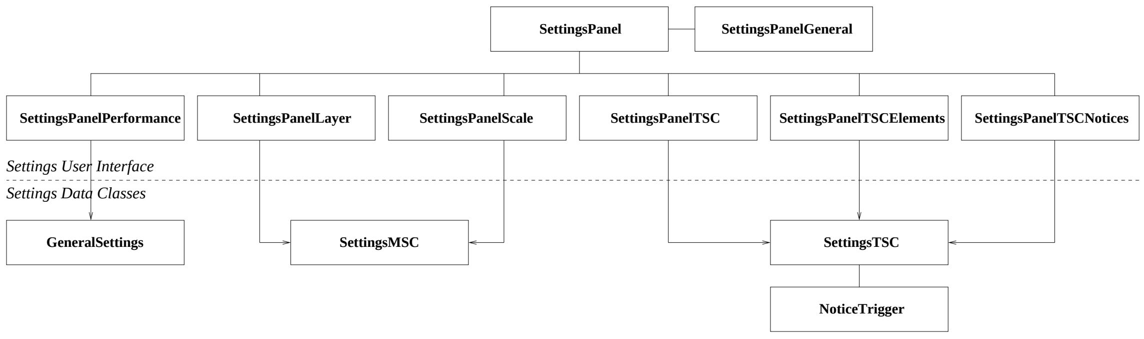


Figure 15: Settings class diagram

their public methods. The actual animations use the getter methods to query the settings. These methods are visible in Figure 16, which illustrates the data classes in more detail. Notice that **GeneralSettings** implements the interface `Saveable`, which allows the Animator to save the settings in a file using the **ScenarioFile** class. (See Section 11.)

6.2 General settings

There are some generic animation settings that are not related to any individual animation type. These values are related to general appearence of DaCoPAn2 animator. General settings include language selection and the possibility to save all settings and restore default settings. These settings are made in the panel **SettingsPanelGeneral**. Language settings, see section 12, are not saved in the settings data classes.

The feature to save all settings is implemented by serializing the settings objects into a global settings **ScenarioFile**.

6.3 MSC settings

When viewing the MSC settings, the settings panel is separated into the following tabbed panels:

- Host variables settings. (**SettingsPanelLayer**) These include choosing both host and protocol variables which are shown in columns and along the packet lines in the drawing area, respectively. Currently possible layers are ARP, Network, Transport and Application.
- Scale-specific settings. (**SettingsPanelScale**) These include the choice between linear and non-linear time scale and their respective settings.
- Performace settings. (**SettingsPanelPerformance**) These include settings for anti-aliasing and refresh rate.

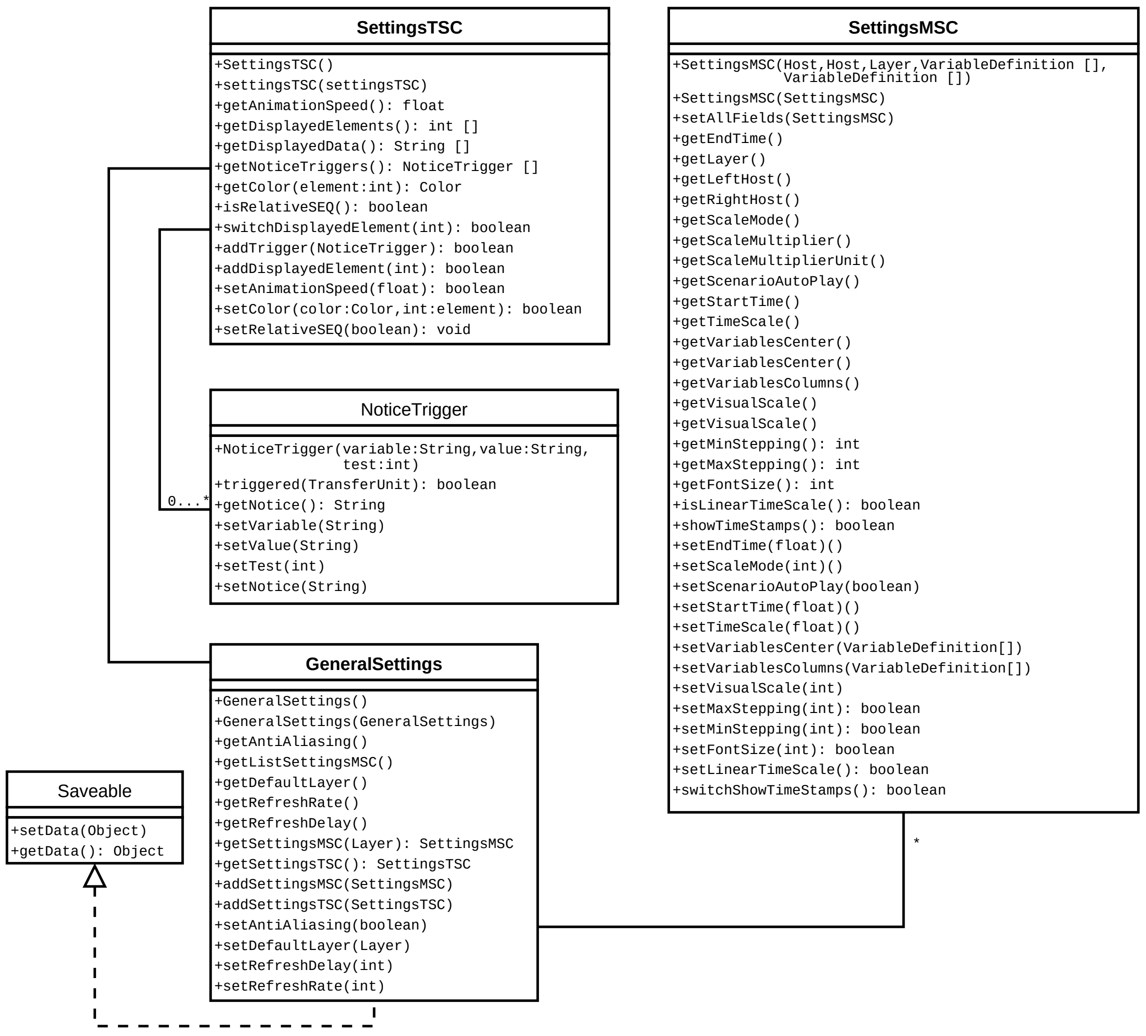

SettingsTSC
+SettingsTSC()
+settingsTSC(settingsTSC)
+getAnimationSpeed(): float
+getDisplayedElements(): int []
+getDisplayedData(): String []
+getNoticeTriggers(): NoticeTrigger []
+getColor(element:int): Color
+isRelativeSEQ(): boolean
+switchDisplayedElement(int): boolean
+addTrigger(NoticeTrigger): boolean
+addDisplayedElement(int): boolean
+setAnimationSpeed(float): boolean
+setColor(color:Color,int:element): boolean
+setRelativeSEQ(boolean): void
SettingsMSC
+SettingsMSC(Host,Host,Layer,VariableDefinition [], VariableDefinition [])
+SettingsMSC(SettingsMSC)
+setAllFields(SettingsMSC)
+getEndTime()
+getLayer()
+getLeftHost()
+getRightHost()
+getScaleMode()
+getScaleMultiplier()
+getScaleMultiplierUnit()
+getScenarioAutoPlay()
+getStartTime()
+getTimeScale()
+getVariablesCenter()
+getVariablesCenter()
+getVariablesColumns()
+getVisualScale()
+getVisualScale()
+getMinStepping(): int
+getMaxStepping(): int
+getFontSize(): int
+isLinearTimeScale(): boolean
+showTimeStamps(): boolean
+setEndTime(float)()
+setScaleMode(int)()
+setScenarioAutoPlay(boolean)
+setStartTime(float)()
+setTimeScale(float)()
+setVariablesCenter(VariableDefinition[])
+setVariablesColumns(VariableDefinition[])
+setVisualScale(int)
+setMaxStepping(int): boolean
+setMinStepping(int): boolean
+setFontSize(int): boolean
+setLinearTimeScale(): boolean
+switchShowTimeStamps(): boolean
NoticeTrigger
+NoticeTrigger(variable:String,value:String, test:int)
+triggered(TransferUnit): boolean
+getNotice(): String
+setVariable(String)
+setValue(String)
+setTest(int)
+setNotice(String)
0...*
GeneralSettings
+GeneralSettings()
+GeneralSettings(GeneralSettings)
+getAntiAliasing()
+getListSettingsMSC()
+getDefaultLayer()
+getRefreshRate()
+getRefreshDelay()
+getSettingsMSC(Layer): SettingsMSC
+getSettingsTSC(): SettingsTSC
+addSettingsMSC(SettingsMSC)
+addSettingsTSC(SettingsTSC)
+setAntiAliasing(boolean)
+setDefaultLayer(Layer)
+setRefreshDelay(int)
+setRefreshRate(int)
Saveable
+setData(Object)
+getData(): Object
*

Figure 16: Settings class diagram

In the host variables settings the user can choose whether or not the time stamp column is shown and which host variables are shown in the variables column. The user can also choose which protocol variables are drawn along the packet line in the drawing area. These settings are stored in **SettingsMSC** objects.

In scale-specific settings the user can choose between a linear and non-linear time scale. If the user chooses the linear time scale, he can adjust the following settings:

- **Scale mode**: Defines a factor of scale to draw the MSC animation. (e.g. Seconds to seconds: one second of network time is animated in one second).
- **Visual Scale**: Adjusts the number of pixels used in the MSC animation to draw one unit of network time.
- **Time Scale**: Number of seconds that it takes for the animation to show one unit of network time. Higher values mean that the animation will move slower.

If the user, on the other hand, chooses the non-linear time scale he can adjust the following settings:

- **Minimum stepping**: The user can adjust the minimum space left between any consecutive events. However, the user cannot adjust the minimum stepping to be smaller than the current font size multiplied by the number of rows in the multi-row column. When the minumum stepping is set smaller, the font size is scaled proportionally to be smaller.
- **Maximum stepping**: The user can adjust the maximum distance a line can traverse in the drawing area. This maximum stepping cannot, however, be smaller than minimum stepping. If the maximum stepping is set to minimum stepping, the user will lose all time information on the y-axis.
- **Font size**: The user can freely choose the font size to be whatever available value he wants. When the font size multiplied by the rows in the multi-row column is bigger than the minimum stepping, the minimum stepping is enlarged.
- **Progress bar speed**: User can set progress line speed on a pixels per second basis.

All the scale-specific settings listed above are stored in **SettingsMSC** objects.

In the performance settings, stored in **GeneralSettings**, the user can adjust the following elements:

- **Refresh delay:** Sets the number of milliseconds between two refreshes of the animation. Increasing its value means that the screen will be refreshed fewer times in a second. The perception of the user is that the animations are less smooth in their movement, but the performance of the animation is increased. If the values are decreased the performance is worsened but the element in the animation move more softly.

- **Anti-Alias:** Sets the smoothness of the elements drawn in the different screens. If the checkbox is selected the elements in the animations are drawn smoother, but on the other hand the performance of the animator is worsened. Deselecting it we get the inverse results, poor graphics but better performance.

In the MSC view, the settings dialog is adapted to the different animation modes, allowing changes only in the setting classes values specific to the active working mode.

- Explore mode: the user is able to modify the performance settings, scale settings for all the layers in the animation and the variables and header fields to include in the animation for every existing layer in the animation.
- Scenario mode: due to the purpose of creating an item for the scenario playlist, a few settings can be tuned. These are the scale settings, and the header fields and variables. All of these can only be modified in the active layer.

There is a a separate `SettingsMSC` object for each layer. In explore mode, the settings dialog applies the chosen settings for all of them. In th scenario mode only the settings for the `SettingsMSC` object of the current layer is modified.

6.3.1 SettingsMSC class

To fulfill the needs for the MSC settings a class **SettingsMSC** is implemented containing:

- List for the variables and header fields selected.
- Attributes to adjust the scale settings and the panels, i.e. scale mode, visual scale, time scale, max stepping, min stepping and font size.
- Variables that are needed for scenario files, i.e. start and end time from scenario playlist and whether or not we are going to start playing scenario items in play or pause mode.
- Host information, i.e. who are the left and right hosts.

The class diagram of `SettingsMSC` can be seen in Figure 16.

The progress line speed is determined using class `CalcYCoord` (see section 5.1.5). The only adjusted variable is time scale, which is calculated as follows: Let $\mathrm{s_r}$ be real (wall-clock time) seconds and $\mathrm{s_a}$ be animation time seconds. Define

$$\begin{aligned} \text{last } y\text{-coordinate in layer} &= y \text{ pixels}, \\ \text{length of animation} &= l \ \mathrm{s_a}, \\ \text{wanted progress line speed} &= v \text{ pixels}/\mathrm{s_r} \text{ and} \\ \text{time scale} &= x \ \mathrm{s_a}/\mathrm{s_r}. \end{aligned}$$

In order to have correct line speed we need to set time scale be

$$x = \frac{lv}{y}.$$

6.4 TSC Settings

The Time Sequence Chart animation type has a number of settings the user can change. They are changed using the following tabbed panels:

- **General TSC settings**, implemented as **SettingsPanelTSC**. Includes the following settings:
  - Display variables; The user can choose from a list which packet variables are shown on the packet information panel while the packet is selected.
  - Relative sequence number; The user can choose via a checkbox whether the sequence numbers on the y-axis are relative (starts from 0) or absolute (reflect the actual sequence numbers of the packets).
  - Pause to display notice; When a new notice comes visible during animation, the user can choose how long the animation is paused. The time is in seconds, entered in a text input field.
  - Animation speed; The user can control the speed of the animation with a slider. The slider values represent “events per second”.
- **Graphical Elements**, implemented as **SettingsPanelTSCElements**. The user can choose which of the different graphical elements possible in the TSC animation are shown and their colors.
  - Displayed units; The user can choose from a list which graphical elements are shown. These elements are also visible on the legend panel.
  - Color of display units; The user can choose the colors used to display different elements. This is chosen using `javax.swing.JColorChooser` in a separate dialog window.
- **Notice settings**, implemented as **SettingsPanelTSCNotices**. Allows the user to add variable-value pairs that trigger automatic notices to be displayed on the notice bar of the animation panel. The panel includes a list where the user can add, delete and modify trigger events and set their variable name, value, trigger test, and notice text.

  For storing the above settings the class **SettingsTSC** is implemented. In addition to the normal settings data it contains a list of all configured **NoticeTrigger**s, and has setter and getter methods for accessing the data. The classes can be seen in Figure 16.

7 Control Signal framework

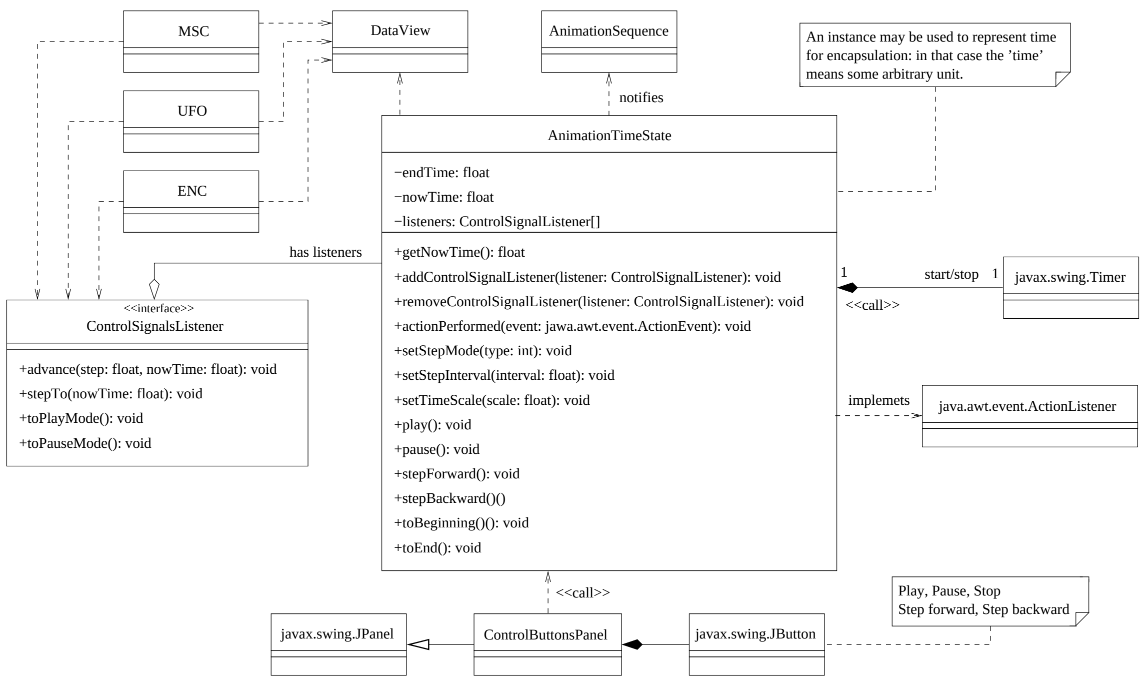

MSC
DataView
AnimationSequence
An instance may be used to represent time for encapsulation: in that case the 'time' means some arbitrary unit.
UFO
notifies
AnimationTimeState
−endTime: float
−nowTime: float
−listeners: ControlSignalListener[]
ENC
has listeners
+getNowTime(): float
+addControlSignalListener(listener: ControlSignalListener): void
+removeControlSignalListener(listener: ControlSignalListener): void
+actionPerformed(event: jawa.awt.event.ActionEvent): void
+setStepMode(type: int): void
+setStepInterval(interval: float): void
+setTimeScale(scale: float): void
+play(): void
+pause(): void
+stepForward(): void
+stepBackward()()
+toBeginning()(): void
+toEnd(): void
1
start/stop
1
javax.swing.Timer
<<call>>
<<interface>>
ControlSignalsListener
+advance(step: float, nowTime: float): void
+stepTo(nowTime: float): void
+toPlayMode(): void
+toPauseMode(): void
implemets
java.awt.event.ActionListener
<<call>>
Play, Pause, Stop
Step forward, Step backward
javax.swing.JPanel
ControlButtonsPanel
javax.swing.JButton

Figure 17: Control Signals framework

7.1 AnimationTimeState class

The central class of the Control Signals framework. The main functionalities are:

- Receive user signals as method calls (`play()`, `pause()`, etc.). The calls are made by the buttons panel component.
- Make calls to all registered `ControlSignalListeners` according to either user signals (in pause mode) or timer events (in play mode).

Methods play(), pause(), stepForward(), stepBackward(), toBeginning(), toEnd(): These methods are called by the toolbar buttons. **AnimationTimeState** reacts to these calls by making necessary state changes and necessary calls to `ControlSignalListeners`.

Methods addControlSignalListener, removeControlSignalListener: Add or removes a `ControlSignalListener`. Control signal listeners receive coordinated events for the animation, like animation 'ticks'.

Method setStepMode: Used to select the desired step mode from possibilities 'time slice', 'host A events', 'host B events', 'send events', 'receive events' and 'all events'.

Method setStepInterval: The desired step interval in physical computer clock milliseconds for the 'ticks' in play mode.

Method setTimeScale: Sets the time scale of the animation. Scale of 1.000 means presenting one second of network exchange takes ONE second of real time and scale 10.000 means presenting one second of network exchange takes TEN seconds of real time.

Other things to note:

- An `AnimationTimeState` object contains a `javax.swing.Timer` instance, which is used to get timed animation 'ticks' to run the animation in play mode. For each timer event the `AnimationTimeState` object gets a call on its own `actionPerformed` method, does the required accounting and makes subsequent calls to all listeners.
- For all other step modes than the 'time slice', the `DataView` is queried for the size of the next tick.
- When run in Scenario mode (for presenting an animation sequence, as opposed to Explore mode) the `AnimationTimeState` needs to contain an `endTime`. When the `endTime` is reached, the `AnimatioTimeState` puts itself in pause mode and calls the `showNext` method of the `AnimationSequence` object. This is a signal for the `AnimationSequence` to start presenting the next item in the animation sequence.
- `AnimationTimeState` takes into account the actual processing rate of the timer events. In times when the CPU load reaches 100%, it is not possible to process the animation ticks at the target rate. In these cases the `AnimationTimeState` adjusts the logical step size to reflect the actual processing speed. The logical step size means the step expressed as network time in the `advance(step, nowTime)` method). Because of this mechanism, the actual time spent in playing an animation from beginning to end depends ONLY on the length of the protocol event data and the time scale, NOT on computer speed (within reasonable accuracy). However, the number of animation frames drawn during the presentation of the animation MAY depend on computer speed.

7.2 ControlSignalsListener interface

All animation panels need to implement the `ControlSignalsListener` interface. The main UI needs to register all visible animation panels as `ControlSignalsListeners` for the current `AnimationTimeState` object.

Method advance(step, nowTime): tells the animation panel to make an incremental advance from previous state to `nowTime`. The difference between previously shown time state and `nowTime` is given in parameter step. If the animation panel doesn't need to make difference between advance operations and stepTo operations, this method can be implemented as just "`public void advance(float` step, float nowTime) stepTo(nowTime); ". It is the responsibility of the animation panel to call `repaint()` on itself after it has updated its state.

Method stepTo(nowTime): tells the animation panel to show the animation time given in parameter nowTime, regardless of the previously shown state.

Method toPlayMode(): tells the animation panel that a continuing sequence of advance(step, nowTime) calls will probably follow. (Some panels may wish to disable scrolling mode as result of this call.)

toPauseMode(): tells the animation panel that the following stepTo-calls will be results of direct user input. (Some panels may wish to enable scrolling mode as result of this call.)

8 Animation Sequence framework

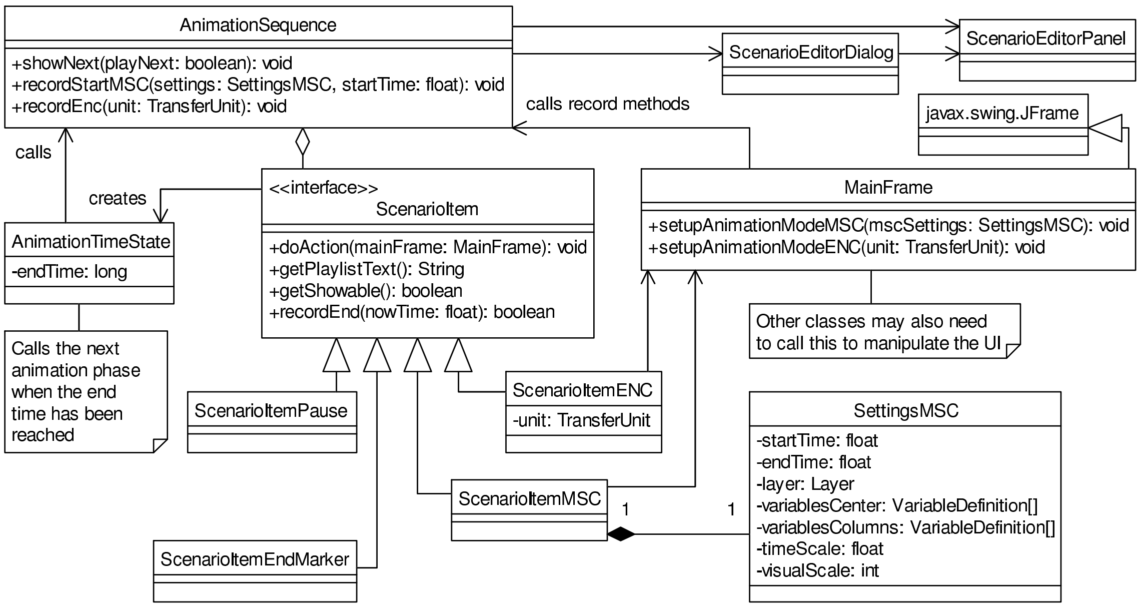

AnimationSequence
+showNext(playNext: boolean): void
+recordStartMSC(settings: SettingsMSC, startTime: float): void
+recordEnc(unit: TransferUnit): void
ScenarioEditorDialog
ScenarioEditorPanel
calls record methods
javax.swing.JFrame
calls
creates
<<interface>>
ScenarioItem
+doAction(mainFrame: MainFrame): void
+getPlaylistText(): String
+getShowable(): boolean
+recordEnd(nowTime: float): boolean
MainFrame
+setupAnimationModeMSC(mscSettings: SettingsMSC): void
+setupAnimationModeENC(unit: TransferUnit): void
AnimationTimeState
-endTime: long
Other classes may also need to call this to manipulate the UI
Calls the next animation phase when the end time has been reached
ScenarioItemPause
ScenarioItemENC
-unit: TransferUnit
SettingsMSC
-startTime: float
-endTime: float
-layer: Layer
-variablesCenter: VariableDefinition[]
-variablesColumns: VariableDefinition[]
-timeScale: float
-visualScale: int
ScenarioItemMSC
1
1
ScenarioItemEndMarker

Figure 18: Animation sequence framework

Animation Sequence framework provides a way for storing a sequence of individual presentation types, and ways for creating such sequences. The main idea is a "playlist" type of list of individual presentations. These presentations may currently be TSC or MSC animations (with or without auxiliary animation types), Encapsulation animations, or pauses.

8.1 AnimationSequence class

The **AnimationSequence** class actually contains the list of presentations. For each item in the list it:

- Creates an instance of `AnimationTimeState` according to the requirements of the item to be presented. (e.g. sets a suitable `endTime`)
- Calls the main UI to set the desired view mode. Gives the newly created `AnimationTimeState` instance as parameter to main UI, so that it can be registered as listener for control button signals.
- Receives a call from the `AnimationTimeState` instance when the `endTime` is reached, and initiates the presentation of the next item in the list.

8.2 ScenarioItem interface

An interface that all the items in the animation sequence need to implement. The interface functionality is related to providing a way to call for the item to carry out its 'action' and to have it give the necessary information about itself for e.g. presentation in the scenario play list.

8.3 ScenarioItemMSC class

Initiates showing an MSC animation. Contains a reference to `SettingsMSC` object which contains settings specific to the MSC animation panel.

Any settings in `SettingMSC` or other settings objects are created and edited through specific editing dialogs invoked through the main UI.

The semantics for the start time and end time for an MSC scenario item are as follows: when the item is selected from the scenario play list, the current `Anima-tionTimeState` instance is set to point to the **start time** of the item. When the animation is played past the **end time** of the item, the `AnimationTimeState` will call the `AnimationSequence` to select the next item in the play list, and thus end the presentation for the previous item.

8.4 ScenarioItemTSC class

Starts the playing of a TSC animation. The implementation is essentially the same as with the corresponding **ScenarioItemMSC**, described above.

8.5 ScenarioItemENC class

Instructs the `MainFrame` to show the encapsulation for the designated `Transfer-Unit`.

8.6 ScenarioItemPause class

An item in the playlist that can be used to wait for user input before continuing.

8.7 Settings objects

The settings objects for animation panels, edited through specific editing dialogs and used by their respective animation panels. Currently the only two actual settings objects are `SettingsMSC` and `SettingsTSC`.

8.8 ScenarioEditorDialog and ScenarioEditorPanel classes

The **ScenarioEditorDialog** class is related only to presenting the editing widgets inside a dialog, and how the dialog itself behaves with the `MainFrame`. The actual editing functionality is implemented as a separate panel to make it general so it can be used in the UI in other ways than in its own dialog. For all editing functionality, the panel class and the AnimationSequence class communicate directly with each other.

The **ScenarioEditorPanel** presents the scenario play list in an interactive list component. All changes from editing will be instantly visible in the list. At all times the list can be used to select the desired item from the list for playing or editing. The `ScenarioEditorPanel` also contains the necessary buttons for inserting, deleting and editing scenario items.

8.9 Recording an animation sequence

Recording an animation sequence happens as an interaction between an `AnimationSequence` object and the main UI. When the user wants to start recording a new animation sequence, he selects 'Scenario mode' command from the 'Animation' menu. At that time the `ScenarioEditorDialog` is shown, with the `ScenarioEditorPanel` as its contents. The dialog is non-modal, so that it is possible to operate all UI widgets while it is visible. The recording mode is toggled on and off by using the 'Recording mode' toggle button in the `ScenarioEditorPanel`.

When the recording mode is on, whenever the user presses play, the main UI makes a method call to the `recordStart` method of `AnimationSequence`. That causes the `AnimationSequence` to create a new `ScenarioItem` object. The exact type and settings correspond to the active mode and settings in the main UI. The created `AnimationAction` object is otherwise complete, but (in case of actions that do have a duration) it lacks the `endtime`.

Whenever the user changes animation settings, layer or view mode, the main UI makes method call `recordEnd` to the `AnimationSequence`. The next `recordEnd` after a `recordStart` call is considered the end of the item being recorded. When the `recordStop` is called, the `AnimationSequence` is able to complete the `ScenarioItem` with the `endTime` information, and insert the complete object in the animation sequence list.

A summary of responsibilities for different classes regarding recording an animation sequence:

Main UI:

- Receive user input events about starting the recording, play events, mode switches etc.
- Keep track of recording mode (on/off).
- Signal play events to `AnimationSequence (params: start time, settings)`.
- Signal mode switch to `AnimationSequence (params: end time)`.

AnimationSequence:

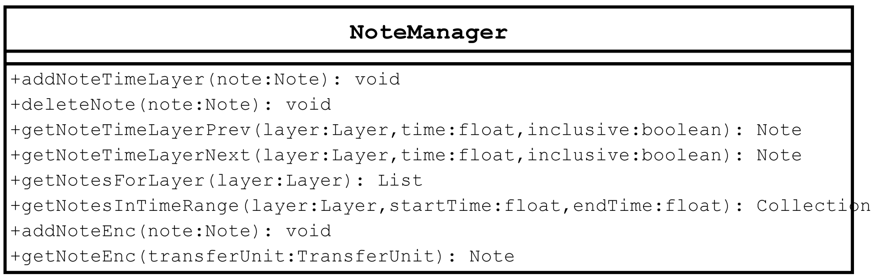
- Receive 'play' and 'mode switch' events from main UI.
- Create `ScenarioItem` objects.
- Insert `ScenarioItem` objects in the list that makes up the animation sequence.
- Update information in `ScenarioEditorPanel`.

9 Notes framework

In this section we will present a framework to work with the notes that will be shown in the different types of animation to increase the educational abilities of the software.

There is only one **Note** class, which is used for the MSC, TSC, and ENC animation types. The **NoteManager** class acts as a container for all notes in the animations, and contains separate methods for handling the notes in the ENC, TSC, and MSC animations.

The classes for both types of notes are included into the data structures due to some associations that need these classes with other ones present in the data structures to make easier the access to all this data to the Control Signals framework through the `NoteManager` (see Figure 19). Any notes added to the `NoteManager` (see Figure 19) are stored along with other animation scenario data, so they're persistent.



```
NoteManager

+addNoteTimeLayer(note:Note): void
+deleteNote(note:Note): void
+getNoteTimeLayerPrev(layer:Layer,time:float,inclusive:boolean): Note
+getNoteTimeLayerNext(layer:Layer,time:float,inclusive:boolean): Note
+getNotesForLayer(layer:Layer): List
+getNotesInTimeRange(layer:Layer,startTime:float,endTime:float): Collection
+addNoteEnc(note:Note): void
+getNoteEnc(transferUnit:TransferUnit): Note
```

Figure 19: Interface to access notes in the data structures

9.1 MSC Notes

These notes represent an action that happens in a specific layer in a specific moment in time. Then we only need to map this information into the class that will represent the MSC notes, but this class also aims to serve as note for any other Time-Layer event in an animation. An identifier will also be added into it to make easier referring to the notes to edit and modify them and locating them more efficiently.

In the `NoteManager` that is used to access to the data contained in the data structures has to contain the following methods to manage the adding, editing and deleting of the notes, and fetching them as well.

addNoteTimeLayer (note:Note):void
Add a note for a specific layer in a specific moment of time. In case that the note already exists in that layer and time the note should be edited and the text replaced with the new value.

deleteNote(note:Note)
- Deletes the given note from the manager.

getNotesTimeLayerPrev (layer:Layer, time:float, inclusive:boolean): Note
- Returns the note on the given layer that's located before the given time.

getNotesTimeLayerNext (layer:Layer, time:float, inclusive:boolean): Note
- Returns the note on the given layer that's located after the given time.

getNotesForLayer (layer:Layer): List
- Returns a list with all the notes present in the specified layer passed as parameter. This has a similar function as getUnits has with TransferUnit

getNotesInTimeRange(layer:Layer, start:float, end:float): Collection
- Returns a list with all the notes present in the specified layer passed as parameter in an interval between the start and end times.

9.2 ENC Notes

The notes for encapsulation have to be represented in a different manner than the notes for MSC due to the different purpose of the representation that will have in the animation. Thus, the encapsulation notes can have one note per encapsulation tree and this note is shown at the end or during the ENC animation.

This note for encapsulation should be in a different class as is information contained in the data structures, but represents a different concept than the information coming from the protocol events and must be separated from them.

The methods to use the Encapsulation notes that are added to the `NoteManager` are:

addNoteENC (note:Note)
- Adds the note (that's specific to a `TransferUnit`) to the manager. - If already exists a note in the specified unit the existing text will be replaced with the one provided as parameter. (edit function).

getNoteEnc(transferUnit:TransferUnit):Note
- Returns any encapsulation -specific note of the given unit.

9.3 TSC Notes

The notes used in the TSC view are similar to encapsulation notes. The notes are linked to individual transfer units using the unique id of the transfer unit. TSC notes also contain a shorter notice-text that the TSC view can display separately from the regular note in the notice bar. The notices are generated automatically as the `DataView` is scanned for instances of the `NoticeTriggers` that trigger them. These variables are chosen by the user in the TSC Settings (see Section 6.4).

Methods to use TSC notes through the `NoteManager` are:

addNoteTSC (note:Note)
- Adds the note (that's specific to a `TransferUnit`) to the manager. - If there already exists a note in the specified unit the existing text will be replaced with the one provided as parameter. (edit function).

getNoteTSC(transferUnit:TransferUnit):Note
- Returns any TSC -specific note of the given unit.

Also, a new method is added to the Note class to support getting and setting the notice-text:

setNotice (text:String)
- Sets a notice-text for the Note.

getNotice ():String
- Gets the notice-text for the Note.

10 Protocol Events File reader

The Protocol Events File (PEF) is the interface between the Analyzer and the Animator components. The Animator contains a customized reader whose purpose is to read in the protocol events data and create a corresponding presentation of the data as instances of the Animator's internal data structure classes.

Even though the format of choice for the PEF is XML (eXtensible Markup Language) at the moment, the functionality of the PEF reader is abstracted to an interface called **ProtocolEventsReader**, of which the **XMLProtocolEventsReader** is only one possible implementation: future versions may provide other implementations, should more suitable options be found

10.1 ProtocolEventsReader interface

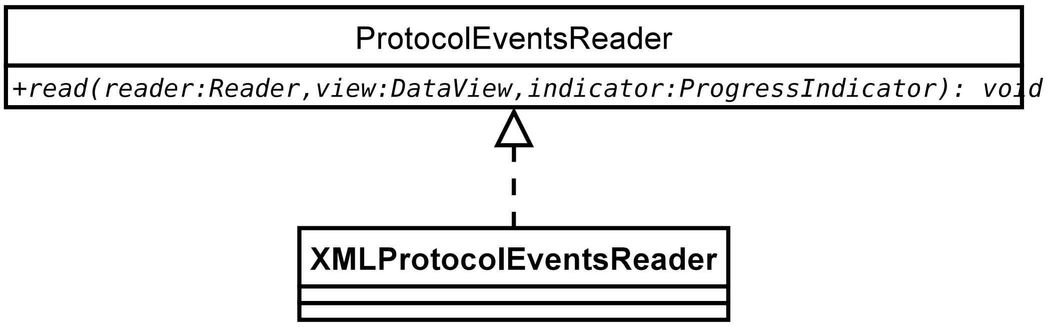

ProtocolEventsReader
+read(reader:Reader,view:DataView,indicator:ProgressIndicator): void
XMLProtocolEventsReader

Figure 20: Class diagram for the protocol events reader

The `ProtocolEventsReader` interface specifies a single method `read()` that is used to process a PEF. It accepts three parameters, a Reader that the PEF will be read from, a `DataView` object that the data in the PEF will be passed to and a `ProgressIndicator` for indicating progress to the user interface. That is, the `ProtocolEventsReader` uses a callback mechanism: whenever it encounters a piece of data in the PEF, it constructs the Java object that corresponds to the data. For example, if the reader reads in an XML element containing information on a host, it constructs the corresponding Host object and calls the `addHost(host)` -method on the `DataView` instance.

Using a callback mechanism instead of reading all the data in first and then providing getter methods for it (e.g. `getHosts()`, `getFlows()`) in the `ProtocolEventsReader` interface has many advantages: firstly, the data doesn't need to be stored in the reader itself, instead it can be passed to the `DataView` instantaneously. Secondly, having `addXXX()` -methods in the `DataView` interface makes writing unit tests easier, as the view can be easily populated with test data.

See figure 21 for an example where the reader first encounters a host, then a transfer unit and adds them to the `DataView` instance.

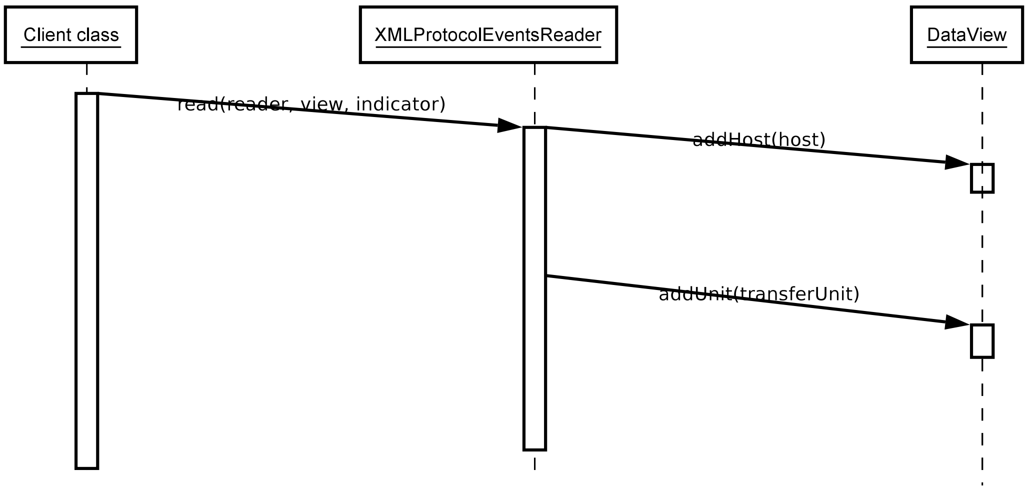


Figure 21: Sequence diagram for PEF reader

10.2 ProgressIndicator interface

As loading a large PEF file can easily take a long while, it's preferable to indicate the progress of loading to the user. A special interface, `ProgressIndicator`, is defined for this purpose. The PEF reader can signal any classes implementing this interface when any progress is made, and these classes can in turn update a progress monitor in the user interface, for example.

10.3 XML protocol events reader

10.3.1 XML support in Java 1.4

The Java 2 platform versions 1.4 and up (which is required by the Animator) provide good facilities for processing XML without the need for any external libraries: namely, the JAXP (Java Api for XML Processing). An XML parser is included in the standard libraries along with support for the two most common XML parsing APIs: DOM (Document Object Model) and SAX (Simple API for XML). The main difference between the two APIs is that DOM creates an in-memory object presentation of the XML tree while SAX simply produces events for any XML elements it encounters. Thus DOM uses more memory while SAX is more gentle on system resources: on the other hand, using the DOM API requires less effort. As the performance of reading in the protocol events data is not an issue for the Animator (it is only done once for every scenario), DOM is currently used to make the code for the reader as clear and concise as possible.

10.3.2 Implementation using the JAXP classes

The XML-based events reader can obtain a DOM parser using the class **javax.xml.parsers.DocumentBuilderFactory**. The parser can then be used to parse the PEF to an in-memory XML tree, which can then be processed to create instances of the internal data structures of the Animator.

Classes for accessing the actual DOM XML tree are located in a package called **org.w3c.dom**. For example, each element (tag) in the document is represented as a **org.w3c.dom.Element** in the object tree.

10.4 PEF file enhancements

The new TSC component requires some more information, which is currently not available from PEF files. Since this project does not include extending the Analyzer program, required information must be added into PEF files by hand.

10.4.1 SACK

The Selective Acknowledgment (SACK) information is not present in current Analyzator-generated PEF files. The Animator will assume that the sack information can be read from TCP layer `unit_sent` -event from a `sack` variable. The variable value format will be as follows: $seqno_i - seqno_j, seqno_k - seqno_l, \ldots$

For example: 100-105,340-350,360-363 (white space doesn't matter). More information on how SACK works can be found from `http://rfc.net/rfc2018.html.`

10.4.2 Transfer unit size display

The vertical axis of TSC view shows TCP sequence numbers, but also represents the amount of data sent in bytes. This does not include header size information but only payload data size. Since the `TransferUnit` objects created from TCP protocol events do not contain data size information, size must be calculated using the IP layer `TransferUnit` objects referenced by TCP transfer units.

The IP layer `unit_sent` -events of current PEF files contain a `tot_len` variable, which represents the total length of a single IP packet. Also the TCP layer events contain a `data_offset` variable, which represents the length of a TCP header in 32-bit words. To be able to calculate the TCP transfer unit data length, the PEF files should contain also an IP packet header length information. This information is currently missing from Analyzator-generated PEF files.

The TSC component will query the TPC segment's payload size from `TransferUnit` using method `getPayloadSize`. This method, however, assumes that the IP layer events contain an `ihl` variable, which is a standard IP header field representing the length of a IP header in 32-bit words. Since the current PEF files do not contain this variable, the TSC view will not be correct until `ihl` variable is added into PEF files by hand or until Analyzer program is fixed by some future DaCoPAn project.

On the other hand, MSC should view only variables from `TransferUnit`, i.e. it does not use any special methods to find out some variable-like information. This means that future Analyzers should also include the information from TCP pseudo header to PEF files in order to get this information available to MSC.

11 Scenario data

In addition to the protocol events data, each networking scenario that's visualized using the Animator can contain additional data that's used to set the scenario presentation up: this data includes animation panel settings, notes and animation sequence data. This additional information needs to be saved along with the protocol events data so that a scenario doesn't need to be set up every time it is opened using the Animator.

11.1 Scenario file and settings

All the information is contained in a single file (the **scenario file**) so that viewing a scenario doesn't require the user to download multiple files from the course homepage (for example). The scenario file is implemented as a Jar archive that allows multiple files to be stored within one file (the Java platform has built-in support for both Zip and Jar archives). See figure 22 for an example.

The settings objects which contain the settings made by the user are saved in scenario files along with other scenario data, but can also be saved in a default global settings file **SettingsFile** which is implemented as a separate jar archive using the same **ScenarioFile** class as the scenario files. Settings objects loaded from scenario files always override those loaded from the global settings file.

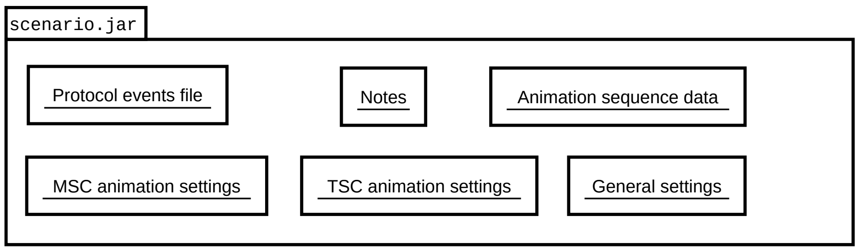


Figure 22: Scenario archive file contents

The Animator will be able to read both plain protocol events files and scenario files. When the animation sequence is altered, the user has the option of saving the protocol events (that remain unchanged) along with the settings data to a separate Scenario file.

11.2 Saving and loading the scenario data objects

Scenario data (notes, settings etc.) is only of interest to the Animator itself, i.e. it is not necessary for any other party (such as the Analyzer) to read it. Thus the format in which the data is stored is not as important as with the protocol events data. This in turn allows the Animator to take advantage of existing persistence mechanisms for objects: for example, instead of specifying a XML format for notes and writing special classes for saving and loading them, the Animator could simply use standard Java serialization facilities to save and load the data.

However, the standard Java serialization procedure is prone to class format incompability errors: if a settings class is modified, for example, older serialized instances of that class might not be readable anymore. Also, serialized objects are stored in a binary format, so they are not human-readable (let alone editable) in any easy way.

The scenario data is made persistent using a better and less error-prone alternative: an open-source Java library called **XStream** (available at `http://xstream.codehaus.org`), which is able to serialize Java objects to and from a proprietary XML format that is also human-readable.

11.3 Putting it altogether

The object persistence features required by the scenario file are specified in the interface **ObjectSerializer**. Instances of this interface should be able to save objects to the specified target and load them from the specified source. The default implementation uses XStream, as discussed in the previous chapter.

The Scenario file is accessible for the rest of the system via the class **ScenarioFile**. This class is basically a front-end to the Jar archive that contains the scenario data. It provides methods for saving and loading scenario data objects, such as notes and settings. Each of the methods for saving delegates the actual serialization of objects to ObjectSerializer, and stores the serialized object is as an entry in the Jar archive. The same applies to loading, vice versa (see figure 23). Each class wishing to save its state to the scenario file needs to implement the interface **Saveable**, which defines two methods: `getData()`, that allows the `ScenarioFile` to query the object for the data it wishes to save in the Scenario file, and `setData(Object)`, which is used to pass loaded data back to an object instance.

The loading and saving of scenario data is coordinated by `MainFrame`, which keeps track of all the `Saveable` objects in the Animator.

12 Localization

In order to make the Animator universally available as a teaching tool, its user interface was designed to support many languages, i.e. to be easily localized.

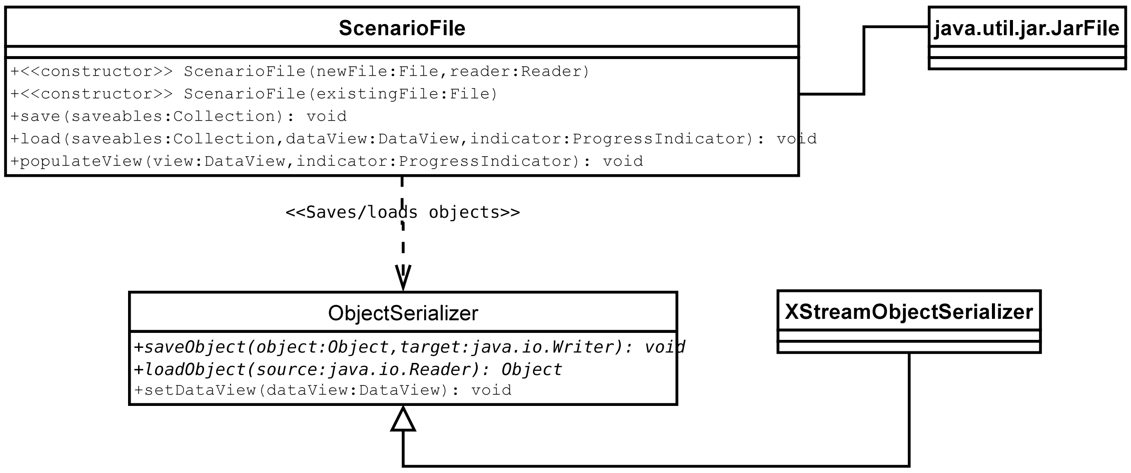

ScenarioFile
+<<constructor>> ScenarioFile(newFile:File,reader:Reader)
+<<constructor>> ScenarioFile(existingFile:File)
+save(saveables:Collection): void
+load(saveables:Collection,dataView:DataView,indicator:ProgressIndicator): void
+populateView(view:DataView,indicator:ProgressIndicator): void
java.util.jar.JarFile
<<Saves/loads objects>>
ObjectSerializer
+saveObject(object:Object,target:java.io.Writer): void
+loadObject(source:java.io.Reader): Object
+setDataView(dataView:DataView): void
XStreamObjectSerializer

Figure 23: Scenario file class diagram

12.1 The Localization class

The localization functionality of the Animator is centralized in one central class, **Localization**. At startup, the Localization class loads the property file locales.properties from the classpath. This file contains a list of all the available translations for the Animator. For each available language, the localized strings are stored in a file called **dacopanxx.properties**, where **xx** is a two-digit language code recognized by the java.util.Locale class. As English is the default language of the Animator, strings localized to English are stored in the file dacopan.properties. The standard Java class ResourceBundle takes care of resolving take localized strings from the property files.

The Localization class provides the method **getString(String key)** for retrieving the localized strings from the property files. A localized string is not restricted to static text only. It can contain placeholders for variables in the format supported by the class java.text.MessageFormat. To populate a string with variables, the method **getString(String key, Object[] params)** can be used.

For the purposes of listing the available languages in the user interface, the method **getAvailableLanguages()** can be used. To change the active language, use the method **setCurrentLanguage(Localization.Language)**. Localization.Language is a simple inner class that wraps the name of a language and the Locale it represents. The selection of a language is persistent: it is saved using the Java preferences framework (see package java.util.prefs).

References

1 DaCoPAn Software Engineering project, *Design: Animator.* Release 1.0. Universities of Helsinki and Petrozavodsk, 2004.

Appendix A. Glossary

The following are definitions, acronyms and abbreviations used in this document.

ACK packet: ACK packets are used to acknowledge the receipt of a packet in the Transmission Control Protocol (TCP). They are used by both ends of the connection to move in between states, and are the basis of TCP's reliability.

Analyzer: A module of the DaCoPAn project. It reads and analyzes packet trace files and produces a protocol events file (PEF) as its output.

Animator: A module of the DaCoPAn project. It reads protocol events files (PEF) and scenario files (SCE files), and animates the protocol exchanges using various parameters. Its main use is as a teaching tool to instruct students on protocol basics found in the protocol event file.

balloon: See **notices**.

DaCoPAn: DaCoPAn stands for visualization of Data Communication Protocol through Animation.

disappearing unit: See **dropped packet**.

dropped packet: A unit which has vanished during the course of network transmission.

drawing area: The section of the MSC and TSC animation views where the actual network data is visualized and animated.

ENC: An abbreviation for the Encapsulation view. The Encapsulation (ENC) panel is one of four animation views which the software can display.

event: The occurance of a protocol exchange. For example, an event can be the sending or receiving of a unit.

header variable: Various TCP or other network layer variables which are contained in the transfer unit header fields.

MSC: An abbreviation for the Message Sequence Chart view. The Message Sequence Chart (MSC) panel is one of four animation views which the software can display.

notices: Notices are automatically generated by the TSC view based on various events in the PEF file and displayed inside small rectangles in the notice bar.

notes: Notes are free format textual information that can be added, edited and removed freely by the user. Both the MSC and TSC animation views contain a separate notes panel.

PEF: The abbreviation stands for Protocol Events File, and is synonymous with the term PEF file and PEF files, as used throughout the document. It is an output file generated by the Analyzer and read by the Animator. It contains data about the network traffic that will be animated by the Animator.

progress line: The blue horizontal line in the MSC panel, which scrolls down the view as the animation proceeds and reveals transpiring events as they unfold.

SACK: Selective Acknowledgment. An optional TCP feature which allows the receiver to specify which segments it has received and which ones require retransmission.

scenario file: A native file written and read by the Animator. It contains all the data from a protocol events file and additional data related to how the scenario can be presented to the user, including notes and breakpoints.

SCE file(s): See **scenario file**.

sequence number: Every segment of data sent over a TCP connection has a sequence number. The sequence number is the number of the first data byte in the segment.

SEQNO: See **sequence number**.

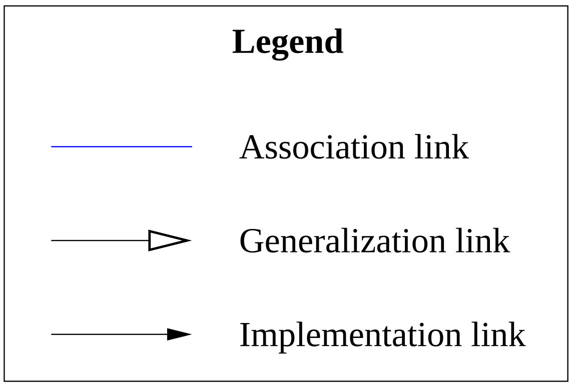
TSC: An abbreviation for the Time Sequence Chart animation view. The Time Sequence Chart (TSC) panel is one of four animation views which the software can display.

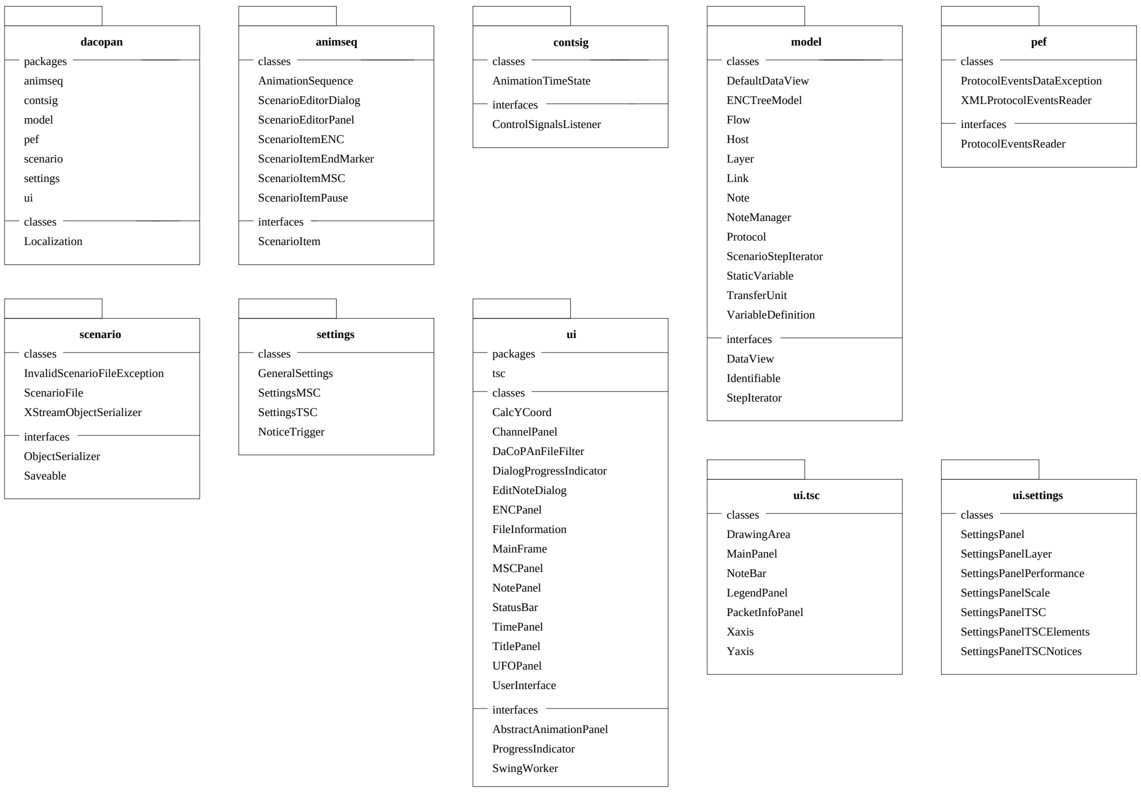
UFO: An abbreviation for the Unit Flow Orchestration view. The Unit Flow Orchestration (UFO) panel is one of four animation views which the software can display.

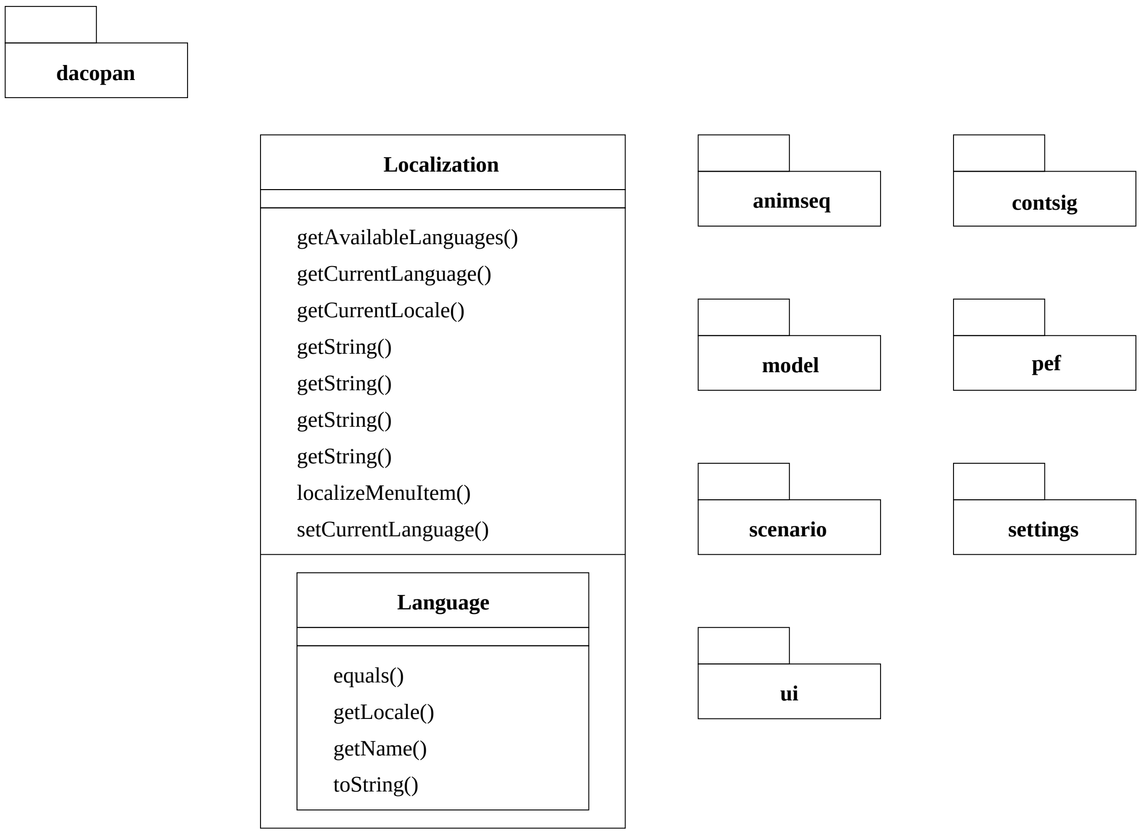
unit: A transfer unit, which refers to all logical units that are transferred regardless of network layer. Thus, a unit can be an IP packet, a TCP segment or an application layer message.

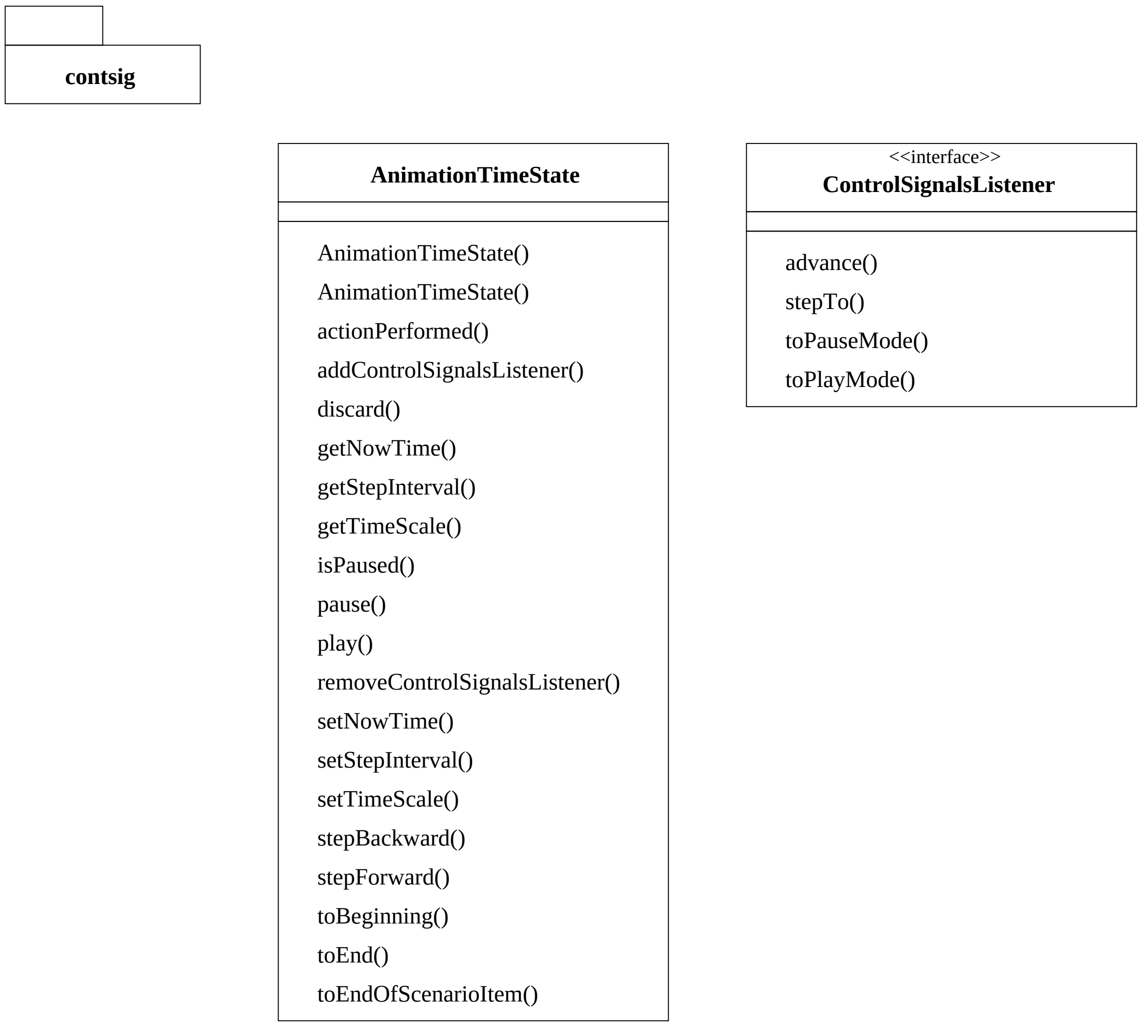
Appendix B. Packet-level class diagrams

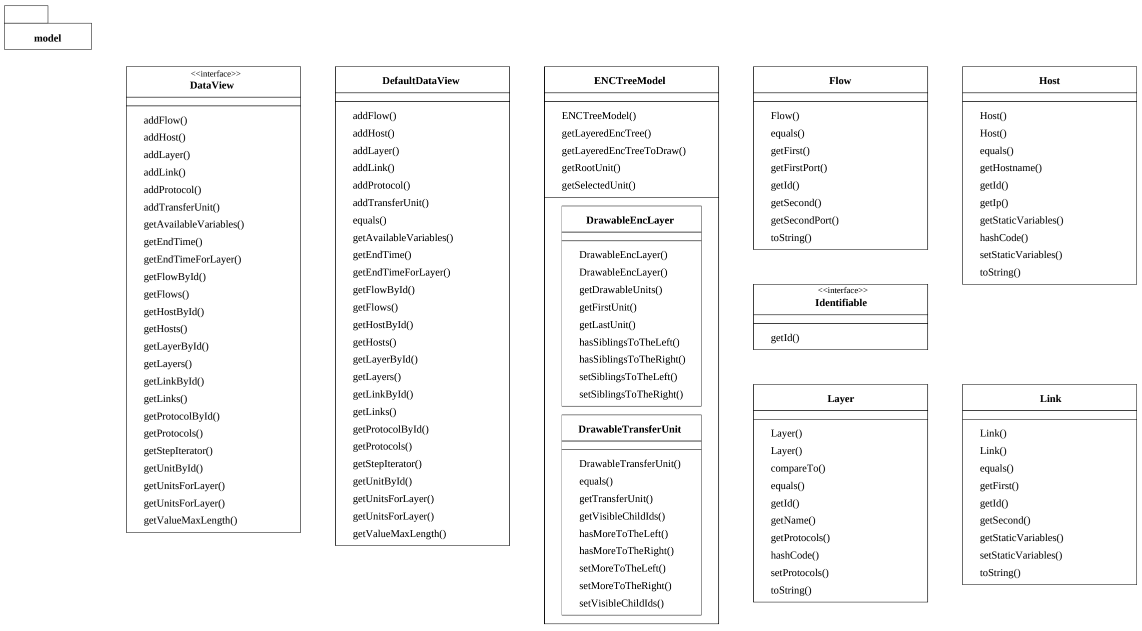
The following are packet-level class diagrams for each individual packet.

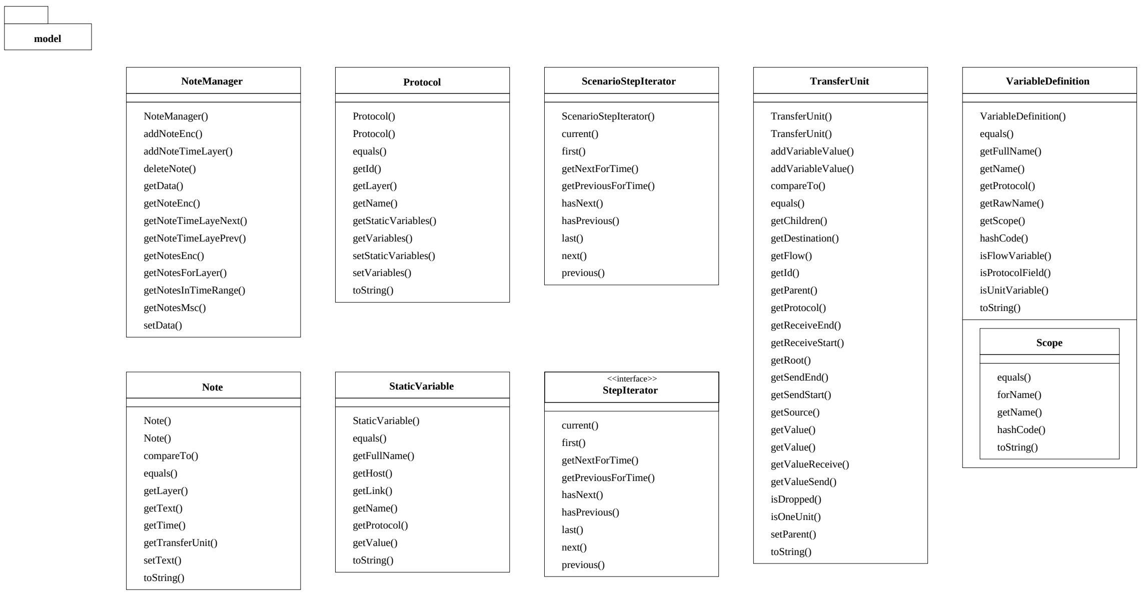


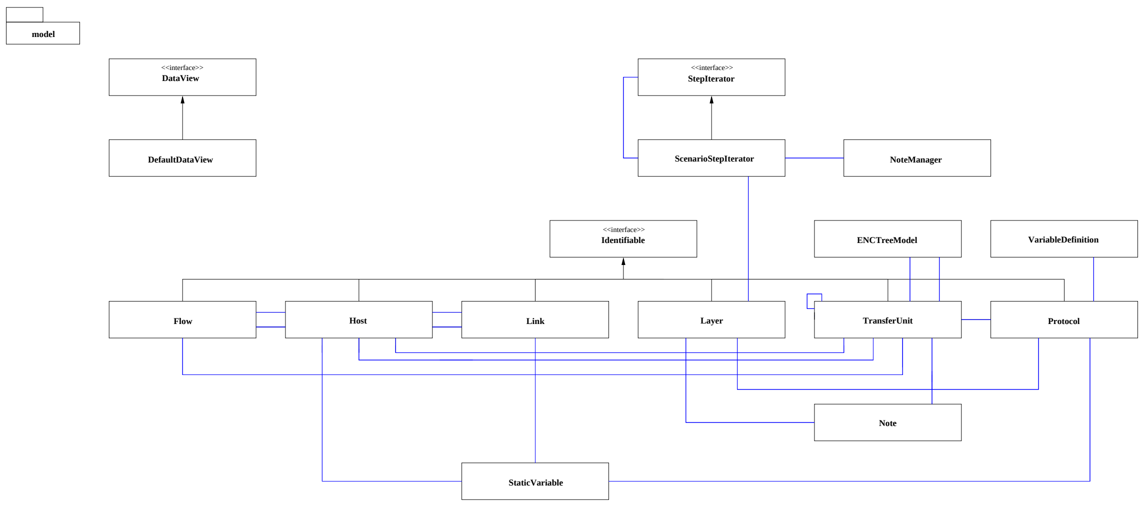


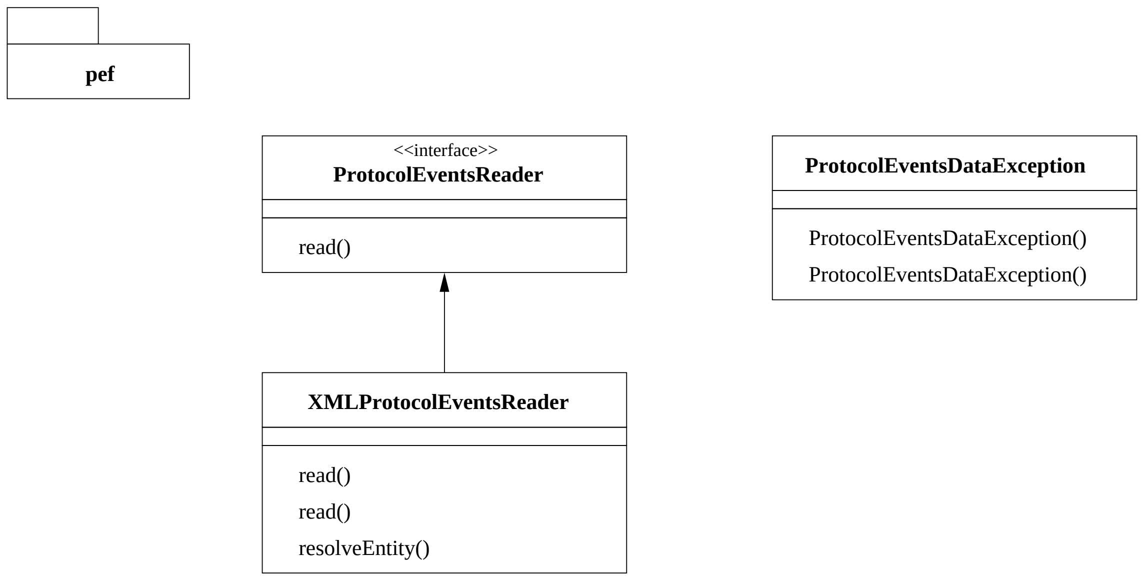

dacopan
Localization
getAvailableLanguages()
getCurrentLanguage()
getCurrentLocale()
getString()
getString()
getString()
getString()
localizeMenuItem()
setCurrentLanguage()
Language
equals()
getLocale()
getName()
toString()
animseq
contsig
model
pef
scenario
settings
ui

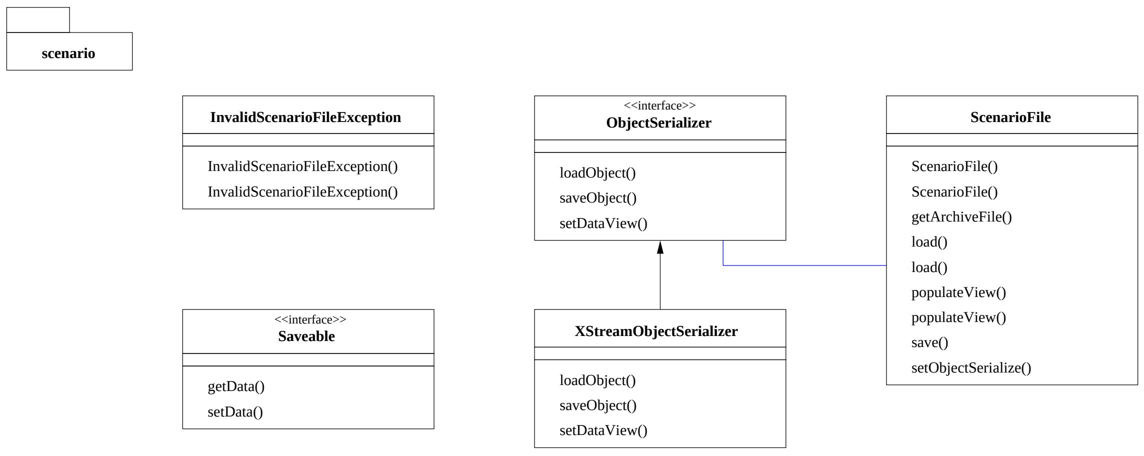

contsig
AnimationTimeState
AnimationTimeState()
AnimationTimeState()
actionPerformed()
addControlSignalsListener()
discard()
getNowTime()
getStepInterval()
getTimeScale()
isPaused()
pause()
play()
removeControlSignalsListener()
setNowTime()
setStepInterval()
setTimeScale()
stepBackward()
stepForward()
toBeginning()
toEnd()
toEndOfScenarioItem()
<<interface>>
ControlSignalsListener
advance()
stepTo()
toPauseMode()
toPlayMode()

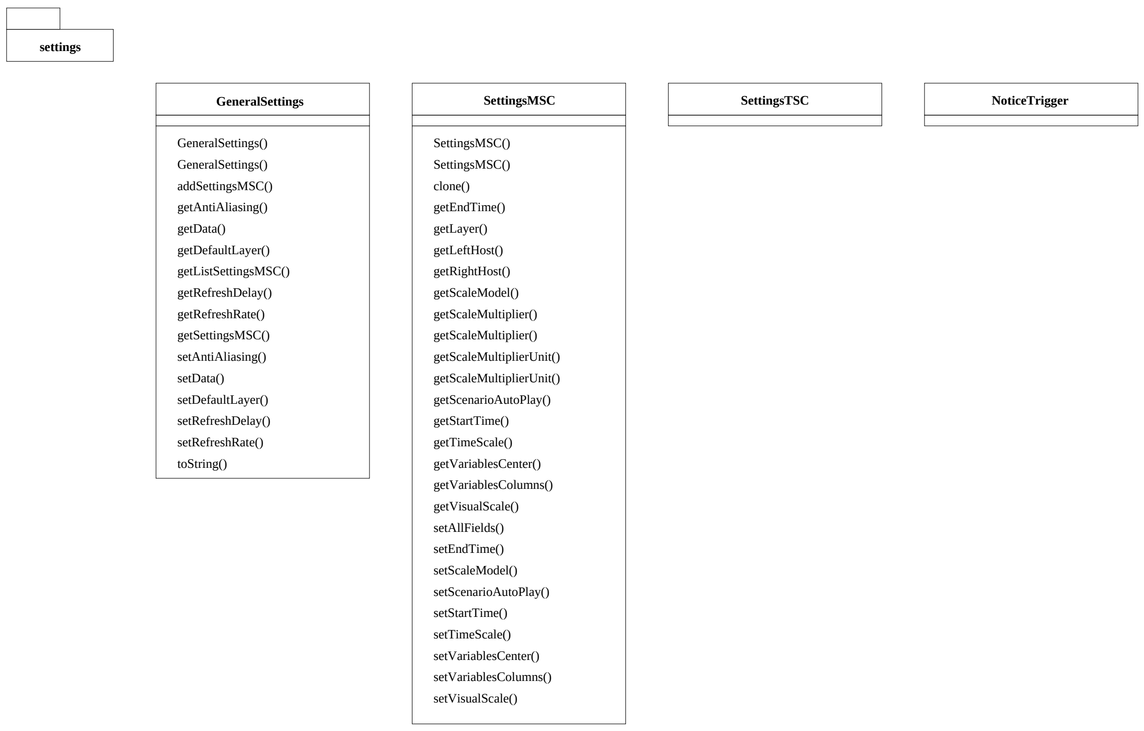

model
<<interface>>
DataView
addFlow()
addHost()
addLayer()
addLink()
addProtocol()
addTransferUnit()
getAvailableVariables()
getEndTime()
getEndTimeForLayer()
getFlowById()
getFlows()
getHostById()
getHosts()
getLayerById()
getLayers()
getLinkById()
getLinks()
getProtocolById()
getProtocols()
getStepIterator()
getUnitById()
getUnitsForLayer()
getUnitsForLayer()
getValueMaxLength()
DefaultDataView
addFlow()
addHost()
addLayer()
addLink()
addProtocol()
addTransferUnit()
equals()
getAvailableVariables()
getEndTime()
getEndTimeForLayer()
getFlowById()
getFlows()
getHostById()
getHosts()
getLayerById()
getLayers()
getLinkById()
getLinks()
getProtocolById()
getProtocols()
getStepIterator()
getUnitById()
getUnitsForLayer()
getUnitsForLayer()
getValueMaxLength()
ENCTreeModel
ENCTreeModel()
getLayeredEncTree()
getLayeredEncTreeToDraw()
getRootUnit()
getSelectedUnit()
DrawableEncLayer
DrawableEncLayer()
DrawableEncLayer()
getDrawableUnits()
getFirstUnit()
getLastUnit()
hasSiblingsToTheLeft()
hasSiblingsToTheRight()
setSiblingsToTheLeft()
setSiblingsToTheRight()
DrawableTransferUnit
DrawableTransferUnit()
equals()
getTransferUnit()
getVisibleChildIds()
hasMoreToTheLeft()
hasMoreToTheRight()
setMoreToTheLeft()
setMoreToTheRight()
setVisibleChildIds()
Flow
Flow()
equals()
getFirst()
getFirstPort()
getId()
getSecond()
getSecondPort()
toString()
<<interface>>
Identifiable
getId()
Layer
Layer()
Layer()
compareTo()
equals()
getId()
getName()
getProtocols()
hashCode()
setProtocols()
toString()
Host
Host()
Host()
equals()
getHostname()
getId()
getIp()
getStaticVariables()
hashCode()
setStaticVariables()
toString()
Link
Link()
Link()
equals()
getFirst()
getId()
getSecond()
getStaticVariables()
setStaticVariables()
toString()

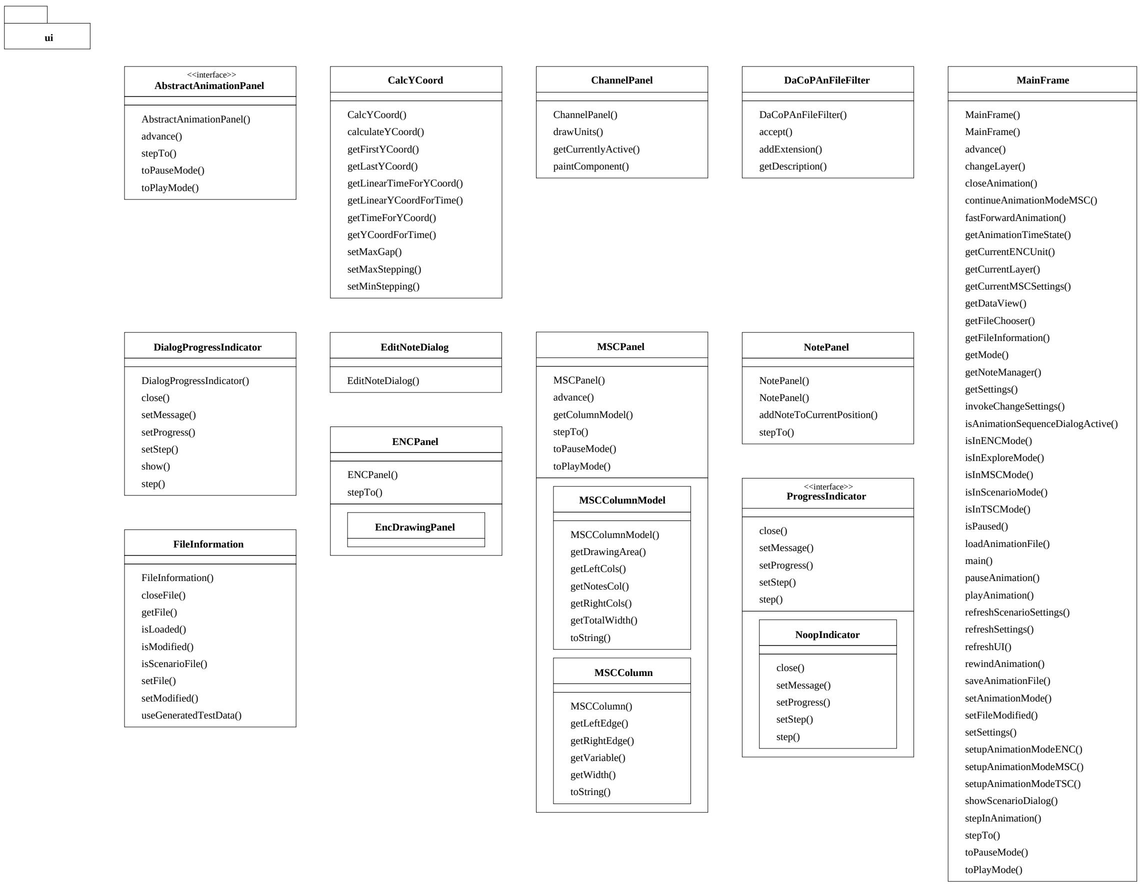

model
NoteManager
NoteManager()
addNoteEnc()
addNoteTimeLayer()
deleteNote()
getData()
getNoteEnc()
getNoteTimeLayeNext()
getNoteTimeLayePrev()
getNotesEnc()
getNotesForLayer()
getNotesInTimeRange()
getNotesMsc()
setData()
Note
Note()
Note()
compareTo()
equals()
getLayer()
getText()
getTime()
getTransferUnit()
setText()
toString()
Protocol
Protocol()
Protocol()
equals()
getId()
getLayer()
getName()
getStaticVariables()
getVariables()
setStaticVariables()
setVariables()
toString()
StaticVariable
StaticVariable()
equals()
getFullName()
getHost()
getLink()
getName()
getProtocol()
getValue()
toString()
ScenarioStepIterator
ScenarioStepIterator()
current()
first()
getNextForTime()
getPreviousForTime()
hasNext()
hasPrevious()
last()
next()
previous()
<<interface>>
StepIterator
current()
first()
getNextForTime()
getPreviousForTime()
hasNext()
hasPrevious()
last()
next()
previous()
TransferUnit
TransferUnit()
TransferUnit()
addVariableValue()
addVariableValue()
compareTo()
equals()
getChildren()
getDestination()
getFlow()
getId()
getParent()
getProtocol()
getReceiveEnd()
getReceiveStart()
getRoot()
getSendEnd()
getSendStart()
getSource()
getValue()
getValue()
getValueReceive()
getValueSend()
isDropped()
isOneUnit()
setParent()
toString()
VariableDefinition
VariableDefinition()
equals()
getFullName()
getName()
getProtocol()
getRawName()
getScope()
hashCode()
isFlowVariable()
isProtocolField()
isUnitVariable()
toString()
Scope
equals()
forName()
getName()
hashCode()
toString()

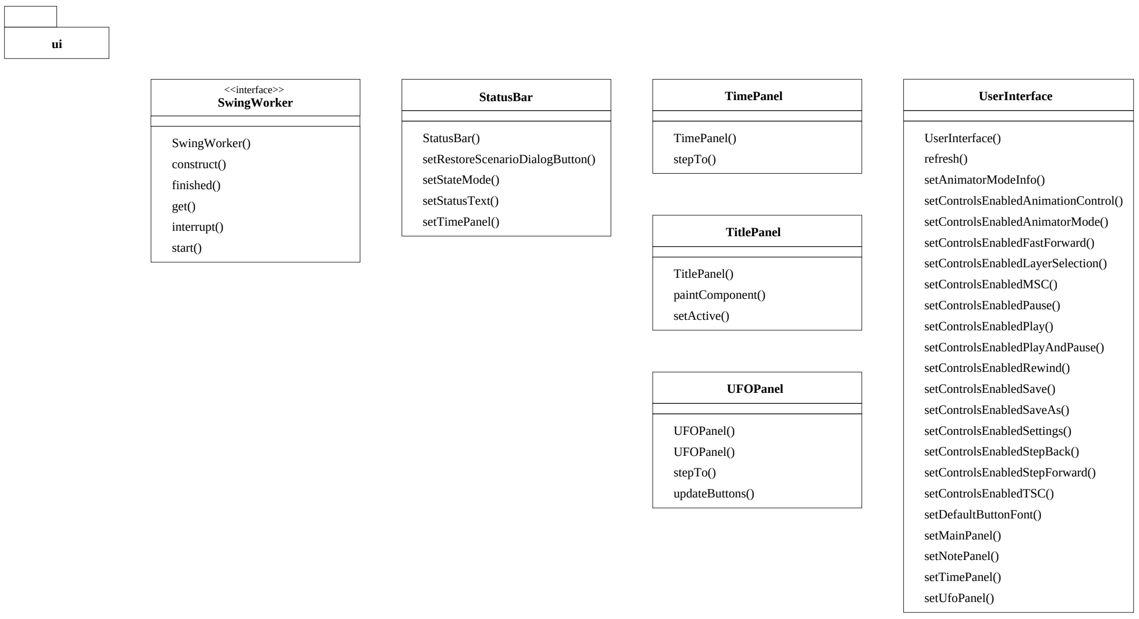

model
<<interface>>
DataView
DefaultDataView
<<interface>>
StepIterator
ScenarioStepIterator
NoteManager
<<interface>>
Identifiable
ENCTreeModel
VariableDefinition
Flow
Host
Link
Layer
TransferUnit
Protocol
Note
StaticVariable

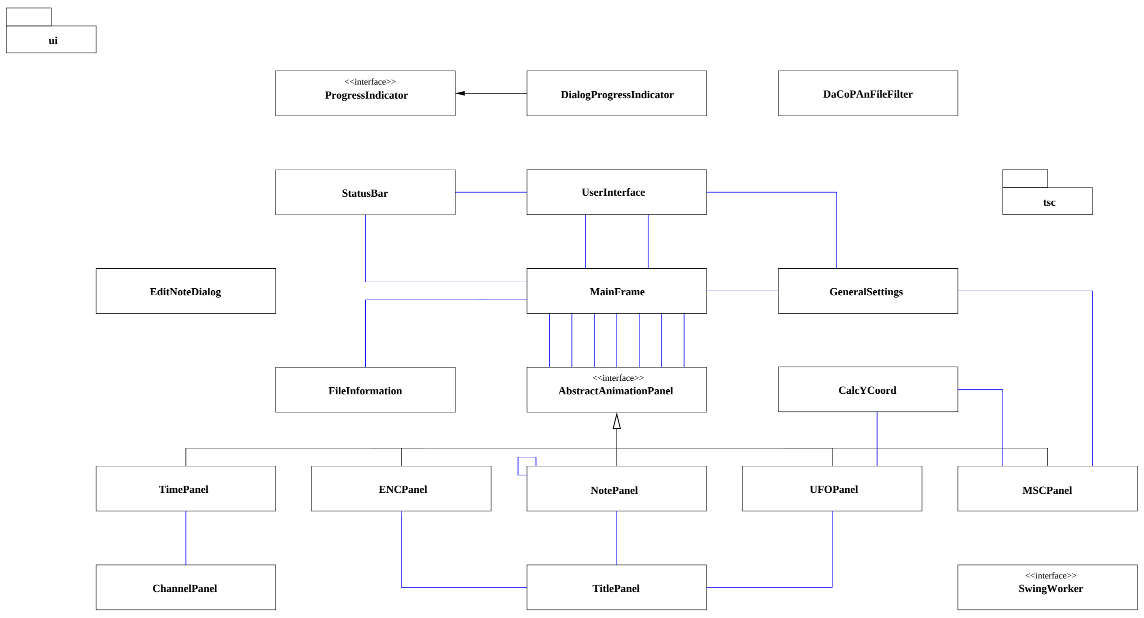

pef
<<interface>>
ProtocolEventsReader
read()
ProtocolEventsDataException
ProtocolEventsDataException()
ProtocolEventsDataException()
XMLProtocolEventsReader
read()
read()
resolveEntity()

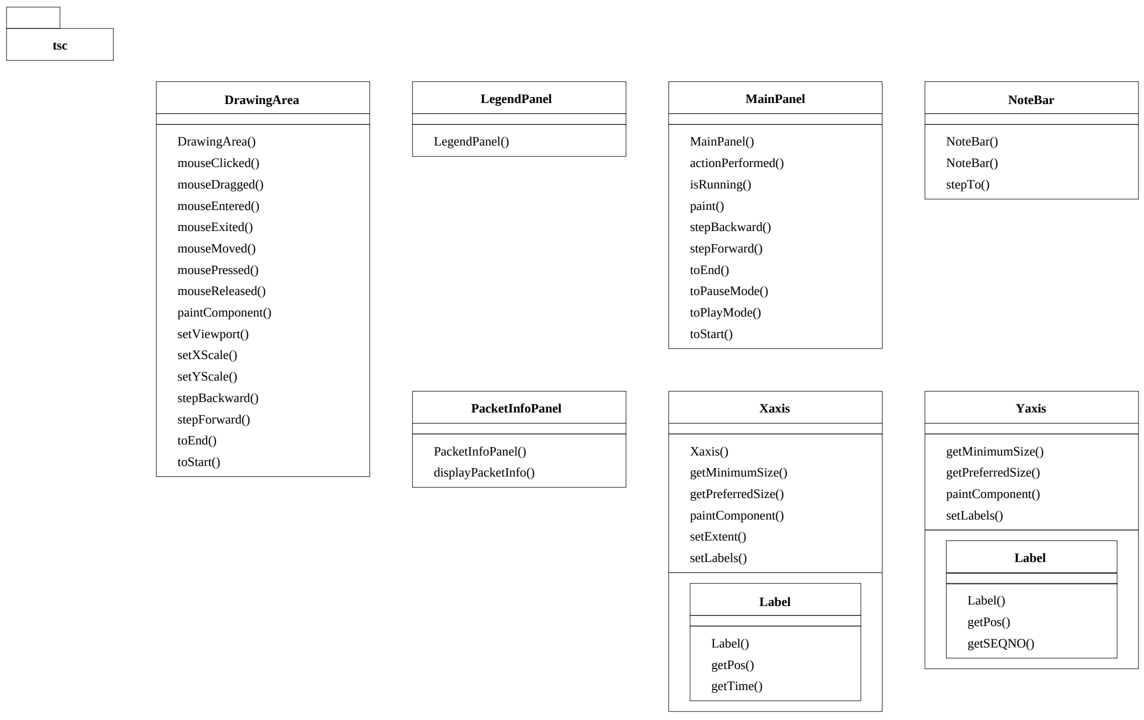

scenario
InvalidScenarioFileException
InvalidScenarioFileException()
InvalidScenarioFileException()
<<interface>>
ObjectSerializer
loadObject()
saveObject()
setDataView()
ScenarioFile
ScenarioFile()
ScenarioFile()
getArchiveFile()
load()
load()
populateView()
populateView()
save()
setObjectSerialize()
<<interface>>
Saveable
getData()
setData()
XStreamObjectSerializer
loadObject()
saveObject()
setDataView()

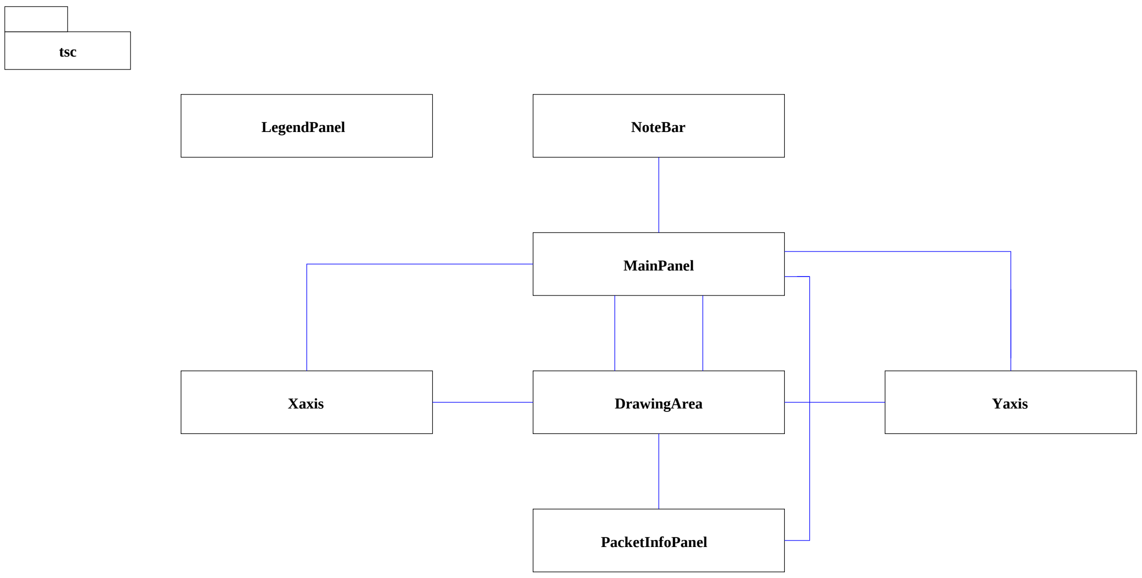

settings
GeneralSettings
GeneralSettings()
GeneralSettings()
addSettingsMSC()
getAntiAliasing()
getData()
getDefaultLayer()
getListSettingsMSC()
getRefreshDelay()
getRefreshRate()
getSettingsMSC()
setAntiAliasing()
setData()
setDefaultLayer()
setRefreshDelay()
setRefreshRate()
toString()
SettingsMSC
SettingsMSC()
SettingsMSC()
clone()
getEndTime()
getLayer()
getLeftHost()
getRightHost()
getScaleModel()
getScaleMultiplier()
getScaleMultiplier()
getScaleMultiplierUnit()
getScaleMultiplierUnit()
getScenarioAutoPlay()
getStartTime()
getTimeScale()
getVariablesCenter()
getVariablesColumns()
getVisualScale()
setAllFields()
setEndTime()
setScaleModel()
setScenarioAutoPlay()
setStartTime()
setTimeScale()
setVariablesCenter()
setVariablesColumns()
setVisualScale()
SettingsTSC
NoticeTrigger

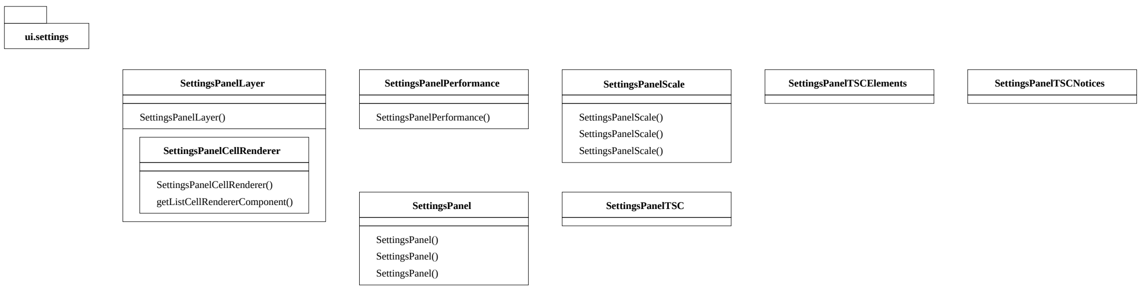

ui
<<interface>> AbstractAnimationPanel
AbstractAnimationPanel()
advance()
stepTo()
toPauseMode()
toPlayMode()
CalcYCoord
CalcYCoord()
calculateYCoord()
getFirstYCoord()
getLastYCoord()
getLinearTimeForYCoord()
getLinearYCoordForTime()
getTimeForYCoord()
getYCoordForTime()
setMaxGap()
setMaxStepping()
setMinStepping()
ChannelPanel
ChannelPanel()
drawUnits()
getCurrentlyActive()
paintComponent()
DaCoPAnFileFilter
DaCoPAnFileFilter()
accept()
addExtension()
getDescription()
MainFrame
MainFrame()
MainFrame()
advance()
changeLayer()
closeAnimation()
continueAnimationModeMSC()
fastForwardAnimation()
getAnimationTimeState()
getCurrentENCUnit()
getCurrentLayer()
getCurrentMSCSettings()
getDataView()
getFileChooser()
getFileInformation()
getMode()
getNoteManager()
getSettings()
invokeChangeSettings()
isAnimationSequenceDialogActive()
isInENCMode()
isInExploreMode()
isInMSCMode()
isInScenarioMode()
isInTSCMode()
isPaused()
loadAnimationFile()
main()
pauseAnimation()
playAnimation()
refreshScenarioSettings()
refreshSettings()
refreshUI()
rewindAnimation()
saveAnimationFile()
setAnimationMode()
setFileModified()
setSettings()
setupAnimationModeENC()
setupAnimationModeMSC()
setupAnimationModeTSC()
showScenarioDialog()
stepInAnimation()
stepTo()
toPauseMode()
toPlayMode()
DialogProgressIndicator
DialogProgressIndicator()
close()
setMessage()
setProgress()
setStep()
show()
step()
EditNoteDialog
EditNoteDialog()
MSCPanel
MSCPanel()
advance()
getColumnModel()
stepTo()
toPauseMode()
toPlayMode()
MSCColumnModel
MSCColumnModel()
getDrawingArea()
getLeftCols()
getNotesCol()
getRightCols()
getTotalWidth()
toString()
MSCColumn
MSCColumn()
getLeftEdge()
getRightEdge()
getVariable()
getWidth()
toString()
NotePanel
NotePanel()
NotePanel()
addNoteToCurrentPosition()
stepTo()
ENCPanel
ENCPanel()
stepTo()
EncDrawingPanel
<<interface>> ProgressIndicator
close()
setMessage()
setProgress()
setStep()
step()
NoopIndicator
close()
setMessage()
setProgress()
setStep()
step()
FileInformation
FileInformation()
closeFile()
getFile()
isLoaded()
isModified()
isScenarioFile()
setFile()
setModified()
useGeneratedTestData()


ui
<<interface>> SwingWorker
SwingWorker()
construct()
finished()
get()
interrupt()
start()
StatusBar
StatusBar()
setRestoreScenarioDialogButton()
setStateMode()
setStatusText()
setTimePanel()
TimePanel
TimePanel()
stepTo()
TitlePanel
TitlePanel()
paintComponent()
setActive()
UFOPanel
UFOPanel()
UFOPanel()
stepTo()
updateButtons()
UserInterface
UserInterface()
refresh()
setAnimatorModeInfo()
setControlsEnabledAnimationControl()
setControlsEnabledAnimatorMode()
setControlsEnabledFastForward()
setControlsEnabledLayerSelection()
setControlsEnabledMSC()
setControlsEnabledPause()
setControlsEnabledPlay()
setControlsEnabledPlayAndPause()
setControlsEnabledRewind()
setControlsEnabledSave()
setControlsEnabledSaveAs()
setControlsEnabledSettings()
setControlsEnabledStepBack()
setControlsEnabledStepForward()
setControlsEnabledTSC()
setDefaultButtonFont()
setMainPanel()
setNotePanel()
setTimePanel()
setUfoPanel()


ui
<<interface>>
ProgressIndicator
DialogProgressIndicator
DaCoPAnFileFilter
StatusBar
UserInterface
tsc
EditNoteDialog
MainFrame
GeneralSettings
FileInformation
<<interface>>
AbstractAnimationPanel
CalcYCoord
TimePanel
ENCPanel
NotePanel
UFOPanel
MSCPanel
ChannelPanel
TitlePanel
<<interface>>
SwingWorker


tsc
DrawingArea
DrawingArea()
mouseClicked()
mouseDragged()
mouseEntered()
mouseExited()
mouseMoved()
mousePressed()
mouseReleased()
paintComponent()
setViewport()
setXScale()
setYScale()
stepBackward()
stepForward()
toEnd()
toStart()
LegendPanel
LegendPanel()
MainPanel
MainPanel()
actionPerformed()
isRunning()
paint()
stepBackward()
stepForward()
toEnd()
toPauseMode()
toPlayMode()
toStart()
NoteBar
NoteBar()
NoteBar()
stepTo()
PacketInfoPanel
PacketInfoPanel()
displayPacketInfo()
Xaxis
Xaxis()
getMinimumSize()
getPreferredSize()
paintComponent()
setExtent()
setLabels()
Label
Label()
getPos()
getTime()
Yaxis
getMinimumSize()
getPreferredSize()
paintComponent()
setLabels()
Label
Label()
getPos()
getSEQNO()


tsc
LegendPanel
NoteBar
MainPanel
Xaxis
DrawingArea
Yaxis
PacketInfoPanel


ui.settings
SettingsPanelLayer
SettingsPanelLayer()
SettingsPanelCellRenderer
SettingsPanelCellRenderer()
getListCellRendererComponent()
SettingsPanelPerformance
SettingsPanelPerformance()
SettingsPanelScale
SettingsPanelScale()
SettingsPanelScale()
SettingsPanelScale()
SettingsPanelTSCElements
SettingsPanelTSCNotices
SettingsPanel
SettingsPanel()
SettingsPanel()
SettingsPanel()
SettingsPanelTSC