Prof. Dr. Gerhard Reinelt
Dipl.-Math. Marcus Oswald
Dipl.-Inf. Dino Ahr
Institut für Informatik
Universität Heidelberg

http://www.iwr.uni-heidelberg.de/iwr/comopt/

VisAlg - Visualisation of Algorithms Programmer-Manual

July 11, 2001

Contents

1 Introduction

VisAlg is a software used for the visualisation of algorithms. First, reading the *user manual* is highly recommended. This manual describes all VisAlg concepts and their implementation from the programmers point of view. All the information needed to develop enhancements can (hopefully) be found here. Furthermore, it explains all internal implementations of the VisAlg concepts. This information is just for educational purposes, it is not recommended to make use of any of these implementation facts (like protected member names, pointers ...) in your own modules, since they are most likely to be changed in future versions.

This manual has been written by Matthias Vigelius. If you encounter any problems, mistakes and so on please contact me at : *mvigelius@gmx.de*.

2 Concepts of VisAlg

2.1 The Application

The *Application* is the main object in VisAlg, connecting the visual part (see section 2.8) and the History (see section 2.2). Its main purpose is to construct both a new *History* object and a new *ApplicationWindow* object at startup. It is furthermore responsible for the serialization of whole projects (which is the wrong expression, since in fact it serializes the History).

2.2 The History

The *History* manages all currently loaded projects (see section 2.3). It contains a list of these projects and a reference to the current project. If the user chooses another project, the History loads it and marks it as the current object. If the user saves the current project, the list is enhanced by this project, which is duplicated first. If the *step()* method is invoked (either by pressing the step button or by the step thread), the history calls the step method of the current project. Futhermore an undo functionality is realised by another list of projects, which is increased by every step invokation. If the users presses the undo button, the last project in the undo history is loaded. Beyond this, the history is capable of discarding and restoring the project windows. This is done by calling the *newWindows()* method of the current project.

Only one history object should exist at any time, although it can contain more than one project.

2.3 The Project

The *Project* mainly consists of a list of currently loaded modules (see 2.4.3) and a counter representing the number of performed steps up to now. The project represents one certain state of the demonstration, including loaded modules, currently opened windows, etc. More than one project object can exist at one time, but they are all connected to one history (see section 2.2). The project also contains a comment string, which is shown in the history window.

As a module manager (see section2.4.2), the project contains methods to add and remove modules. Although this functionality is already implemented, the user can not make use of it yet. The most important method, however, is *step*, which is invoked by the history if a global step has been inititated. Internally, the private method projectStep() will then be called, which asks every module if it is steppable, and then calls its *step()* method.

As for the module windows in the project, the project theoretically can discard and restore the windows by calling *discardWindows()* and *newWindows()*. The project will then search the modules list, and call *Module.newModuleWindow()* if its state is currently set to "showing". Additionally the project time is increased by one.

2.4 The Modular Concept of VisAlg

One of VisAlgs key features is its modular concept. In principle, a module is just a piece of code, which is capable of executing itself step by step. A module could be everything : An algorithm (which probably is the most commonly used application) but also a visual element (like a window) or even a game. By registering itself as a module manager, the module can start sub-modules and in such a way implement a recursive algorithm. Due to the object model of java, every user can develop his own modules in a safe way, without depending on the current VisAlg version. These modules could also be downloaded from the internet and thus VisAlg provides a great variety of applications.

2.4.1 The VisAlg Event Model

In order to give modules the possibility of reacting to each other, VisAlg uses an event model, which is very close to the Swing event concept. The communication between different modules is done using two interfaces : *OtherModule* and *ModuleListener*.

If a certain module wishes to let other modules be connected to itself, it just implements the interface *OtherModule*. It then overrides the two methods *addModuleListener(ModuleListener)* and *removeModuleListener(ModuleListener)* which are called if a module listener wants to register itself. Normally, these implementations add listeners to a list and call *moduleChanged(ChangeEvent)* of all modules in the list if a change has occured. Additionally, the methods *getTime()* and *toString()* have to be implemented. This standard functionality is encapsulated in the class *AbstractModule* and it is more easy to derive from this class (see section 2.4.4).

On the other hand, if a module wants to be notified if a change has occured (e.g. the data has been changed), it first gets a reference to this module by exploiting the module manager functionality (see section 2.4.2), and then registers itself as a listener. For this, it is necessary to implement the interface *ModuleListener*. Every time the module has changed, it will then call the method *moduleChanged(ModuleChangeEvent)* which of course has to be overwritten.

It is worth mentioning here, that a class can register itself as a project listener in the same way, by implementing the interface *ProjectListener*. This functionality is up to now only used by the class *ProjectWindow*.

2.4.2 The Module Manager

As already mentioned, every module (which in fact means the instance of a class) is connected to exactly one module manager. The standard module manager is the project, but a module can also act as a module manager. A module manager is responsible for the administration of modules. This is commonly done by a list, containing all modules. The module manager implements the interface *ModuleManager*. In this way, it provides functions for adding or removing modules and invoking a global step.

2.4.3 The Module

In order to have a standard interface, every module must at least implement the interface *Module*. These methods are necessary for the communication with the module manager, which in most cases is the project itself. It is highly recommended, to derive from the class AbstractModule (see section 2.4.4) rather than implementing these methods yourself.

2.4.4 Deriving from AbstractModule

AbstractModule is the recommended base class for all modules. Every module should be derived either from this class or from one of its subclasses. It provides functionality for the communication with the module manager as well as for registration and notification of submodules.

The *constructor* is called with two arguments : the name of the module and a reference to the module manager. It provides the member method *fireEvent()*, that can be called if the module has been changed. This function will notify all connected listener as well as the associated module window of the change. Using the functions *getDataModules()* respective *getModules()* the associated viewer module can easily access the list of all modules on the same level as this. That means, that these functions will not return the submodules of this one. *AbstractModule* also provides access to important member variables. Furhermore all administration for adding and removing listener is done by this class.

The following methods should be overwritten by own modules :

- *getNodeIcon()*, if another than the standard icon should appear in the tree.
- *getSubModules()* returns null. If any submodules are used, this function should be overwritten.
- *newModuleWindow()* should create a new instance of the associated viewer window.
- *step()*. This method contains all module functionality from VisAlgs point of view. It is called every time, the user initiates a step event. In your own method, the superclass method should always be called at the end, in order to set member states correctly.

2.4.5 Data Modules

The data module is the VisAlg representation of data. There is no restriction on what exactly this data is, since due to the modular data concept, every kind of data can be modelled. For more information on the data concept see section 2.6. The class *DataModule* has two main access functions, *getData()* and *getDataClass()*. These functions can be used by any module to determine the type of the data this module contains and to get a reference to the data itself.

The DataModule itself acts as a ModuleManager (see section 2.4.2). If new data is loaded or created, it reads out the available viewer classes and registers them as submodules. These can then be opened by the user.

This class is also capable of loading and saving data to a file as well as print it to the standard output stream. Its associated module window class is *DataModuleWindow*, in which the user interaction takes place.

2.4.6 Algorithms and Commands

Most VisAlg modules implement certain algorithms like sorting and so on. The easiest way to write an algorithm module is to derive it from the class *Algorithm*.

The idea behind this class is, that an algorithm consists of various commands (like pointer comparison, data manipulation) which are executed step by step. A command is implemented by an inner class derived from *AbstractCommand* and overriding its two member functions, *doIt()* and *toString()*. The first one contains the code, which is executed for this command, the latter one merely returns a string naming the command. Another possibility is, to rather implement the interface *Command* which is exactly the same. At the beginning all commands should be stored

in the array *m_commands* in order to make this list available to other modules (like the *Trigger Module*). Every command should set the next command after its execution, using *setNextCommand(Command)*. If the algorithm has finished, *setDone(boolean)* must be called with value *true*, in order to mark this algorithm as finished.

It is in principle not necessary to overwrite any of these functions, except the constructor, which must set the command list as well as the first command, and the *newModuleWindow()* method. Practically, it can be useful to overwrite for example the *step()* method. In this case, it is necessary, to call the superclass method at the end. It might also make sense, to override specific functions of the class *AbstractModule*.

2.5 The Data/Viewer Concept of VisAlg

Since VisAlg is a software, that visualises algorithms, one of its most important tasks is, to visualise data. This is done by so called Viewer Modules (see section 2.5.3). A Viewer module represents a certain kind of data, like an array of integer etc., which is hold in the Data module (see section 2.4.5). The communication between both is done using Properties (see section 2.5.1). Note, that VisAlgs concept of properties is slightly different to the JavaBean property concept.

2.5.1 Properties

The idea behind the property concept is, that information consists not merely of data, but also needs an interpretation of this data. The interpreter is in our case the module, that works on this data, like an algorithm, for example. In order to visualise an algorithm, it might be useful, to not just visualise the data, but on the other hand, some internal information of the algorithm itself. This internal data is stored in the algorithm module using properties. The algorithm registers its properties at the data module, and the viewer module looks for all properties, it can understand.

The base class for all properties is the class *PropertyObject*. Properties should be hold and serialized in the algorithm module.

Standard sorting algorithms can make use of the class *IndicatorBarProperty*, which are recognised by the class *ArrayViewer*.

2.5.2 The PropertyManager Interface

The Data module (see section 2.4.5) manages all properties, that are connected to the data hold. The communication between an algorithm and the data module is done using the interface *PropertyManager*.

An example, of how this functions are used, is given in the section 3.

2.5.3 Viewer Modules

Viewer modules visualise a certain kind of data including the properties. When the Data module (see section 2.4.5) loads or creates new data, it calls the function *VisAlgData.registerViewerModules(ModuleManager, DataModule)*. This function creates all viewer modules, that fit this data type, and registers them at the specified ModuleManager.

A Viewer module must be derived by the class *ViewerModule* and implement the methods as described there.

Most of the work is done in the Viewer module windows paint(GraphicsContext) method. The window can access the data by the data module member function *getData()* and can query the properties it can recognise by using *queryProperties(String)()*.

An example for a viewer module is the class *ArrayViewer*.

2.6 Data Representation

In principle, the data module (see section 2.4.5) can contain every data, which is represented by a java object. However, there are some classes, which make life easier for simple data.

Every data object must implement the interface *VisAlgData*. If possible, the member functions *getComponent* and *paintComponent(java.awt.Graphics, int, int)* should be overwritten in order to return a convenient graphics object and to paint itself in a useful way. For example, the class *VisAlgInt* will return a *JLabel* object and paint itself as a string.

Most data objects, for example an *int* value, can be compared to another data of the same type. For this kind of data, the class can implement the interface *VisAlgComparableData*. Apart from the methods described above, this class must also implement the java interface *Comparable*, which exactly means to implement the method *CompareTo(Object)*. There are no more specifications to the algorithm, classes can be compared to another, than the java specification gives.

Since both of the above described classes implement the interface *Cloneable*, it might necessary, to override the function *clone()*, depending on the data. For more information about this topic, consult the *Java API Documentation*.

There are two types of ready-made data classes in VisAlg : *VisAlgInt* and *VisAlgMatrix*. These classes should be sufficient for most standard algorithm purposes.

2.7 Serialization

2.7.1 Project Serialization

If the user chooses to store the project, the serialization process will be initiated. This task is very simple. The application just stores its history object, using the API function *writeObject(Object)*. This function will store the whole *History* object including all members. Hence, it is necessary for each class to implement the API interface *Serializable*. This is done automatically, if deriving from predefined superclasses.

In order to load a previously stored object, the API function *readObject()* is invoked. After that, the *transient* references have to be set. To restore the corresponding window state, the member method *newWindows()* will be called, which creates all windows of the history, including all module windows.

Additionally, the function *afterDeserialisation()* is called, after the deserialisation has been completed.

2.7.2 How the History Remembers Project States

The History is capable of remembering a certain project state, which can be restored afterwards. Internally, this is done by calling the member function *saveProject()*. This method duplicates the current *Project* object and appends it to its internal list. If the user wants to reopen this state, the list member is duplicated again, and set to be the current Project. After this, the windows need to be repainted using *newWindows()*.

2.7.3 Serialization of VisAlg Data and Creating New Data

The data in a Data Module (see section 2.4.5) can be stored and reloaded binary. This is simply done by the API serialization function *writeObject(Object)* respective the *ObjectReader* class which simply invokes the API function *ReadObject()*.

A more sophisticated task is performed, if new data should be created out of an ASCII file. After a data file is specified, the program tests the first character of the first line for *ObjectReader.converterMagicNumber*, which is currently "!". If this string exists, the rest of this line is interpreted as a class name. The class referred to by this name is then instanciated using the API function *Class.NewInstance()* and a test is performed, if this class is an instance of *Converter*. If this is the case, the member function *getData(File)* will be called, which then arranges the converting of the ASCII file to a VisAlg data object, that can be set as the current data. If the converter is not explicitly given in this file, the user will be asked, to name a converter class, that can be used for this kind of data. Converter classes are described in greater detail in the section 2.7.4.

2.7.4 Converters

A converter must implement the interface *Converter*, which consists of one method *getData(File)*. The implementation reads in the file line by line and extracts the data information, for example by using the API *StringTokenizer* class. As a result, the function must return either an VisAlg data object as described in the section 2.6 or an array of this.

2.8 Visual Concepts of VisAlg

2.8.1 Screen Layout and Interaction

The root object in VisAlgs windows hierarchy is the class *ApplicationWindow*. It sets up the main menubar and is responsible for the handling of the menu events (like loading project, edit preferences etc.). The VisAlg window is then divided into two parts. The right part contains the module windows. For this part, *ApplicationWindow* provides a desktop manager, which resets the window bounds of each module window, if the users changes its size. The left part contains the History Window and the Project Window represented by one *JSplitPane* swing object. On the bottom is a status window, containing internal information. The History window and the Project window can be set using the functions *SetProjectWindow(ProjectWindow)* and *SetHistoryWindow(HistoryWindow)*. These functions set the windows at the according position in the splitted left pane.

The History window will be created, if the function *History.newWindows()* is called, which is the case at startup. It is represented by the class *HistoryWindow*. The window itself contains the CD-Panel, which is responsible for both, the navigation through the saved project states, and the stepping of the modules. If the user chooses another project state, the history will load this state as described in the section 2.7.2. Furthermore, the user has the possibility to either perform just one step, which internally invokes the member function *History.step()*, or to start or stop a continuous stepping. This is done by an own thread, that is started, at the creation of the *HistoryWindow* object. This thread is activated or deactivated by pressing the play/pause button and invokes the function *History.step()* as well. The second visual object of this window is the table, which contains the saved project states. The user can choose one as described above.

The Project window is the second part of the split window. The associated class is *ProjectWindow* which is created, if the function *Project.newWindows()* is called at startup of if the window configuration needs to be restored, for example after deserialization. It contains a tree showing all loaded

modules. If the user doubleclicks one of them, the associated method *Module.newModuleWindow()* will be called. The window also contains a panel, giving the user the possibility to remove modules or change the module order. In future versions, the user will be able to add or remove modules freely.

2.8.2 Module Windows

A module window is the visual counterpart of a module. It is the only possibility for the user, to interact with a module. It is created by the module member function *newModuleWindow()*, which should be overwritten in order to create a new instance of the module window class as described in the section 2.4.4.

Every module window must be derived from the class *ModuleWindow*. The constructor must first call the superconstructor *ModuleWindow(String, Module, Rectangle, ModuleWindowContainer)* with the necessary arguments. These arguments must be provided by the calling function in the associated module class and can be received by the member functions of *AbstractModule*. It might be useful, to overwrite the function *moduleChanged(javax.swing.event.ChangeEvent)* to react on module events. Most module windows, for example, contain a step counter, which is increased on every step action.

The functionality, a module window provides, depends up to now only on the programmers special likings. It is indeed planned, to create a single module window superclass, which might contain a standard interface for each module window, including viewer module windows.

2.8.3 Algorithm Windows

When writing a window for an algorithm module, one can make use of the class *AlgorithmWindow*. The advantage of this class is, that it provides some standard functionality.

First, it shows the algorithm time in the window. This class then gives the user the possibility to choose a data module. At last, the developer can provide an own Menu, that will be added automatically to the main menu bar and removed, if the user closes the window.

To add a menu, the method *onLoadMenu(JMenu menu)* must be overwritten. Own menu items can be added there.

3 How To Write an Algorithm Module

This section describes on an example, how the writing of an algorithm module works. The example module, that is used, is the `DummyAlgorithm` module, written by *Martin Spoden*.

3.1 Deriving From Algorithm

Every algorithm should be derived from the class *Algorithm*, such is this :

```
public class DummyAlgorithm extends Algorithm
```

3.2 Member Variables

Member variables should be declared at the beginning of a class declaration. The only member variable, we use here, is `m_state` :

```
private int m_state;
```

Every member state should be declared `private`, unless it is absolutely needed to let other classes access member states directly.

3.3 The Constructor

The constructor calls the superclass constructor first with `super(s,mm);` and passes the arguments to it. It then sets the member variable `m_done=false;`. This command is obsolete, since the *Algorithm* sets the algorithm state to `false` automatically.

```
public DummyAlgorithm(String s, ModuleManager mm) {
     super(s, mm);
     m_done = false;
     m_commands = new AbstractCommand[4];
     m_commands[0] = new Command0();
     m_commands[1] = new Command1();
     m_commands[2] = new Command2();
     m_commands[3] = new Command3();
     m_state = 1;
     m_nextCommand = new Command0();
}
```

The next step is, to create a new *AbstractCommand* array in order to tell other modules, which commands are available in this algorithm. The array is initialised with four different command classes, which are declared as private inner classes.

The `m_state` variable is initialised with `1`, this can of course be different at other algorithms.

At last, the `m_nextCommand` variable is initialised with a `Command0` object. This is the first command, that will be executed.

3.4 The Command Classes

All command classes are declared as private inner classes. In this case, we have four different command classes.

The first one, `Command0`, increases the member variable `m_state` with one, until it has reached 50. Then the next command will be `Command1`.

```
private class Command0 extends AbstractCommand {
     public void doIt() {
          if (m_state < 50) {
               setNextCommand(new Command0());
               m_state++;
          }
          else {
               setNextCommand(new Command1());
```

```
            m_state++;
        }
    }
    public String toString() {
        return "operation 0";
    }
}
```

The command execution takes place in the overwritten metho `doIt()`, in which the member variable `m_state` is increased with one. The next command is set with the line `setNextCommand(new Command1());` with an anonymous `Command1` class created. It is absolutely necessary to set a new command, after the current has been executed. The algorithm will else get caught in an inifinite loop.

The command description is provided by the method `toString()`, that will return a short string, containing informations to this command.

The classes `Command1` and `Command2` are similar, they will not be described here.

The fourth command class is `Command3` :

```
private class Command3 extends AbstractCommand {
    public void doIt() {
        if (m_state < 200) {
            setNextCommand(new Command3());
            m_state++;
        }
        else {
            setNextCommand(null);
            setDone(true);
            m_state++;
        }
    }
    public String toString() {
        return "operation 3";
    }
}
```

The difference to the other classes is in the `else` branch of the condition. Since the algorithm has finished its work, the next command will be set to `null`. Additionally, the superclass method `setDone(true);` is called in order to tell the *ModuleManager* that this algorithm has finished.

3.5 Creating the Module Window

The last method in this class is `newModuleWindow()` :

```
public void newModuleWindow() {
    try {
        m_moduleWindow =
            new DummyAlgorithmWindow("DummyAlgorithm" + m_name,
                                     this,
                                     m_moduleWindowBounds,
```

```
                                        m_moduleManager.getModuleWindowContainer());
    } catch (Exception e) {}
}
```

The associated *ModuleWindow* class is `DummyAlgorithmWindow`. How this is written is described in the next section. Note that the creation is embedded in a `try / catch` environment in order to catch possible exceptions, that could be thrown by the Java API.

4 How To Write a Module Window Class

4.1 Deriving From Class AlgorithmWindow

A window to an algorithm module should be derived from the class *AlgorithmWindow* :

```
public class DummyAlgorithmWindow extends AlgorithmWindow {
```

4.2 Member Variables

This dummy algorithm window should show three things :

- The current module time
- The data that is currently used
- The command, that has been executed last

Note that the first two are done automatically by deriving from *AlgorithmWindow*.

We use a `JLabel` object for each information :

```
    private transient JLabel m_commandTesterLabel;
```

Additionally, it is necessary to store a reference to the `DummyAlgorithm` module :

```
private transient DummyAlgorithm m_dummyAlgorithm;
```

4.3 The Constructor

The constructor is responsible for three tasks. It first must create the window itself. This is done by the superclass constructor. It then must create the components of the window, which are in our case one `JLabel` objects. It must at last initialise internal states.

```
public DummyAlgorithmWindow(String s, DummyAlgorithm da, Rectangle bounds,
    ModuleWindowContainer mwc) {

    super(s, da, bounds, mwc);
    m_dummyAlgorithm = da;
    m_commandTesterLabel = new JLabel("last command:  " + m_dummyAlgorithm.getLastCommand());
    m_contentPane.add(m_commandTesterLabel);
    revalidate();
}
```

The superclass constructor is called at the beginning. After that, a reference to the module is stored in order to receive information of it. The next thing, that is done here, is the initialisation of the `JLabel` objects with an initial text. They have to be added to the window pane. At last, `revalidate()` is called in order to repaint the window including its components.

4.4 The Methods

The only method, that is overwritten in this example, is `moduleChanged(ChangeEvent)` :

```
public void moduleChanged(ModuleChangeEvent cEvt) {
     super.moduleChanged(cEvt);
     m_commandTesterLabel.setText("last Command:  " + m_dummyAlgorithm.getLastCommand());
     invalidate();
     repaint();
}
```

It simply sets the labels to show the right information, and then causes the window to be repainted. Note that the superclass method is called first in order to make sure, that all standard information will be shown correctly.

5 Known Bugs

- Not really a bug, but not nice : If a module acts as a ModuleManager, some methods are not unique. Developers must really take care of this !
- On serialization, not all windows (esp. viewer windows) can be restored