Paper 27

Developing Data-Driven Task-Oriented SAS® Powered Programs in Java™

Danielle Davis, SAS Institute Inc., San Diego, CA

Don Chapman, SAS Institute Inc., Cary, NC

Abstract

This paper describes the development of SAS powered Java programs using a toolkit that integrates key concepts from several different programming models. The toolkit introduces powerful functionality that simplifies the process.

The toolkit utilizes the following programming models:

Data-Driven	Define how a program gets built, maintained, and enhanced through changes to underlying data files.
Task-Oriented	Define how the user will interact with the program.
Event-Driven	Provide a mechanism for components to communicate with each other.

While the toolkit makes developing Java programs (applets, applications, and servlets) easier, it requires that you have experience with Java and understand the Java development environment. This paper also covers the productivity gains, development benefits, and maintainability of a program built with the toolkit.

Introduction

Designing and building a Java program can be a daunting task, especially if the program is complicated, part of a client/server system or will be deployed at multiple sites that require individual customization. In this paper we will look at a SAS powered Java program, Growth Solution, that falls into all three categories. Before jumping into the Growth Solution, let's look at some programming models that were useful in developing the program and a toolkit that integrates these concepts.

In the data-driven model the program can be built, maintained, and enhanced through changes to underlying data files. Features of the data-driven model include:

Flexibility	As requirements change a program can be modified without writing new code.
Adaptability	A program can be customized for a specific site or a specific user. Customization of visual (e.g. tools) and non-visual (e.g. login information) components are possible.
Reusability	This model requires a well-defined component model to work. In Java this means using JavaBeans™.

The task-oriented model defines how the user interacts with the program. In this model the user identifies an object, e.g. selects a node in a tree, and then performs a task on that object. Features of the task-oriented model include:

Direct Manipulation	Direct manipulation provides the user with a visual representation of what they are going to manipulate along with a set of tasks to do the manipulation.
Flexibility	The clean break between object and task make it easy to add and remove tasks and objects as requirements change.

In the event-driven model, components communicate with each other by passing events. Features of the event-driven model include:

Flexibility	By isolating program behavior in event listeners, it is easier to change that behavior- the behavior can be changed by replacing the listener.
It's a Standard	The Java component model, JavaBeans, use this model for communications.

Fortunately, these programming models are complimentary. The data-driven model provides the mechanism for defining the objects in the program and the tasks that can operate on those objects. The task-oriented model provides the framework for presenting the objects and their tasks to the user. The event-driven model provides the mechanism for connecting user-specified events to tasks.

This paper describes a toolkit that integrates concepts from these three programming models and steps through the development of a program using it. The internal SAS name for the toolkit is the ATK. SAS Consulting using the ATK developed the Java program, Growth Solution. This program is the Java version of the SAS/AF® Growth Solution program developed for Deloitte and Touche. Note – The Java version of the Growth Solution program can be run as either a stand-alone application or an applet.

ATK Overview

The ATK is a toolkit consisting of Java classes, documentation, and samples. It was built using JDK™ 1.1.7a, Swing 1.1 and SAS/IntrNet™ software. It runs with SAS Versions 6.12, 7, and 8. The ATK can be used to develop a variety of programs, but is especially useful for development of thin-client programs that use a SAS server for decision support.

As mentioned in the previous section, the name ATK is a SAS internal name and subject to change.

ATK Goals

One goal of the ATK is to promote good programming practices. Along with promoting model/view separation and code reuse, the toolkit integrates concepts from the data-driven, task-oriented, and event-driven programming models.

Another goal of the ATK is simplicity. Currently, the ATK contains less than 125 core classes that ship in a compressed jar file that is less than 175KB. The Swing support classes that are shipped with the ATK add another 75 files and a compressed jar file that is less than 100KB. The ATK includes other component libraries that may be useful. These libraries offer SAS session management, charting, and query building. The compact size not only improves download time by reducing the size of deliverables; it also decreases the learning curve required to learn how to use the toolkit.

Another goal is to simplify the development of SAS powered Java thin-clients. The ATK contains commands, models, and visuals that are frequently used when interacting with a SAS server. Below are just a few examples:

- A command and prompt for logging onto a SAS server.
- A command that allows you to submit code to a SAS server and retrieve the results.
- A command that allows you write a SQL query to retrieve data.
- A view to display a table.
- A view to display a chart.

The final goal of the ATK is to be unobtrusive. With the exception of Command classes, the ATK does not require objects to subclass ATK classes or implement ATK interfaces. This allows developers to use familiar or new class libraries along with ATK classes without modification.

Since the ATK is a toolkit, it does not require a change in development tools either. It can be used in a visual build environment or command line build environment.

It is also 100% pure Java. This allows development to occur on any platform the JDK runs on.

Programming Model Implementation

The ATK makes use of the programming models mentioned earlier. This section will review those models and show how the ATK takes advantage of their benefits.

Data-Driven Model

An advantage to a data-driven model is the ability to change and control a program by modifying one or more parameter file. A parameter file can come in many different forms. In the ATK a parameter file can be defined in two possible ways, 1) an ATK configuration file which is no more than a text file that consists of following a certain structure, or 2) a SAS data set or view.

A configuration file is a text file that contains AtkFactoryTemplates. An AtkFactoryTemplate can be used to define the properties of an object or it can be used to run a method on an object. Since a text file can be easily edited, additional templates can be easily added. These templates can define almost any object. In the Growth Solution program described in this paper, the reports displayed in the tree are defined in the TreeNodes.cfg configuration file. A new report can be added with a simple change to this configuration file.

A SAS data set or view can also serve as a parameter file with the ATK. In the Growth Solution program described later, list boxes are dynamically populated from SAS data sets. The ATK provides all the tools necessary for downloading and processing these data sources.

Task-Oriented Model

An advantage to a task-oriented model is it provides the user with the tasks that are appropriate for the data they are manipulating. To support this model, a program must be able to track the object the user is manipulating and provide a set of tasks that are applicable to the object being manipulated.

The ATK provides a mechanism for mapping data to a list of tasks that can be performed on that data. This mechanism is the AtkRegistry. The Growth Solution program does not require the complexity of the AtkRegistry because there is only one task per report. This task is run when you activate the report node.

Event-Driven Model

An advantage to the event-driven model is that objects communicate with each other by passing events. The ATK supports the standard Java mechanism of defining event sources and event listeners. In addition, the ATK provides an easy way for developers to map events to tasks. These tasks is typically a command or chain of commands.

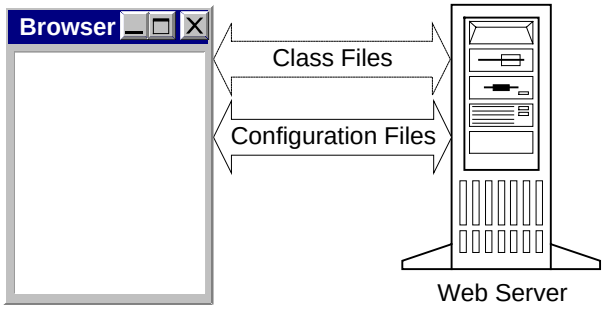
The ATK handles event processing by creating an event queue that listens for events from registered objects. This queue is called the AtkEventQueue. This queue provides a known location where objects can fire their events and other objects can listen for those events without the two knowing about each other.

In the ATK the main listener of the AtkEventQueue is the AtkEventDirector. The AtkEventDirector is useful because it takes user-defined event-to-command mappings and runs a command when the specified event occurs.

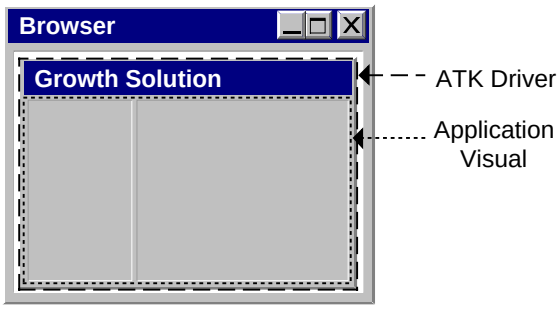
Development Issues

The process of writing a program is guided and simplified by using the ATK, but it does require a slightly different development approach. The five predominant differences are outlined below.

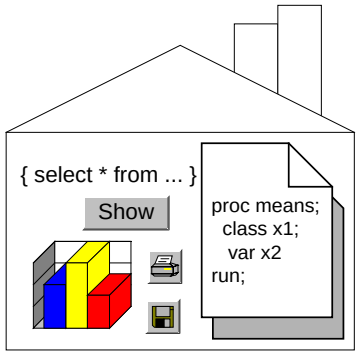
The first difference is that there is one or more configuration files used to define new object templates, customize existing object templates, define object interactions, and define user preferences. The AtkFactory loads these configuration files when the program is initialized.



The second difference is how the program is instantiated. With the ATK, a driver class provided by the ATK is specified as the applet or main class. That driver is responsible for initializing the system, displaying a user-defined visual, and running startup commands. As the developer, you provide a visual that extends Container instead of extending Applet.

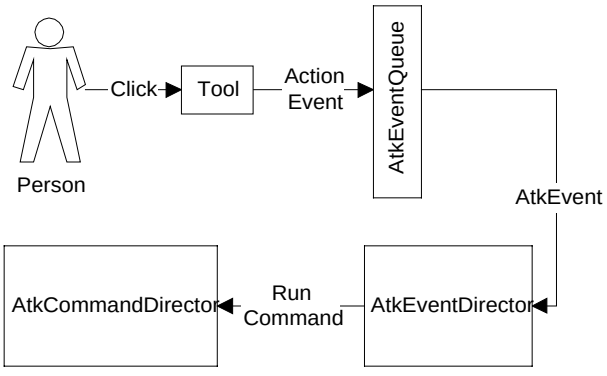


The third difference is that the ATK provides a mechanism, the AtkFactory, for managing both visual and non-visual objects. These objects can be defined in a



configuration file or in Java code. For example, a tool is an object that can be defined in a configuration file.

The fourth difference is how objects are connected to each other. The ATK manages object connections with event queues. These queues connect event sources to event listeners. For example, a tool in the toolbar is an event source. When the user selects the tool it fires an ActionEvent which is put on the AtkEventQueue. The AtkEventDirector listens for this ActionEvent and instructs the AtkCommandDirector to run a command when the event occurs.



The fifth difference is how objects are manipulated. The ATK defines it's own command architecture. This architecture is task-oriented. A command is given an object to operate on and, when it is finished, it sets its results to another object.

ATK Components

In the previous sections, terms such as AtkDriver and AtkFactory were introduced. This section will cover ATK concepts and terminology.

AtkDriver

The AtkDriver is responsible for defining the init, start, stop, and destroy behavior of an applet . It is also the entry point for a stand-alone application and is responsible for displaying a user-defined panel, known as the program visual. It is also responsible for displaying other user-defined visual features, such as a tool bar and a menu bar.

AtkFactory

The AtkFactory manages AtkFactoryTemplates, their resulting objects, and user-defined objects. An AtkFactoryTemplate is a description of how to initialize an object when it is instantiated. The resulting object is the object that is created when a template is instantiated.

The AtkFactory is loaded in phases at startup. The first phase loads templates from the configuration files. The second phase loads templates and user-defined objects from user-specified classes.

You can retrieve objects from the factory by calling the `getObject()` method. If the requested object does not exist in the factory, it is created from a template. If the requested object already exists, it is returned. If a unique object is required, you can retrieve it from the factory by calling the `newObject()` method.

AtkEventQueue

The AtkEventQueue is a mediator for event passing. It is registered as an event listener to objects in the program. Any event it receives is passed on to the appropriate listeners of the queue. Based on the type of AtkEventQueue and the type of event, the queue may transform an event into a new event and pass that event on to the appropriate listeners.

A program is composed of multiple AtkEventQueues: three created by the ATK and zero or more user-written queues. User-written queues are used as needed. The queues that are automatically created by the ATK are stored in the AtkFactory. The AtkEventQueue is the most important of the ATK-generated queues. This queue passes `AtkEvents` from program visuals to any event listeners, of which the AtkEventDirector is one.

AtkEventDirector

The AtkEventDirector is responsible for mapping an AtkEvent to a message chain and running the message chain using the AtkCommandDirector when the event occurs. A message chain defines one or more messages that map to commands. The events are passed to the AtkEventDirector from the AtkEventQueue.

AtkCommandDirector

The AtkCommandDirector is responsible for mapping a message to an AtkCommand and, when requested, running that command.

AtkCommand

An AtkCommand is the basic unit of work in the ATK. When a command is run, it is passed an object to operate on. This object is called the command source. When the command is finished it sets a result object. If two commands are chained together, the results of the first command are used as the source for the second command.

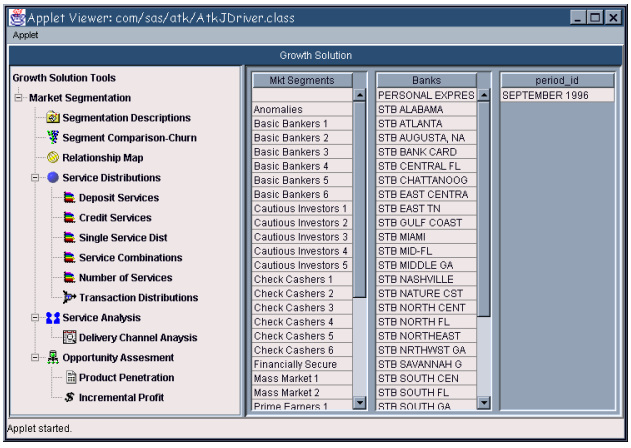
All the commands provided with the ATK require an object model, called AtkData, as their source and generate an AtkData object as their results. The results returned by these commands are the source, with new information added to the AtkData object model. This data pass-through mechanism is very useful in command chaining.

AtkData

An AtkData object is a generic piece of data. It is composed of name/value pairs, where the name is specified as an AtkDataKey.

Growth Solution

Now that you have an overview of the ATK components let's see how they apply to a real program. We will show how each component of the ATK was implemented in the program and step through key sections of the configuration files. First, let's look at the main screen of the Growth Solution program.



The tree menu to the left provides the task-oriented design. Each node on the tree is assigned a task. The list boxes on the right allow the user to subset the data before running a task and displaying the results. This tree diagram is defined in the main configuration file. Before we discuss the configuration files let's look at the html file for the applet.

```
<html><body><center>
<applet code="com/sas/atk/AtkJDriver.class" codebase="../../"
  archive="atkcore.jar, atkjfc.jar, atkx.jar, ssiwq.jar, chart.jar,
           swingall.jar"
  width=710 height=450>
<param name="ProgramClass"
  value="sample.growSol.GrowthSolutionModel">
<param name="ConfigFiles"   value= "GrowthSolution.cfg,
TreeReports.cfg, TreeNodeViews.cfg, Host.cfg">
</applet>
</center></body></html>
```

HTML

Notice several things in this html file:

- The main class you are calling for the applet is the `AtkJDriver.class` file.
- The archive consists of several different jar files. All except the `atkcore.jar` file are optional.
- The parameter "`ProgramClass`" is required.
- The parameter "`ConfigFiles`" requires at least one main configuration file. Ours is GrowthSolution.cfg.

The AtkJDriver class is a Swing implementation of the AtkDriver defined earlier in the paper. It is always the main class you call for your program. All of the .jar files contain different functionality to use with the ATK depending on your needs. The atkcore.jar contains the essentials for using the ATK. The "ProgramClass" is the main model for your program. It contains key objects such as the SAS session your applet is connected to. The "ConfigFiles" contain information that provides the data-driven model capabilities.

The information in the configuration files defines object interactions and user preferences by creating an AtkFactoryTemplate. A template is specified by [*template name*]. The AtkFactory uses templates to control and manage the behavior of the program. Every ATK program will have one main configuration file. Most configuration files will define the following:

- Customization of the ATK driver [AtkDriver].
- Definition of the program model [ProgramModel] and program visual [ProgramVisual].
- Connections between objects [GrowthSolutionEventQueue] and [AtkEventQueue].
- Prompts used by commands. Growth Solution defines [LoginPrompt] and [ExitPrompt].
- Mappings between messages and commands [AtkCommandDirector].
- Commands. Growth Solution defines [ExitCmd] and [LoadMenuCmd].

GrowthSolution.cfg

The following code segment customizes the AtkDriver by defining the factory Loaders and defining which commands are run at start-up. It also defines the visual that is used once all of the commands have been run. Notice that each of these parameters start with a "set". They all have matching methods in the driver that map to each of these.

```
[AtkDriver]
setFactoryLoaders({"ProgramVisual"})
setStartMessage("Login|Libname|LoadSubsetCriteria|
  LoadTreeReports|UpdateProgram")
setStopMessage("Exit")
setVisual(%getObject("ProgramVisual"))
setTitle("Growth Solution")
```

The following code segment is the template that defines the main program model. This is a custom model and contains its own custom methods (such as setTreeKey() and setSubsetKey()).

```
[ProgramModel]
$Class=sample.growSol.GrowthSolutionModel
setTreeKey("TreeData")
setSubsetKey("SubsetData")
```

The following code segment defines the main program visual (the tree diagram and the list boxes on the right). This is a custom view that uses java Swing components for the display. Its model is described above. The program model stores the models needs to populate these visuals.

The following code segment defines the GrowthSolutionEventQueue. This event queue is the mechanism that allows the program model to send PropertyChangeEvents to the program visual.

```
[GrowthSolutionEventQueue]
$Class=com.sas.atk.event.AtkPropertyChangeEventQueue
addSource(%getObject("ProgramModel"),
  "addPropertyChangeListener")
addPropertyChangeListener(%getObject("ProgramVisual"))
setName("Growth Solution Event Queue")
```

The following code segment defines the templates for any prompts that may be used in the program:

```
[LoginPrompt]
$Class = com.sas.atk.command.AtkJLoginPrompt
setMessage(%getStringResource("LoginPrompt.Title"))

[ExitPrompt]
$Class = com.sas.atk.command.AtkJConfirmPrompt
setMessage(%getStringResource("ExitPrompt.Title"))
```

The following code segment customizes the AtkEventDirector. The AtkEventDirector maps the action of double clicking on a node in the tree to a command. The tree and its supporting action adapters, "TreeActor", were defined in the program visual and added to the AtkFactory in Java code. By adding these objects to the AtkFactory, they are available in the configuration file.

```
[AtkEventDirector]
addMapping(%getObject(TreeActor), actionPerformed,
  "GetTreeNodeReport")
```

The following code segment customizes the AtkCommandDirector. This maps event messages to commands. We will only show a few lines to give you a feeling of how this command gets defined. (The messages used in the AtkDriver for initial start up are defined here as well.)

```
[AtkCommandDirector]
# messages mappings
addCommand("Exit", "ExitCmd")
addCommand("GetTreeNodeReport", "GetTreeNodeCmd")
addCommand("LoadTreeReports", "LoadTreeReportsCmd")
addCommand("LoadSubsetCriteria", "LoadSubsetCriteriaCmd")
addCommand("Login", "LoginCmd")
addCommand("Libname", "LibnameCmd")
addCommand("UpdateProgram", "UpdateProgramCmd")
addCommand("ViewChart", "ViewChartCmd")
addCommand("ViewTwoChartTable", "ViewTwoChartTableCmd")
addCommand("ViewTable", "ViewTableCmd")
addCommand("ViewTabChart", "ViewTabChartCmd")
addCommand("GetResultsTable", "GetSubmitTableCmd")
addCommand("SegCompareChurn", "SegCompareChurnCmd")
addCommand("SegCompareChurnQuery","SegCompareChurnSQL")
addCommand("DepositServ", "DepositServCmd")
addCommand("DepositServQuery", "DepositServQueryCmd")
addCommand("DelivChannel", "DelivChannelCmd")
```

Notice that each message has a correlating command to go with it. We will see how these are defined next. Remember, all of the lines following the “$Class” definition line map to corresponding methods in the class.

The following code segment defines a template for a command that loads the individual nodes that make up the tree diagram. We have defined a method, addTreeNode(), in our custom command that takes the following parameters: NodeId, NodeName, Parent, messageChain. We are only showing a portion of defining the tree nodes for brevity.

Each of the messages in the chain maps to a command in the AtkCommandDirector defined previously. When you “chain” messages together (a “|” is the chain deliminator) the ATK automatically takes the results of the previous command and allows it to be accessible by the next command. Notice some method calls have no message chain assigned. This defines them as just parent nodes.

```
[LoadTreeReportsCmd]
$Class = samples.growSol.LoadTreeReportsCmd
setDescription("Loading Tree of Available Reports")
addTreeNode("GSTools", "Growth Solution Tools", "Root")
addTreeNode("Mkt", "Market Segmentation", "GSTools")
addTreeNode("report1", "SegmentationDescriptions", "Mkt",
  "SegDescription|ViewListOutput")
addTreeNode("report2", "Segment Comparison-Churn", "Mkt",
  "SegCompareChurn|GetResultsTable|ViewTable")
addTreeNode("ServDist", "Service Distributions", "Mkt")
addTreeNode("report4", "Deposit Services", "ServDist",
  "DepositServ|GetResultsTable|DepositServQuery|
    ViewTwoChartTable")
addTreeNode("ServAna", "Service Analysis", "Mkt")
addTreeNode("report10", "Delivery Channel Anaysis", "ServAna",
  "DelivChannel|GetResultsTable|ViewTabChart")
setResultsKey("TreeData")
```

The following code segment defines a template for the custom command we use to query parameter data sets to

define models for each of the list boxes shown on the right. If the developer needed to add another list box all he would need to do is add the data set name here along with the variable that contained the values. The command and visual would take care of the rest.

```
[LoadSubsetCriteriaCmd]
$Class = samples.growSol.LoadSubsetCriteriaCmd
setCriteriaTables({ "SASDL.MKTSEG", "SASDL.BANKS",
  "SASDL.PTHMDIST" })
setCriteriaDisplayValues({ "MKTSEG", "BANK", "PERIOD" })
setResultsKey("SubsetData")
```

The following code segment defines a template for the command that gets run when a user selects a node. This command accesses the message chain defined for a node and runs the associated commands.

```
[GetTreeNodeCmd]
$Class = samples.growSol.GetTreeNodeCmd
setNodeSource(%getObject("ProgramModel"))
```

The following code segment defines a template for the results of the previous commands, LoadTreeReportsCmd and LoadSubsetCriteriaCmd, and stores that information in the program model. When the program model gets updated, it will notify the program visual that it has changed and the program visual will redisplay the tree and lists.

```
[UpdateProgramCmd]
$Class = com.sas.atk.command.AtkUpdateModelCmd
setModel(%getObject("ProgramModel"))
setUpdateMethod("setModel")
```

The following code segment defines four templates that all utilize the same command and providing different information to the same method. This command provides the mechanism to extract the correct type of data model from the results of the previous command run. It also links the actual visual that will be expecting this type of model and displaying the results, setViewName(). For our example, we always need a “table model”. This could change and therefore a different command may be needed to provide the link between model and view.

```
[ViewChartCmd]
$Class = com.sas.atkx.command.AtkViewTableCmd
setViewName("ChartWindow")

[ViewTabChartCmd]
$Class = com.sas.atkx.command.AtkViewTableCmd
setViewName("TabChartWindow")

[ViewTwoChartTableCmd]
$Class = com.sas.atkx.command.AtkViewTableCmd
setViewName("TwoChartTableWindow")

[ViewTableCmd]
$Class = com.sas.atkx.command.AtkViewTableCmd
setViewName("TableWindow")
```

This completes the definition of the GrowthSolution.cfg file. However, we also have three other configuration files that define other types of templates. There is no other reason to have separate configuration files except for easier maintenance.

TreeNodes.cfg

For this program we have a separate configuration file for defining the commands that are used in the message chains of the tree nodes. Let’s review a few templates in this configuration file.

The following code segment defines a template for a command that can retrieve a “table format” model from the results of SAS code that was submitted.

```
[GetSubmitTableCmd]
$Class=com.sas.atk.command.AtkGetSubmitResultCmd
setResultLabel("Submit Table Results")
```

The following code segment defines three templates for commands that submit SAS code. The method setSubmitString() will pass a string into the command that will contain the needed SAS code. You will notice that we are using a %run() within setSubmitString() to call a method in a custom object. This object defines the SAS code needed for each template definition. The %run(), a macro, call expects its first parameter the to be the name of the class. The second parameter is the method in the class to call. That method getReportCode() is passed the program model. This allows us to retrieve any subset information needed from the user selecting values from our list boxes from the main program visual. The next method addTableResults() tells the command to provide a “table format” model as part of the results. Notice that the key name that we provide, “Submit Table Results”, is the same name that is used in the template above. This provides the previous template the key to look for when accessing the results for a table model.

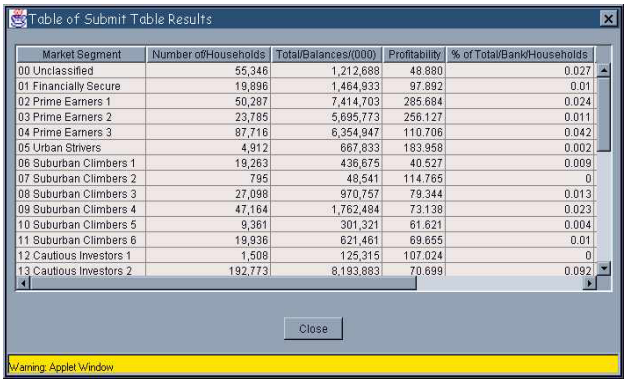
```
[SegCompareChurnCmd]
$Class = com.sas.atk.command.AtkSubmitCmd
setSubmitString(%run(samples.growSol.sascode.SegCompareChurn
Code,
   "getReportCode(%getObject("ProgramModel"))" ))
addTableResults("Submit Table Results", "WORK", "SEGCOMP")
setDescription("Segment Comparison Churn Report")

[DepositServCmd]
$Class = com.sas.atk.command.AtkSubmitCmd
setSubmitString(%run(samples.growSol.sascode.ServiceDistCode,
  "getReportCode(1,%getObject("ProgramModel"))" ))
addTableResults(("Submit Table Results", "WORK", "SD")
setDescription("Deposit Services Report")

[DelivChannelCmd]
$Class = com.sas.atk.command.AtkSubmitCmd
setSubmitString(%run(samples.growSol.sascode.DelivChannelCode,
   "getReportCode(%getObject("ProgramModel"))" ))
addTableResults("Submit Table Results","WORK","DC")
setDescription("Delivery Channel Report")
```

The following code segment defines a template for a SQL query command. This is the next command in the chain after the DepositServCmd. Therefore, it will take the results of the previous command and run a query with the following “select clause” and “where clause”. There is no

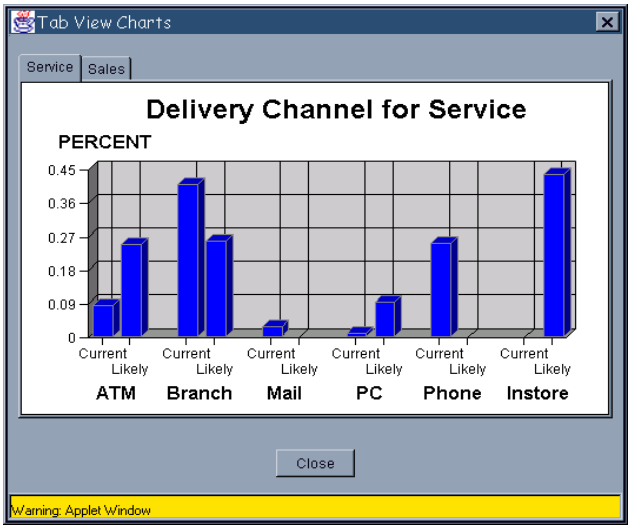
"from clause" defined since that information is coming from the results of the previous command.

```
[DepositServQueryCmd]
$Class = com.sas.atk.command.AtkJDBCQueryCmd
setSelect({ "PRODTYPE","PRODCAT", "HH","pcthh", "totres",
  "pctres","AVGRES" })
setWhere("_TYPE_ EQ 2")
```

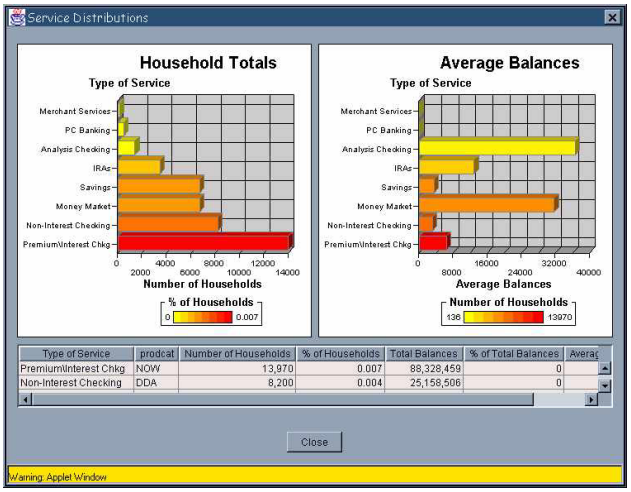
We also have another configuration file that defines the resulting views for each node in the tree. Before we review this configuration file Let's take a look at the resulting views of three different nodes from the tree.



Segment Comparison/Churn Results view.



Delivery Channel Results View



Deposit Services Results view.

Now let's review this configuration file that allowed us to generate these results. The following code segment defines three templates for windows that will contain a custom panel. The three are calling the same class and the same method but sending different values to the

```
[TableWindow]
$Class = com.sas.atk.visual.AtkJDefaultView
setPanelClass(com.sas.atkx.visual.AtkJTablePanel)
setTitle("Table of Results")

[TabChartWindow]
$Class = com.sas.atk.visual.AtkJDefaultView
setPanel(%newObject("TabChartPanel"))
setTitle("Tab View Charts")

[TwoChartTableWindow]
$Class = com.sas.atk.visual.AtkJDefaultView
setPanel(%newObject("TwoChartTablePanel"))
setTitle("Service Distributions")
```

method.

The following code segment defines the templates for custom panels that will be used for the view. Notice the second template is also dynamically creating objects for its method calls using %newObject(). These new objects also have templates defined for them. This allows the developer to easily change the type of chart that will be displayed in the panel.

```
[TabChartPanel]
$Class = samples.growSol.TabChartVisual
setTabVariable("AREA")

[TwoChartTablePanel]
$Class = samples.growSol.TwoChartTableVisual
setRightChart(%newObject("ChartOne"))
setLeftChart(%newObject("ChartTwo"))
setTable(%newObject("Table"))

[ChartOne]
$Class = com.sas.graphics.chart.Bar
setTitle("Household Totals")
setCategoryVariableName("Type of Service")
setResponseVariableName("Number of Households")

[ChartTwo]
$Class = com.sas.graphics.chart.Bar
setTitle("Average Balances")
setCategoryVariableName("Type of Service")
setResponseVariableName("Average Balances")
setVertical(false)

[Table]
$Class = javax.swing.JTable

[TablePanel]
$Class = com.sas.atkx.visual.AtkJTablePanel
```

Host.cfg

Lastly we have a configuration file that defines the host-dependent information about the SAS Server connection needed. This template takes advantage of a ATK command that handles connecting to a SAS server.

```
[LoginCmd]
$Class = com.sas.atk.command.AtkSASConnectCmd
setHost(localhost)
setProtocol(connect)
setPort(2323)
setPrompt1("Hello>")
setResponse1("c:\\sas\\sas -nosyntaxcheck -cleanup -dmr")
setPromptTimeout1(0)
setPromptTimeout2(0)
setPromptTimeout3(5)
setResponseTimeout1(5)
setResponseTimeout2(5)
setResponseTimeout3(5)
setSASPortTagTimeout(20)

[LibnameCmd]
$Class = com.sas.atk.command.AtkSubmitCmd
setDescription("Accessing Data Servers")
setSubmitString("libname sasdl 'c:\\dev\\sasdl';
  libname system 'c:\\dev\\system';")
```

We have finished reviewing the configuration files. There is a lot there but it also provides the user the ability to customize and maintain the program without having to modify/write any Java code.

Custom Java Classes

Many of the classes used in this program are provided by the ATK. These ATK classes are designed to handle a certain type of task and be easily defined in a configuration file. Other classes are sub-classed from existing ATK classes. Typically these are commands. Commands are required to be sub-classed from an ATK command class. Only the visual pieces were written from scratch and they take advantage of Java's Swing components. This section will review several of the custom classes created for this program.

GrowthSolutionModel Class

This class is the ProgramModel. It is a composite model that extends AtkPropertyChangeSource which emits PropertyChangeEvents. It will notify the program visual when one of its models has changed. The methods are:

getTreeModel()

> Retrieves the model to populate the tree diagram.

getSubsetModel()

> Retrieves the model to populate the list boxes.

getModel()

> Retrieves the program model. There is also a setModel() method.

addSelection(subsetName, selections)

> Adds selections from a particular list box. There is also a getSelection() method.

getNumberSelections()

> Retrieves the number of list boxes the user made choices from.

setTreeKey(String treeKey)

> Sets the key in the program model to locate the tree diagram model. There is also a getTreeKey() method.

setSubsetKey(String subsetKey)

> Sets the key for the model to populate the listboxes. There is also a getSubsetKey() method.

The reason this class has quite a few additional methods is due to the fact that multiple models need to populate the visual.

GrowthSolutionMenuVisual Class

This class is the ProgramVisual. It simply extends a java Swing component JPanel. It does implement the PropertyChangeListener interface so that it can listen for changes in the Program model. The methods are:

loadAtkFactory(AtkFactory factory)

> Loads the ATK factory do that it can load and communicate with the factory's templates.

showMenu()

Draws the visual.

setTitle()

Sets the title of visual window

propertyChange( )

Provides a listener to the program model

getTreeCellRenderer()

Called in the constructor in order to apply appropriate icons to each node displayed.

LoadTreeReportsCmd Class

Since this is a command it needs to extend the AtkCmd class. The methods are:

init()

Initializes the command and validate the source coming in. In our example the source for this class would be the Growth Solution program model.

run()

Runs the command. In our example we create a tree model

addTreeNode()

Adds a node to the tree.

getResultsKey()

Gets the unqiue key that defines the location of the tree model in the program model. There is also a setResultsKey() method.

LoadSubsetCriteriaCmd Class

Once again this class extends the AtkCmd class contained in the ATK toolkit. The methods are:

init()

Initializes the command and validate the source coming in. In our example the source for this class would be the Growth Solution program model.

run()

Runs the command. In our example we create a tree model.

setResultsKey()

Gets the unique key that defines the location of the tree model in the program model. there is also a setResultsKey() method.

setCriteriaTables()

Set the names of the tables used to populate the list boxes. This comes in as an array. The number of items in the array decides the number of list boxes in the visual. There is also a getCriteriaTables() method.

setCriteriaDisplayValues()

Set the variables names in the corresponding tables mentioned above. This variable contains the values used to populate the list boxes. There is also a getCriteriaSubsetValues() method.

TabChartVisual Class

This class was used to display the results of the node “Delivery Channels”. It subclasses the JPanel class of the java Swing package. The methods are:

addNotify()

This is an overriden method. Before super() is called we add the necessary visuals to the panel and attach their models.

setModel()

This method is required to allow the final command to link to the visual. This sets the appropriate model needed for the visuals. There is also a getModel() method.

setTabVariable()

Sets the name of the column in the model that will uniquely define the tabs for each graph as well as creating separate models for each chart on each tab.

setCategoryVariableName()

Sets the category variable for the chart.

setResponseVariableName()

Sets the response variable for the chart.

setGroupVariableName()

Sets the group variable for the chart.

TwoChartTableVisual Class

This class was used to display the results of the node “Deposit Services”. It subclasses the JPanel class of the java Swing package. The methods are:

addNotify()

This is an overriden method. Before super() is called we add the necessary visuals to the panel and attach their model(s).

setModel()

This method is required to allow the final command to link to the visual. This sets the appropriate model needed for the visuals. There is also a getModel() method.

setRightChart()

Sets the developer-defined chart. Remember this was dynamically defined in the TreeNodeView.cfg file. This allows the developer to choose the type of chart needed without having to write any java code.

setLeftChart()

Sets the developer defined chart.

setTable()

Sets the developer defined table.

Summary

This paper provided an overview of the ATK described the major components that comprise it, and presented an example of a real-world program, Growth Solution, built with the toolkit. We demonstrated the three key benefits of the ATK:

Productivity gains	Providing a developer a data-driven, task-oriented, event-driven environment for quickly developing a SAS Powered Java program.
Development benefits	The configuration files that run with the ATK provide the developer the ability to create a Java program with minimal Java programming.
Ease of maintenance	The ability to enhance or modify the program can easily be done through changing the configuration file(s).

References

http://www.sas.com/rnd/web – This is the home page for SAS/IntrNet software.

Authors

Danielle Davis
SAS Institute Inc.
10755 Scripps Poway Pkwy Suite 8F
San Diego, CA 92131
(619) 566-8502
sasdcj@sas.com

Don Chapman
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000 x6707
sasdnc@sas.com

Acknowledgements

The authors would like to thank the following reviewers:

- ♣ Bryan Boone
- ♣ Renee Harper
- ♣ Matthew Labarge
- ♣ Barbara Walters