A Toolbox versus a Tool — A Design Approach

H.-P. Bischof
Department of Computer Science, Rochester Institute of Technology, Rochester, NY, USA
Center for Computational Relativity and Gravitation, Rochester Institute of Technology. Rochester, NY, USA

Abstract— *Tools are built to be used as they are. The members of the development team of the tool are most likely the only ones who can make modifications to the tool. Others, even if the source code is available, could in principle modify the tool, but this is in most cases extremely difficult to achieve. A toolbox, on the other hand side, is intended to be extended, or modified often by many people. Unix is a good example for a toolbox. Everybody can add new libraries or commands used by one or many. Unix itself does not need to be touched in order to add user-defined functionality to it. This paper describes design criteria for a visualization toolbox which can easily be extended.*

Keywords: Visualization System Design, Toolbox Design, Tool Design, Data Flow

1. Introduction

The first notion about the functionality of a Unix[3] pipe was written by Doug McIlroy[7]. McIlroy wrote on October 11, 1964: "We should have some ways of coupling programs like garden hose - screw in another segment when it becomes necessary to massage data in another way." The idea of how to use a pipe was later described in[5]. In order for the individual components to be able to interact with each other they must agree on a communication mechanism. McIlory described in a conversation how this could be accomplished[8]: "Write programs that handle text streams, because that is a universal interface." One philosophy behind Unix was write programs, which could do one thing well, and combine the programs in shell scripts using pipes for the communication. The author of a new command needs to know how write the output to *stdout* and how to read from *stdin*.

This allows pipe sequences like:

```
cat file | sort | lpr
```

cat sends the file to *sort* and then it gets printed. It is trivial to make this line into a distributed version without changing any of its used components:

```
cat file | ssh computer sort | lpr
```

The shell is responsible for redirecting in and output channels. Adding new components to the Unix toolbox requires little special knowledge and is easy to accomplish: add an executable anywhere in the file system and make sure that the directory is in the search path. The rest of this paper describes how the same philosophy is applied to a visualization toolbox and discusses the consequences of this.

The paper describes how to design a system, which is intended to be modified by many.

2. Visualization Tools

Most visualization systems like ParaView[6], VisIt[4], etc. apply a data flow principle to execute their programs. A visualization program is typically represented by a graph where the nodes are components and the data flows along the edges. The components' purposes ranges from read the data, filter parts of the data, stream, analyze the data, create a visual of the data, store the image, and others. Distributing the components across a network, controlling the data flow through the system, etc. is done by the runtime system of the visualization system.

The visualization graph is typically constructed with the use of a graphical programming environment. The graphical programming environment allows, besides creating the program, typically many more things, like selecting colors, specifying labels, adding decorative features, etc.

A Unix pipe is a linear graph in which each component reads from *stdin* and writes to *stdout*. The program in Listing 1 illustrates how little knowledge is required to fulfill the communication requirement for a pipe (include files are omitted). First let us look at the writer. The writer writes the content of a character array to the standard out file descriptor and then exits with exit code 0.

Listing 1: writer.c.

```
int main()
{
  char * msg = "Hello World\n";
  write(STDIN_FILENO, msg, strlen(msg));
  exit(0);
}
```

The reader reads from the standard in file descriptor and stores the read information in a character array of length 20, prints the input text, and then exits with exit code 0. This program does not deal with buffer overflow issues.

Listing 2: reader.c.

```
int main()
{
    char msg[20];
    read(STDIN_FILENO, msg, 20);
    write(STDOUT_FILENO, msg, strlen(msg));
    exit(0);

}
```

The two programs can be used in any UNIX shell environment like:

```
$ cc -o writer writer.c
$ cc -o reader reader.c
$ writer | reader
Hello World
```

The communication requirements for a visualization program are a bit more complicated, because it must be possible to connect n output channels with k input channels, n and $k \geq 1$. This is required in order to write components that do one thing, but do this one thing very well. A program of this kind is shown in Figure 1. This program reads the data for Black Hole and Gravitational Wave visualization, and sends the data to the visualization components and both components send the visual representation to a display component where the visuals will be combined.

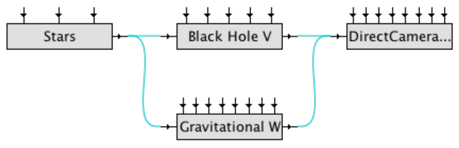


Fig. 1: A snippet of a visualization program.

3. Design Principles for a Visualization Toolbox

There is no reason to modify a visualization system if its functionality provides what is needed. The source code for many visualization systems is available. In the case of ParaView it is even possible to request a new feature. ParaView is a wonderful visualization system and as such has received the Best HPC Visualization Product Editor Choice 2016 award. As of April 5 2017, the top four requests have more than 150 votes and many more requests have single digit support. The question is: How easy or difficult is it to add functionality to an existing system. Two “Community Contributed Plugins” are available for download. Why only two? The modification of ParaView is in its complexity comparable to modifying a Unix Kernel; doable but not easy. Spiegel[2], a visualization toolbox, implemented in Java, shall be used as an example of how to design a visualization toolbox in such a way that it is easy to add functionality.

4. Separation of Functionality

A visualization tool should be divided up into a runtime system, which executes then visualization program, and programming environment. Most programming environenment are graphical programming environment because this is the easiest way to construct a program. We modeled our system based on how a shell executes components in a Unix pipe. Unix is buffering the data, and sending the data along through the system. The shell is responsible for connecting the input channels and output channels of the individual components. The functionality to execute a command of a pipe sequence remotely is implemented in the shell, not in the command.

5. Need to Know about Adding New Components

Adding a new component to the system should only require an understanding of the communication agreements and where and how to add it to the system. The next few chapters describe how this can be achieved in Spiegel.

6. Communication

The following code illustrates the definition of the communication channels. The input channels must have a name, and a type, which later on allows type safe connections. In Listing 3 we define two input channels, of type Boolean and Integer.

Listing 3: Input Channel Definition.

```
public static DataInput.Info[] input = {
  new ParamInput.Info("randomColor",
          Boolean.class)
  new ParamInput.Info("aInteger",
          Integer.class)
 }
```

The output channel follows the same structure as the input channel. More channels could be added to the block structure if needed. In Listing 4 we define one output channel, of type Integer.

Listing 4: Output Channel Definition.

```
public static DataOutput.Info output[] = {
  new DataOutput.Info("Channel",
          Integer.class)
}
```

Having typed input channels prevents the connection of an input with a ncompatible output channel. Most visualization systems use graphical editors to build the visualization programs. The connectable connections can be highlighted if typed channels are used.

7. How to Add a new Component

Adding a new component in a Unix system requires adding a component to a directory, adding the directory to the search path if necessary, and making the command executable. The Java equivalent to this is: Create a component, compile it, and add a class to $CLASSPATH$. Adding a new component to the Spiegel visualization system requires only an understanding of the Java $CLASSPATH$ mechanism, and how to code in Java. In other words no additional knowledge is needed in order to extend the system.

It is useful to have a visual programming editor to write programs for visualization systems. This requires having menus populated in a meaningful way. The Spiegel system examines the directory structure below the directory $plugin$. Every directory in this sub tree becomes a menu item like “visual”. Directories below first level directories, like “atomic” become automatically sub menu items. It is not desirable to include every class found in the class path below, and it is now and then desirable to include a component in more than one category. Determination of including a class and in which menu it appears, is decided by the values of a variable named “category”. Currently the configuration is analyzed during compile time, but in the future it will be done via reflection. Listing 5 shows the example for a component appearing under the visual and filter menu. Figure 2 shows an automatically populated menu.

Listing 5: Category.

```
public static String[] category =
           {"visual", "filter",};
```

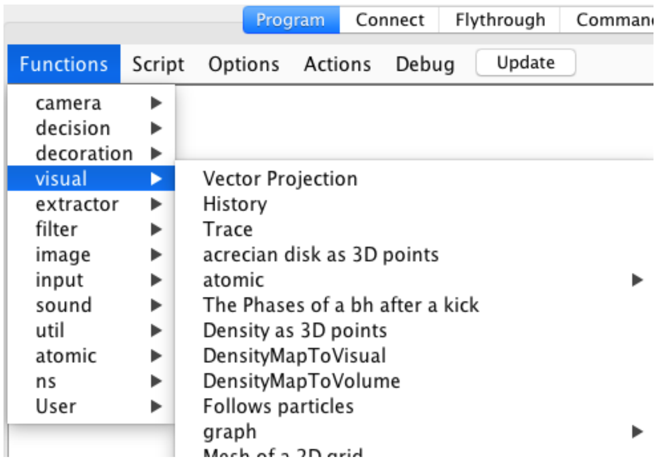


Fig. 2: Automatically populated Menu.

8. A Complete Example

This chapter is a complete exmaple of how to add a new filter to the system. The filter will read black hole information from an input stream. The black hole data structure needs to be added to the system, because it does not exist at this point. In most cases this is not required because a large number of data structures do exist. We add this data structure just to show how it is done in principle.

The file structure for the input data is describe in Listing 6

Listing 6: Category.

```
black hole id x y z
```

An example of the input is shown in Listing 7

Listing 7: Input Example snippet.

```
1.3  1.5  4.5  # time step 1
1.5  0.2  0.2  # time step 2
1.0  0.0  0.0  # time step 3
 ...
# and so on
```

The following new developments need to be made in this order:

1) a new data structure, and we will name it $BlackHole.java$
2) a component which reads the input data and sends out the BlackHole data structure and we will name it $BHreader.java$. The new data structure will be used in this component. We assume there is only one black hole per time step.
3) a filter, and we will name it $BHfilter.java$. The new data structure will be used in this component.
4) a test program and we will call it $bhTest.sprache$.

The complete system can be downloaded from http://spiegel.cs.rit.edu/∼hpb/Spiegel. $bhTest.sprache$ will not be shown in this paper. A visual representation of $bhTest.sprache$ is shown in Figure 3.

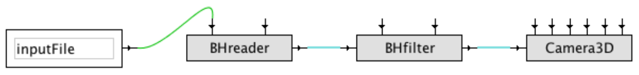


Fig. 3: Visual Representation of bhTest.sprache.

In most cases only the filter component would need to be developed, because the other pieces do already exist.

We implement 2) only in order to be able to develop a complete and running program. In most cases this would not be required, because the component would exist.

First, we implement $BlackHole.java$. Listing 8 shows the complete source code. $BlackHole.java$ is a data structure which represents one Black Hole, to be more precise the x, y, and z position of it. The constructor sets the three position instance variables, which can be accessed via the get-methods.

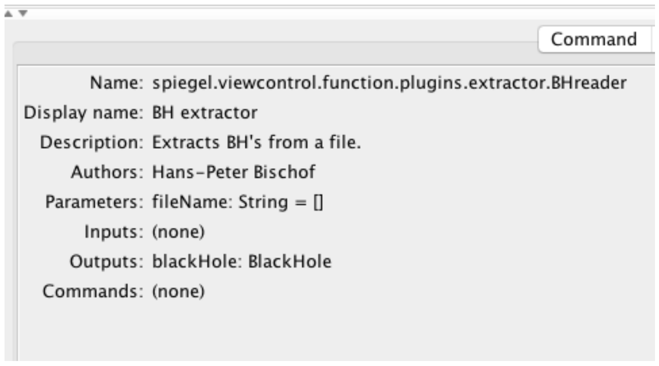
Listing 8: BlackHole.java

```
package spiegel.
        viewcontrol.
        function.
        datatypes;

public class BlackHole {

    private double x;
    private double y;
    private double z;

    public BlackHole(double x,
                     double y,
                     double z )    {
        this.x = x;
        this.y = y;
        this.z = z;

    }
    public double  getX() {
       return x;
    }
    public double  getY() {
       return y;
    }
    public double  getZ() {
       return z;
    }
    public String toString()    {
       return x+"/"+y+"/"+z+"/";
    }
}
```

The next class we implement is responsible for reading Black Hole data one by one from a file. The first part of this file is shown in Listing 9. This part shows all necessary initializations. The BHReader component is highlighted in the visual program, and the values are shown below. The input hostName is automatically set, because the base class Function includes this instance variable. The component would automatically distributed to a host, if this value would be initialized with a host name. The DataInput and DataOutput variables store the type and names of the communication end point channels.

The category for this component is extractor, because of the functionality of the component. The values of the initializations are used by the visual programming environment as shown in Figure 9 and Listing 4. This allows the developer to see the parameter types for the component, and information about who wrote the component, and a very short description about the functionality of the component.

Listing 9: BlackHoleReader Part 1



Name: spiegel.viewcontrol.function.plugins.extractor.BHreader
Display name: BH extractor
Description: Extracts BH's from a file.
Authors: Hans-Peter Bischof
Parameters: fileName: String = []
Inputs: (none)
Outputs: blackHole: BlackHole
Commands: (none)

Fig. 4: Component Information.

```
package spiegel.viewcontrol.
      function.plugins.extractor;

# some imports are not shown

import spiegel.viewcontrol.function.
      datatypes.BlackHole;

public class BHreader extends Function {
    public static String displayName
         = "BH extractor";
    public static String description
         = "Extracts BH's from a file.";
    public static String[] authors
        = { "Hans-Peter Bischof" };

    public static DataInput.Info[]
       input = {
           new ParamInput.Info
                <String>("fileName",
               ull,  String.class),
    };
    public static DataOutput.Info
        output
        = new DataOutput.Info("blackHole",
                  BlackHole.class);
    public static String[] category
         = {"extractor"};

    private ParamInput<String> fileNameIn;
    private DataOutput blackHoleOut;
```

The last part of the BHreader, shown in Listing 10, implements the method *update*. This method will be called from the runtime system when needed. First, the file name is read and then the file is opened. The file will only be opened the first time *update* is called.

Only one line will be read to initialize the BlackHole data structure. This data structure is then sent to the next component, in this case BHfilter.

Listing 10: BlackHoleReader Part 2

```
BufferedReader theBlackHoleFile = null;
protected void update() {
  String thePosition[] = null;
  String fileName = fileNameIn.get();
  try {
    if ( theBlackHoleFile != null ) {
      theBlackHoleFile =
         new BufferedReader(
         new FileReader(fileName));

       String aBH = theBlackHoleFile.
         readLine();
       thePosition = aBH.split("\\s+");
       aBlackHole = new BlackHole(
         new Double(thePosition[0]),
         new Double(thePosition[1]),
         new Double(thePosition[2]));
         blackHoleOut.set(aBlackHole);
      }
    } catch(Exception e) {
         blackHoleOut.set(null);
    }
}
```

The last component we need to implement is the class BHfilter. The first part is shown in Listing 11. Like before we need to initialize the components used for the visual programming environment.

Listing 11: BlackHoleReader Part 1

```
public class BHfilter extends Function {
    public static String
        displayName = "BHfilter";
    public static String description
        = "Displays BH's";
    public static String[] authors
        = { "Hans-Peter Bischof" };
    public static DataInput.Info[]
        input = {
          new ParamInput.Info<Double>
             ("blackHoleSize",
              new Double(0.03)
             , Double.class),
          new SingleDataInput.Info
             <BlackHole>("bh",
             BlackHole.class)
    };
    public static DataOutput.Info
        output =
           new DataOutput.Info
              ("object",
              BranchGroup.class);
    public static String[]
          category = {"visual"};

    private ParamInput
        <Double> blackHoleSizeIn;
    private SingleDataInput
         <BlackHole> bhIn;
    private DataOutput objectOut;
```

The second part is shown in Listing 12. The method *update* gets the values for size and the Black Hole Data structure first. This creates a Branchgroup, representing the visual which will be sent to the next component via objectOut.set(theBH).

Listing 12: IBlackHoleReader Part 2

```
 public void update() {
        BlackHole aBH = bhIn.get();
        double size = blackHoleSizeIn.get();
        if ( aBH != null )       {
           BranchGroup theBH
              = new BranchGroup();
           # committed
           objectOut.set(theBH);

        } else {
                objectOut.set(null);
        }
    }
```

9. Conclusion

The described paper lays out a framework which has been proven that it is easily extensible. Many undergraduate students used it on their REU experience here at RIT. We gave the students on purpose minimum instructions on how to add components and within an hour they started to add components. The students did not need to modify the runtime system.

10. Future Work

At this point the runtime system and the graphic programming environment is combined. These individual components need to be separated.

The runtime system allows a distribution of the components across a network, but there is no functionality provided for a finer distribution mechanism. This needs to be improved.

Functions can be created, but not easily. This functionality needs to be improved.

The runtime system is not optimized for speed, memory use, disk access etc. This functionality needs to be improved.

Neither the visual programming environment or the runtime system is using reflection[10] to detect usable components. The use of Java reflection would enhance the modifiability of the system.

11. Acknowledgements

The authors would like to thank all members of *The Center for Computational Relativity and Gravitation at RIT.* Their visualization needs drove much of the development of Spiegel system.

References

[1] D., Foulser, “IRIS Explorer: a framework for investigation,” *ACM SIGGRAPH Computer Graphics - Special focus: modular visualization environments (MVEs)*, vol. 29, Issue 2, pp. 13-16, Nov. 1995.

[2] H.-P. Bischof, E. Dale, and T. Peterson, “Spiegel - A Visualization Framework for Large and Small Scale Systems”, in *Proc. MSV’06*, 2006, paper, pp. 199-205.

[3] Dennis M. Ritchie, “The Evolution of the Unix Time-sharing System”, (April/2017) [Online]. Available: http://www.princeton.edu/~hos/Mahoney/expotape.htm.

[4] DHank Childs and Eric Brugger and Brad Whitlock and Jeremy Meredith and Sean Ahern and David Pugmire and Kathleen Biagas and Mark Miller and Cyrus Harrison and Gunther H. Weber and Hari Krishnan and Thomas Fogal and Allen Sanderson and Christoph Garth and E. Wes Bethel and David Camp and Oliver Ruebel and Marc Durant and Jean M. Favre and Paul Navr at all, “VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data”, High Performance Visualization–Enabling Extreme-Scale Scientific Insight, pp. 357-372, Nov. 2012.

[5] Brian W. Kernighan and P. J. Plauger, “Software Tools”, Addison-Wesley Publishing Company, 0-201-03669-X

[6] Paraview. (April/2017) [Online]. Available: http://http://www.paraview.org/

[7] Paraview. (April/2017) [Online]. Available: http://www.princeton.edu/~hos/Mahoney/expotape.htm

[8] Paraview. (April/2017) [Online]. Available: http://doc.cat-v.org/unix/pipes/

[9] David Eberly. (April/2016). Kochanek-Bartels Cubic Splines (TCB Splines). [Online]. Available: http://www.geometrictools.com/Documentation/KBSplines.pdf

[10] Ira R. Forman, and Nate Forman. *“Java Reflection in Action.”* Manning Publishing Company, 2004.