Developer Manual

Guide to Library Implementation

1 Content

1 CONTENT 1

2 TERMINOLOGY 2

3 LIBRARY IMPLEMENTATION 3

4 COMPONENT IMPLEMENTATION 4

2.1 SETTING GRID SIZE AND GRID LOCATIONS 5

2.2 STORAGE PART 6

5 COMPONENT GRAPHICAL REPRESENTATION 7

5.1 OVERRIDING WRAPPER ROTATION AND FLIPPING FUNCTION 9

5.2 EXTENDING ROTATABLEFLIPPABLEWRAPPERPAINTED 10

6 COMPONENT PROPERTIES 13

7 COMPONENT TYPES 14

7.1 WIRE SPLITTER 14

7.2 DIGITAL DEVICE 16

2 Terminology

We use following names to represent standard Java classes:

- *String* – java.lang.String
- *Exception* – java.lang.Exception
- *Point* – java.awt.Point
- *Dimension* – java.awt.Dimension
- *Rectangle* – java.awt.Rectangle
- *Component* – java.awt.Component
- *Container* – java.awt.Container
- *Frame* – java.awt.Frame
- *Graphics* – java.awt.Graphics
- *Color* – java.awt.Color
- *Image* – java.awt.Image
- *Applet* – java.applet.Applet

Also following names are used to represent ReTrO classes:

- *Wrapper* – sim.Wrapper
- *RotatableFlippableWrapperPainted* – sim.RotatableFlippableWrapperPainted
- *GuiFileLink* – sim.GuiFileLink
- *GuiEngineLink* – sim.GuiEngineLink
- *MainWindow* – sim.MainWindow
- *CentralPanel* – sim.CentralPanel
- *RunShortcut* – sim.RunShortcut
- *Grid* – sim.Grid
- *SimException* – sim.SimException
- *Junction* – sim.lib.wires.Junction
- *JunctionList* – sim.lib.wires.JunctionList
- *WireList* – sim.lib.wires.WireList
- *Wire* – sim.lib.wires.Wire
- *BufferedContainer* – sim.util.BufferedContainer
- *EnginePeer* – sim.engine.EnginePeer
- *EngineException* – sim.engine.EngineException
- *Data* – sim.engine.Data
- *NodeList* – sim.engine.NodeList
- *EnginePeerList* – sim.engine.EnginePeerList

Furthermore, following names are used to represent ReTrO interfaces:

- *EngineModule* – sim.engine.EngineModule
- *SplitterModule* – sim.lib.wires.SplitterModule

3 Library Implementation

In ReTrO each digital component is represented by a separate Java class. When ReTrO loads a new library it will first look for files with **.lib** extension. These files are simple text files that contain list of fully qualified name for components' classes.

The format of **.lib** files is as follows:

RETRO_Library|#*Name*|#*Number*|#*Class1*|#*Class2*|# ... |#*ClassN*|#

where

- "RETRO_Library" – tells ReTrO that this is a component library
- '|' and '#' – are **Wrapper.SEPARATOR** and **GuiFileLink.BLANK** static characters respectively. ReTrO uses these characters to separate string parameters in the file.
- *Name* – name of a library that will be shown to user
- *Number* – number of components in this library
- *Class1, Class2,* … , *ClassN* – fully qualified names of components' classes

Example of library file content is shown below:

RETRO_Library|#Example|#2|#sim.lib.gates.GateAND|#sim.lib.gates.GateOR|#

Note that fully qualified name of a class tells ReTrO all it needs to know about it. For above example, a class name sim.lib.gates.GateAND tells ReTrO following things:

- Component belongs to Java package sim.lib.gates
- Component class file is GateAND.class

When ReTrO loads sim.lib.gates.GateAND, it will first determine whether package sim.lib.gates is installed on user's computer. If it is, ReTrO will load required classes from that package. Otherwise it will look for them in directory where ReTrO is installed inside /sim/lib/gates/.

In summary – when you implement a new library for ReTrO you should:

1. Supply **.lib** file that tells ReTrO where to look for new components
2. Supply appropriate **.class** files to ReTrO

4 Component Implementation

All components in ReTrO extend **Wrapper** class and must implement its following methods:

- `public Image getIcon()`

  Returns an **Image** object, which is used as an icon for

  - button on Component Bar
  - component's property window

  Although there is no constraint imposed on image size, it is strongly recommended to use 28x28 pixels image to preserve ReTrO looks.

  If icon image is loaded from a file, then **getIcon()** should call a static function **GuiFileLink.getImage(String fileName)** to load it. This method return an Image object created from a file specified by *fileName* parameter.

- `public Wrapper createWrapper()`

  Creates a new instance of component to be inserted into circuit model

- `public Wrapper createWrapper(Point gridPosition)`

  Creates a new instance of component and sets its grid position to location indicated by parameter *gridPosition*.

- `public String getBubbleHelp()`

  Returns a **String** object, which is used as a text for

  - help bubble message for button on Component Bar
  - title in component's property window

In addition each component must implement a constructor with no input parameters. This constructor is used by ReTrO to set up initial component's instance to access above methods.

Components should avoid directly using IO functions so that they can be used in both stand-alone and applet versions of ReTrO. However, if this is necessary then components can determine whether ReTrO is running as an applet using static **Applet** parameter **MainWindow.MASTER**. This field contains an applet that is running ReTrO. If it is equals to *null* then ReTrO is running as a stand-alone application.

2.1 Setting grid size and grid locations

It should be noted that **Wrapper** class is a special type of **Component** class. ReTrO uses its own grid coordinates to position and size components. These must be explicitly converted to pixel coordinates so that **Wrapper** instances can be displayed on the screen.

Wrapper instances keep track of their grid location and size using two parameters:

1. *gridLocation* – **Point** type parameter that holds component's grid position

   This variable can be accessed using **Wrapper** function **getGridLocation()**

2. *gridSize* – **Dimension** type parameter that holds component's grid size

   This variable can be accessed using **Wrapper** function **getGridSize()**

Wrapper has following functions to perform task of coordinate conversion:

- `public void setGridLocation(int x, int y)`

  Updates *gridLocation* parameter and sets actual **Wrapper** location using **Component** method **setLocation(int x, int y)**.

- `public void setGridSize(int gridWidth, int gridHeight)`

  Updates *gridSize* parameter and sets actual **Wrapper** size using **Component** method **setSize(int width, int height)**.

By default all components in ReTrO fully occupy their grid cells. However, it is possible to implement components that partially occupy their grid cells by overriding above functions. This might require the knowledge of grid cell's size in pixels, which can be obtained from current active **Grid** using a following statement

CentralPanel.ACTIVE_GRID.getCurrentGridGap()

Component grid size may vary depending on its parameters. However, all components should have default parameter settings and therefore all components should have a default size. ReTrO initializes components to their default size through **Wrapper** function

- `protected void initializeGridSize()`

  Sets component size to default value. All components must implement this function.

2.2 Storage part

Wrapper automatically handles storage and retrieval of components grid location and grid size. Components must implement following functions to store any additional parameters with circuit model:

- public int getNumberOfSpecificParameters()

  Returns number of parameters to be stored or loaded

- public String getSpecificParameters()

  Returns a string representation of component's parameters with following format

  Param1|Param2| ... |ParamN|

  where

  - *Param1, Param2, … , ParamN* are string representation of parameters
  - '|' is **Wrapper.SEPARATOR** static character. ReTrO uses this character to separate string parameters in the file representation.

- public void loadWrapper(String[] specificParameters) throws SimException

  Initializes component's parameters from array of strings, which correspond to *Param1, Param2, … , ParamN*. This method throws a **SimException** when these parameters have a wrong format.

If component does not have any additional parameters to be stored, above function still have to return some parameters to ReTrO. For example:

- *getNumberOfSpecificParameters()* might return 1
- *getSpecificParameters()* might return "null"

Note that in this case, when component is loaded via *loadWrapper(String[] specificParameters)* its input variable will be ["null"].

5 Component Graphical Representation

Wrapper class extends **Container** class. This means that Wrapper graphical representation can contain **Component** object (buttons, choices, etc) that can be used to interact with the user.

By default components that extend **Wrapper** can not be rotated and flipped. These components must implement following graphic related functions:

- public void scale()

  This function notifies component that user has scaled the circuit diagram. All necessary adjustments required to correctly display component on screen are made at this point.

- public void changeColor(Color c)

  Graphical representation of components should have some part that can change color to indicate that component has been selected by user. This function tells component what color those parts should be.

- public void paint(Graphics g)

  Paints graphical representation of component to screen via **Graphic** object. The format of this function must be as follow:

  ```
  public void paint(Graphics g)
  {
      if( this.isVisible() )
      {
          // Paint procedures
      }
  }
  ```

  It is not recommended to draw **Image** object in this function as it considerably slows down ReTrO performance. Instead, this function should use drawing functions provided by **Graphic** object.

Above functions might require the knowledge of grid cell's size in pixels, which can be obtained from current active **Grid** using a following statement

CentralPanel.ACTIVE_GRID.getCurrentGridGap()

Associated with each component is a set of input and output pins. In ReTrO pins are implemented as **Junction** object. Each component must implement following functions to handle its pins:

- public boolean canDrop()

  Determines whether component can be placed on the grid using static function

  boolean Wrapper.canDropJuncion(int gridX, int gridY, int nodes)

  where *gridX* – x coordinates of the pin in grid reference plane
  gridY – y coordinates of the pin in grid reference plane
  nodes – bus size of the pin

  This function returns *true* if component can be placed on the grid. Otherwise it returns *false*.

- public void droped()

  Places component's pins on the grid using static function

  Junction Wrapper. setPinAt(int gridX, int gridY, int nodes)

  where *gridX* – x coordinates of the pin in grid reference plane
  gridY – y coordinates of the pin in grid reference plane
  nodes – bus size of the pin

  Above function returns **Junction** object allocated to component's pin. It is the responsibility of each component to keep track of its **Junction** objects.

- public void selected()

  Notifies all component's pins that this component is no longer connected to them using **Junction** function **removePin**().

  This function also changes component's appearance to indicate that it has been selected. It is recommended to use **Wrapper** function **changeColor(Color.green)** to preserve ReTrO looks.

- public void checkAfterSelected()

  Checks whether Junction objects associated with component's pin should be removed from the grid after component have been selected. This is done by calling static function

  void Wrapper.checkPin(Junction pin)

Wrapper has two variables that handle rotation and flipping

1. *angle* – an integer that indicates the rotation angle
2. *isFlipped* – a boolean that indicates whether component is flipped or not

Rotation and flipping can be implemented in two ways:

1. Overriding **Wrapper** rotation and flipping function
2. Extending **RotatableFlippableWrapper** class

5.1 Overriding Wrapper Rotation and Flipping Function

In this method components that wish to be rotated and flipped must override following functions:

- `public boolean canFlip()`

  Indicates whether this component can be flipped

- `public void setFlipped(boolean newFlip)`

  Sets component's *isFlipped* to new value

- `public void flipHorizontal()`

  Flips component about vertical axis. This involves adjusting component's *angle* and *isFlipped* parameters

- `public void flipVertical()`

  Flips component about horizontal axis. This involves adjusting component's *angle* and *isFlipped* parameters

- `public boolean canRotate()`

  Indicates whether this component can be rotated

- `public void setAngle(int newAngle)`

  Sets component's *angle* to new value

- `public void rotateLeft()`

  Rotates component anti-clockwise. This involves adjusting component's *angle*.

- public void rotateRight()

  Rotates component clockwise. This involves adjusting component's *angle*

- public void restoreAngleFlipped(int oldAngle, boolean oldFlip)

  Restores parameters *angle* and *isFlipped* to values prior to component selection. It is responsibility of each component to keep those values.

When component's parameters *angle* and *isFlipped* are changed, component is responsible to repaint itself on the grid. **Grid** class in ReTrO extends **BufferedContainer** object to speed up its graphical performance. **BufferedContainer** provide three functions to handle its screen drawings:

1. **eraseComponent(Component comp, boolean update)**

   Erases *comp* image from the buffer. This change is redrawn to the screen if *update* is true.

2. **paintComponent(Component comp, boolean update)**

   Paints *comp* image to the buffer. This change is redrawn to the screen if *update* is true.

In addition **Grid** provide a following function to redraw buffer content on the screen:

- **blitWorkplaceToScreen(Rectangle clip)**

  Redraws the content of buffer bounded by *clip* to screen. Active grid can be accessed by ReTrO's objects through static parameter **CentralPanel.ACTIVE_GRID**

It should be noted that this method of implementing rotatable and flippable components requires components to explicitly store any information relevant to rotation and flipping. This also includes *angle* and *isFlipped* parameters.

5.2 Extending RotatableFlippableWrapperPainted

In this method components that wish to be rotated and flipped extend **RotatableFlippableWrapperPainted** class. This class implements rotation and flipping as shown on Figure 1. There are totally eight possible positions that each component can take. These positions can be completely described using *angle* and *isFlipped* parameters.

It should be noted that **RotatableFlippableWrapper** class handles storage of *angle* and *isFlipped* parameters automatically.

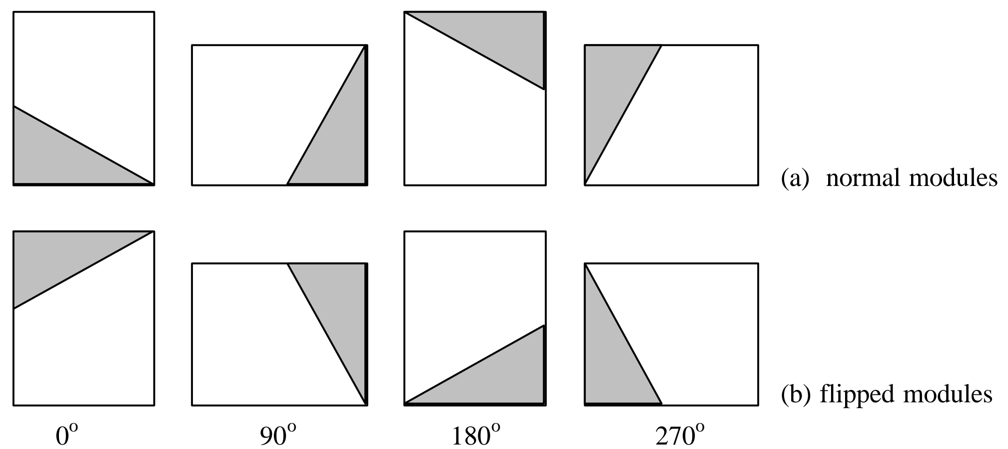

Figure 1: rotation and flipping of modules.

Unlike normal Wrapper components **RotatableFlippableWrapper** modules do not have to implement following functions unless totally necessary:

- *public void scale()*
- *public void changeColor(Color c)*

Instead of implementing function *paint(Graphics g)*, **RotatableFlippableWrapper** modules must implement following paint functions corresponding to position shown on Figure 1:

- protected void paintNormal_0(Graphics g)
- protected void paintNormal_90(Graphics g)
- protected void paintNormal_180(Graphics g)
- protected void paintNormal_270(Graphics g)
- protected void paintFlipped_0(Graphics g)
- protected void paintFlipped_90(Graphics g)
- protected void paintFlipped_180(Graphics g)
- protected void paintFlipped_270(Graphics g)

Unlike normal implementation of *paint(Graphics g)* above methods are not required to check whether component is visible or not before proceeding to paint functions.

Instead of implementing function *canDrop()*, **RotatableFlippableWrapper** modules must implement following similar functions corresponding to position shown on Figure 1:

- protected boolean canDropNormal_0()
- protected boolean canDropNormal_90()
- protected boolean canDropNormal_180()
- protected boolean canDropNormal_270()
- protected boolean canDropFlipped_0()
- protected boolean canDropFlipped_90()
- protected boolean canDropFlipped_180()
- protected boolean canDropFlipped_270()

Instead of implementing function d*roped()*, **RotatableFlippableWrapper** modules must implement following similar functions corresponding to position shown on Figure 1:

- protected void dropedNormal_0()
- protected void dropedNormal_90()
- protected void dropedNormal_180()
- protected void dropedNormal_270()
- protected void dropedFlipped_0()
- protected void dropedFlipped_90()
- protected void dropedFlipped_180()
- protected void dropedFlipped_270()

In addition **RotatableFlippableWrapper** modules must implement following function:

- protected void adjustToChanges()

  Adjust component's size, position and any other parameters after component rotation or flipping. Note that this function is not required to update its changes on the screen – ReTrO does this automatically.

6 Component Properties

Each component can have a property dialog for users to change its parameters. To implement this, components must override following **Wrapper** functions:

- `public boolean hasProperties()`

  Returns true if component have a property dialog. Returns false otherwise.

- `public Component getPropertyWindow()`

  Returns a **Component** object that contains necessary choices and buttons to be displayed on property dialog

- `public void respondToChanges(Component property)`

  Adjusts component's parameters based on settings of dialog and buttons of **Component** object used in property dialog

Note that **Component** object used in property dialog <u>must</u> be implemented as a separate class with appropriate function to retrieve the state of its buttons and choices. This class <u>must</u> override following **Component** function so that it can be displayed correctly:

> **public Dimension getPreferredSize()** – returns size of panel in pixels that is necessary in order to display its continent correctly

Additionally each component can have a set of commands that user can issue to it. These commands are accessed by users through a pop-up menu, which appears when user clicks the right mouse button on component. To implement this, components must override following **Wrapper** functions:

- `public int getNumberOfMenuItems()`

  Returns number of commands implemented by component

- `public String getMenuItemName(int index)`

  Each command is numbered from 0 to N-1, where N is a total number of commands. This function returns the name of command with number indicated by *index* parameter.

- `public void respondToMenuItem(String itemName)`

  Respond to command with name indicated by *itemName* parameter

Implementation of component's property and commands requires overriding of following Wrapper function:

- `public void restoreOriginalProperties()`

  Restores component's parameters to values prior to component selection. It is responsibility of each component to keep track of those values.

7 Component Types

There are two types of components in ReTrO:

1. *Wire splitters* – combine/split wires and buses
2. *Digital devices* – produce digital outputs form digital inputs

7.1 Wire Splitter

Although ReTrO allows users to create circuits with buses, at simulation level it only uses single-bit nodes. During simulation buses are considered as a collection of single-bit nodes.

To avoid confusion ReTrO does not allow wires that carry different number of bits to be directly connected to each other. Connection of these wires is handled by wire splitters. Like all components wire splitters have pins implemented as **Junction** objects. Wire splitters simply interconnect single-bit nodes of its **Junction** objects with each other.

Single-bit nodes in ReTrO are implemented as **Node** objects. Before each simulation ReTrO first groups all wires directly connected to each other. ReTrO then allocates appropriate number of **Node** objects to each of those groups. The result is a set of all single-bit nodes present in the circuit.

Associated with each **Node** object is a list of **Wire** and **Junction** objects that form corresponding node. Similarly associated with each **Junction** is a set of **Node** objects allocated to it.

When ReTrO initially creates nodes from interconnected wires, it ignores the connection pattern of wire splitters. As a result there might be several **Node** objects created for the same single-bit node. The function of wire splitters is to remove this redundancy. This process consist of three steps:

1. Remove redundant **Node** object from a complete set of all nodes
2. Allocate appropriate **Node** object to junctions that forms redundant **Node** object
3. Add wires and junctions that form redundant **Node** object to appropriate **Node** object

All wire splitters must implement **SplitterModule** interface with following function

- public void mergeNodes(NodeList nodeList)
  Removes node duplication from a complete set of nodes *nodeList*

Implementation of above function usually uses following functions:

- **Node** functions
  - *WireList getWires()*
    returns list of wires that forms this node
  - *JunctionList getJunctions()*
    returns list of junctions that forms this node
  - *void addWire(Wire w)*
    add a wire to list of wires that forms this node
  - *void addJunction(Junction j)*
    add a junction to list of junctions that forms this node
- **Junction** functions
  - *NodeList getNodes()*
    returns list of nodes allocated to this junction
- **NodeList** functions
  - *void removeItem(Node n)*
    remove node from the list
  - *Node getItemAt(int index)*
    returns node located at certain position in the list
  - *void changeItem(int index, Node n)*
    replaces the content of the list at certain position with a new value
  - *int indexOf(Node j)*
    returns position of a node in the list
- **WireList** functions
  - *Wire getItemAt(int index)*
    returns wire located at certain position in the list
  - *int getSize()*
    returns number of wires in the list

- **JunctionList** functions
  - *Junction getItemAt(int index)*
    returns junction located at certain position in the list
  - *int getSize()*
    returns number of junctions in the list

7.2 Digital Device

ReTrO uses **Wrapper** objects to model devices and wires on schematic diagrams. During simulation wires and junctions are converted to **Node** objects. Similarly digital devices are converted to **EnginePeer** objects.

Data in ReTrO is modeled by **Data** object with following functions:

- *boolean getValue()*
  returns the logical value of data
- *boolean isUndefined()*
  returns true if data is invalid and false otherwise

The exchange of data between **EnginePeer** objects is modeled by Signal object, which has following constructor:

Signal(boolean value, double t, boolean isUndefined, EnginePeer source, int pin)

where	*value*	– logical value of data
	t	– time when signal was generated
	isUndefined	– indicates whether data on signal is undefined
	source	– component that generated the signal
	pin	– output pin of source component where signal is generated

Signal that indicates the start of node floating has following constructor:

Signal(double t, EnginePeer source, int pin)

where	*t*	– time when signal was generated
	source	– component that generated the signal
	pin	– output pin of source component where signal is generated

Wrapper objects that model digital devices must implement **EngineModule** interface. This interface handles simulation part of digital devices and has following functions:

- public void createEnginePeer(EnginePeerList epl)

  Creates **EnginePeer** objects to represent digital devices. These must be inserted to list of all **EnginePeer** objects in simulation *epl* using its function *insertItem(EnginePeer p)*. Note that each device can have more than one **EnginePeer** object.

  EnginePeer class has following constructor

  EnginePeer(int inputs, int outputs, EngineModule parent)

  where *inputs* – number of input pins

  outputs – number of output pins

  parent – **EngineModule** object that created this **EnginePeer** object

  Once **EnginePeer** object is created, **Node** objects corresponding to its pins must be assigned. This can be done using following **EnginePeer** functions:

  - *setInputPin(int inPin, Node n)*

    assigns node *n* to input pin at position *inPin*

  - *setOutputPin(int outPin, Node n)*

    assigns node *n* to output pin at position *outPin*

  It is responsibility of each component to keep tracks of **Junction** object that correspond to its pin. The list of **Node** objects allocated to **Junction** object can be extracted using its function:

  NodeList getNodes()

  Above function returns a **NodeList** object that contains allocated **Node** objects, which are sorted by their significant position. In other words, the node at the beginning of the list is the least significant and the node at the end of the list is the most significant. These can be extracted using **NodeList** function:

  Node getItemAt(int significantBit)

- public void evaluateOutput(double t, Data[] in, EnginePeer p) throws EngineException

  Evaluate the state of **EnginePeer** object *p* at simulation time *t*. Data at input pins of *p* at simulation time *t* is contained in array *in*.

  Normally this function is called when the data on component's input pins has been altered. However, components can scheduled a wake up time that forces ReTrO to call this function again. This can be done using **EnginePeer** function:

  setWakeUp(double time)

  EnginePeer object can schedule three types of signal transactions on their output pins[1]:

  1. Normal transaction

     - *normalTransaction(int index, Signal s)*
       schedules signal *s* at output pin *index*

  2. Clear-all transactions

     - *setOutputPinValue(int index, boolean v, double t)*

       schedules valid signal with value *v* at time *t* on output pin *index*

     - *setOutputPinUndefined(int index, double t)*

       invalidates data on output pin *index* at time *t*

     - *floatOutputPin(int index, double t)*

       floats pin *index* at time *t*

  3. Clear-uncompleted transactions

     - *clearUncomplitedTransaction(int index, Signal newData)*

       schedules signal *s* at output pin *index*

- public void reset()

  This function notifies component that simulation has been stopped. Any required clean up after simulation are performed at this point.

[1] See Section 4.4 of "Hardware Simulation Kit in Java" thesis for more details.

- public Wrapper getParentWrapper()

  Returns **Wrapper** object that is managing this **EngineModule** interface

Note that components can schedule signal transactions and wake up times at any time during simulation. For example, component can do this when user clicks on it. Note that in this case it is responsibility of component to keep track of **EnginePeer** object that it has allocated for simulation. For these purposes the simulation time can be obtained from **GuiEngineLink** static parameter using following statement:

RunShortcut.LINK.getRealTime()

When necessary components can repaint themselves on the grid at any time during simulation. **Grid** class in ReTrO extends **BufferedContainer** object to speed up its graphical performance. **BufferedContainer** provide three functions to handle its screen drawings:

1. *eraseComponent(Component comp, boolean update)*

   Erases *comp* image from the buffer. This change is redrawn to the screen if *update* is true.

2. *paintComponent(Component comp, boolean update)*

   Paints *comp* image to the buffer. This change is redrawn to the screen if *update* is true.

In addition **Grid** provide a following function to redraw buffer content on the screen:

- *blitWorkplaceToScreen(Rectangle clip)*

  Redraws the content of buffer bounded by *clip* to screen. Active grid can be accessed by ReTrO's objects through static parameter **CentralPanel.ACTIVE_GRID**