

FastPak for Java
Developers Guide

Legal Information

All Software and Computer Systems Company, LLC logos are a trademark (TM) of Software and Computer Systems Company, LLC. *JWaveScope* and *FastPak for Java* are trademark (TM) of Software and Computer Systems Company, LLC. All software produced and licensed by Software and Computer Systems Company, LLC is copyright© Software and Computer Systems Company, LLC 2005 – 2007. The contents of all pages and images contained in this document are copyright© Software and Computer Systems Company, LLC, 2007,

Sun, Sun Microsystems, Solaris and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. X Windows and the X Window System are trademarks of the X Consortium. UNIX is a registered trademark in the United States and other countries of the X/Open Company, Ltd. Apple Macintosh and OS X are trademarks of Apple Computer, Inc. Novell is a registered trademark of Novell, Inc., and SUSE is a trademark of SUSE LINUX Products GmbH, a Novell business. Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Motif is a registered trademark of The Open Group.

Unless explicitly stated, original products and services offered, sold, or licensed to customers by Software and Computer Systems, LLC are the exclusive right of Software and Computer Systems Company, LLC. Clients, users, or interested parties should not assume an affiliation exists between Software and Computer Systems Company, LLC and any of the computer manufacturers, operating system distributors, or other vendors that may be used in the production or completion of a work produced by Software and Computer Systems Company, LLC for a customer or product.

Table of Contents

Introduction	4
Terminology	5
General Application Programming and Development Considerations	8
The FastPak Controller Thread	25
The FastPak API	26
The FastPak Protocol	40
Developing a General Controller for FastPak	45
Developing a GUI Controller for FastPak	50
FastPak Security Issues	53

Introduction

Welcome to the ***FastPak for Java Developers Guide***.

This document will detail programming and development considerations that one may need to confront when using ***FastPak for Java***, or more briefly, ***FastPak***. This document will describe the changes that may need to be made to an application to make it optimally functional under ***FastPak***; discuss the ***FastPak Controller Thread***, the ***FastPak*** API, and the ***FastPak*** protocol; describe how to develop and use the ***FastPak*** API as a custom GUI or part of another remote application; and discuss some security issues. Developers using this document should not be novices.

Although ultimately ***FastPak*** ends up in the hands of an end user that is executing Java applications with it, ***FastPak*** should be seen first and foremost as a developers tool. ***FastPak*** is not a be-all end-all solution for application launching under Java. It is a tool that, if properly understood and implemented, can enhance and improve a system's overall performance and hopefully create an environment where using Java applications is easier than ever.

Before continuing, it should be noted that ***FastPak*** cannot run all applications. For example, ***FastPak*** does not support SWT, hence any application developed using SWT should be executed outside a ***FastPak*** environment. Limitations regarding the use of ***FastPak*** may be found in the document titled ***FastPak for Java A Technical Overview***.

Terminology

When reading this document, the following list of terms and abbreviations will be used and defined as follows:

- **Application Profile:** An application profile is a file that is configured by a user, developer, or administrator to tell **FastPak** how to launch an application. This file allows the user to do such things as send the application command line parameters, override preassigned command line parameters, set the application's main thread priority with respect to other applications running under **FastPak**, and automatically launch specified applications on **FastPak** start up, to name just a few. Details about the application profile are provided in the advanced users guide.
- **EDT:** This stands for the event dispatching thread. This is the thread in the JVM that handles all graphics updates and processing. If this thread is blocked waiting for something (a network connection is a good example) it can end up blocking all graphics processing.
- **Flat application:** A flat application is a group of class files that have been created using a Java compiler. These files are not archived, but rather exist in a directory structure, most often as a set of many individual class files. All references to the class files are absolute and require explicit references to the directories containing the class files if attempts are made to execute the flat application outside its own directory structure. Most Java application launchers cannot launch flat applications, but **FastPak** can, provided they are configured under **FastPak** properly.
- **GUI:** This is an abbreviation for graphical user interface. This term may be used for operating system components as well as specific applications.
- **Jar application:** A "jar application" is a Java application that has been compiled into a series of class files and the class files are then archived into a single bundle with the Java archiving tool, "jar." Users can often (but not always) launch such an application from an OS GUI or a command line without needing to supply any external links, references, library paths, etc. for the application to run. Jar applications under **FastPak** may also reference other jar archives as libraries provided their class paths are accessible to both the JVM and **FastPak**. This document will not go into any of the specifics of creating a jar based application or how to use the "jar" tool.
- **Java application, or application:** This is an application that has been compiled using a Java compiler. In this context, the Java application, or just application for short, is one that has a main() method and is executed by loading it into a JVM which will then launch it. Unless otherwise stated, all references in this document to an application will be referring to a Java application. **FastPak** can run two types of Java applications, which will be referred to as a "jar application" and a "flat application." Both of these are described in this section.
- **JDK:** This is an abbreviation for Java developers kit. This is a kit that includes

the JRE and its associated JVM (both described below) as well as compilers, libraries and a host of tools used to build and assist in the build of a Java application. The release levels of a JDK and its associated JRE are identical.

NOTE: Applications built using a JDK can typically be run on a JVM of an equal or higher release level, but not the opposite. For example, if an application was compiled using JDK 1.2, then in most cases it will run problem free on a JVM associated with releases of 1.2 and above (which would presently be 1.3, 1.4, 1.5, and 1.6). However, an application compiled using a JDK of 1.5 would generally not work on earlier JVMs (in this case, 1.4 and earlier) unless the code was compiled to specifically generate code for older JVMs, which is typically not done.

- **JNI:** JNI stands for Java native interface. Applications using JNI are linking methods and/or libraries that are compiled using a native language such as C or C++ into a Java application. Use of JNI is platform specific and its implementation varies considerably from one operating system/hardware configuration to another.
- **JRE:** This is an abbreviation for Java runtime environment. This is typically a package that is installed on a system that contains the corresponding JVM. Release levels of the two are similar if not identical. For example, a JRE release may be designated 1.4.2 and the corresponding JVM packaged in the release may report something like 1.4.2_09. The JRE typically includes a set of libraries and jar files that are used by applications as well.
- **JVM:** This is an abbreviation for the Java Virtual Machine. Each JVM is seen by the hosting operating system as a resource intensive application. **FastPak for Java** differs from other Java application launchers in that only a single JVM is launched to run multiple applications.
- **Launch time:** This term refers to the amount of time it takes for a given Java application to reach a fully operational state once started. For example, if a standalone jar application named "TestIt.jar" is launched from a terminal or console window with the command "java -jar TestIt.jar", the launch time would be the amount of time it takes for the application to reach a fully operational state once the user hits the return/enter key after typing in the command. Launch time under **FastPak** is different because under **FastPak** the JVM is already in a fully operational state prior to application launch, and launch time will usually consist of the amount of time it takes for the user to select the desired application from a list using a mouse and then just launch the application. Launch times under **FastPak** are much faster than they are when launching applications with discrete JVMs.
- **SCSC:** This is an abbreviation for **Software and Computer Systems Company, LLC**, which is the company responsible for all aspects of **FastPak for Java**.
- **SWT:** SWT stands for standard widget toolkit. This is a non-Java standard GUI implementation that **FastPak** currently does not support.
- **Targeted Instance of FastPak:** This generally refers to a **FastPak** instance that

is interacting with an application via the *Controller Thread*. For example, suppose an instance of **FastPak** is running on machine A and its *Controller Thread* is running. If a developer creates a program using the **FastPak** API, and installs it on machine B, this new application can communicate and control the **FastPak** instance on machine A. With respect to machine B, the **FastPak** instance on machine A is considered to be the targeted instance of **FastPak**. This is an important concept, because if a developer sends a command from machine B to the targeted instance of **FastPak** on machine A instructing it to launch an application, the application (assuming there are no errors) will launch on machine A. The only case where this might be different occurs if the launching machine is using the X-Window system and display variables have been modified so the targetted instance maps all video to what it sees as the remote machine.

General Application Programming and Development Considerations

This section is about Java applications and developing or modifying them for optimum use with **FastPak**.

Generally speaking, if an application has been developed using the Java language and there are no “unusual” extensions such as libraries **FastPak** may not work with, the application should launch and run successfully under **FastPak**. The intent of **FastPak** is, however, to run many applications under a single JVM, and in this context, be able to run, terminate, re-start, and start additional instances of the same application without problems. For this reason, the concepts of a **FastPak friendly** application and a **FastPak hostile** application have been created. These terms are characterized in the sections below.

FastPak Friendly Application – A **FastPak** friendly application is one that meets the following requirements, which will be detailed further in the document:

- **Requirement 1: FastPak friendly applications must not be FastPak specific.** A **FastPak** friendly application can run under **FastPak**, and it can run by manually launching it under a JVM completely independent of **FastPak**, or any other standard means of running Java applications.
- **Requirement 2: Multiple instances of a FastPak friendly application can be launched under FastPak without interfering with one another.** Multiple instances of such an application can be started under **FastPak**, and unless deliberate steps have been taken to do otherwise, each application will behave completely independently of the other, even though each instance of the application may have been started with the exact same set of parameters.
- **Requirement 3: A FastPak friendly application will not, upon termination, cause FastPak or any other applications running underneath it to terminate as well.** An instance of such an application can be started and terminated without terminating or interfering with **FastPak** itself, as well as any applications running underneath **FastPak**.
- **Requirement 4: A FastPak friendly GUI application will show the utmost respect for the event dispatching thread, even if the application appears to run well under a standalone JVM.** If the application is an active graphics application, it has unloaded the event dispatching thread (EDT), meaning the generation of such things as images is not done directly in code such as the `paint()`, `repaint()`, and `update()` methods, to name a few.
- **Requirement 5: A FastPak friendly GUI application will not block the event dispatching thread waiting for some external or other internal source to complete before allowing processing to continue.** The application doesn't block the EDT waiting on other data from an external source.
- **Requirement 6: A FastPak friendly application doesn't use multiple main() methods.** This is self explanatory.
- **Requirement 7: A FastPak friendly application doesn't kill threads under**

its own application or those associated with other applications or *FastPak* itself. This does not refer to stopping a thread by sending it a message or command in some way and having the thread clean up and exit, but rather the deliberate and abrupt termination of a thread by external means. This will be described in more detail later.

- **Requirement 8: A *FastPak* friendly application observes only the careful use of JNI, which is somewhat limited under *FastPak*.** Java applications using JNI have limited use in *FastPak* because redundant loads of the JNI libraries can be problematic.
- **Requirement 9: A *FastPak* friendly application is bug free and is not known to throw exceptions or errors except in rare cases.** An error prone application can cause problems with *FastPak* and/or other applications running underneath it.

***FastPak* Hostile Application** – A *FastPak* hostile program, is quite simply put, one that fails to conform to one or more of the application characteristics identified in items 1 through 9 above. It should be noted that a *FastPak* hostile application is not necessarily one that will fail to run under *FastPak*, but it might, and in some cases definitely will, cause *FastPak* to behave in odd or unpredictable manners. For example, in item 4 above, if a developer is using a very high speed machine with lots of memory, an application may appear problem free on the developers machine only to rear its ugly head on a users system using much slower hardware and/or less memory. Several of the items identified in items 1 through 9 above are clearly not *FastPak* specific items, but simply good design and development practices (item 9 should certainly stand out as one).

With that said, each of the items will now be addressed in detail.

Requirement 1 essentially says that a *FastPak* friendly program is not *FastPak* specific. There is, in fact, no such thing as a *FastPak* specific program. If a developer writes an application that is *FastPak* friendly, the application will also run outside of a *FastPak* environment. The purpose of *FastPak* is not to overthrow or dominate a working Java environment, or force users into developing applications that are of value only when *FastPak* is used to run them, it is to co-exist in that environment with whatever other tools developers and end users alike already have at their disposal.

Requirement 2 says quite simply that applications running under *FastPak* are completely isolated from one another. For example, suppose we have a program called “counter.jar” that opens up a GUI and every second updates it's display to show how many seconds it's been since the program was started. Suppose we take this program and configure it for use under *FastPak*. Suppose we start *FastPak* and then launch one instance of “counter.jar” and 20 seconds later launch another instance of “counter.jar.” What this requirement says is that the second launched version of “counter.jar” does not share or use any of the data specific to the instance of “counter.jar” that preceded it by 20 seconds.

As a final note on this topic, an application that's run under *FastPak* can use

synchronization and locking methods as they would with any other Java application, and **FastPak** should not interfere with them. However, such mechanisms cannot be propagated to other applications running under **FastPak**. A user cannot put a lock on an object and expect it to be respected by other applications launched by **FastPak**.

Requirement 3 involves the termination of applications as well as **FastPak**. Under **FastPak**, an application should not terminate itself abruptly by the direct or indirect use of calls like `System.exit(N)`, `Runtime.exit(N)`, or `Runtime.halt(N)` where N is a return code. All of these methods or their indirect manifestations cause the JVM to terminate, and when the JVM terminates, so does **FastPak** and all applications running underneath it. Remember, **FastPak** does not bring up a new virtual machine for each application launched, it uses a single virtual machine for all applications.

The use of methods that shut down the JVM to terminate an application is most prevalent in GUI applications, but by no means limited to them. Many developers think this is normal, but the fact is that terminating an application in this manner is a technique used to circumvent a bug that existed in the JVM itself on releases prior to 1.4. The bug ID was **4030718**. Complete documentation for this bug fix may be found under the Java API documentation under the heading *AWT Threading Issues*. As of the writing of this document (December, 2006) this appears in the documentation for Java releases 1.4, 1.5, and 1.6 under the heading *Implementation- dependent behavior*.

A major problem with the old implementation was that aside from killing the threads associated with the GUI and the JVM, it also killed all threads that might be in the process of doing actual work. For example, suppose a user had an application that was supposed to back up a database. Suppose the user wanted to bring up a GUI front end, fill it out with information regarding which database to back up and where the data was to be stored, and then press an “OK” button to terminate the GUI and launch the backup process. The backup threads are non- GUI threads. In this case, the application developer cannot use the `System.exit()` or `Runtime.exit()` calls when the GUI is to be disposed of because they will destroy the threads that do all the the backup work. What the developer must do in this case is as follows:

1. Create the GUI
2. When the GUI is filled out and “OK” is pressed, launch the threads to do the backup
3. Dispose of the GUI
4. Wait for the threads to complete their backup process.
5. Issue a `System.exit()` or `Runtime.exit()` (or equivalent) to terminate the helper threads and allow the application to terminate.

With the new JDKs (1.4 and higher) some of the steps above can be removed. Now the process above looks like this:

1. Create the GUI

2. When the GUI is filled out and “OK” is pressed, launch the threads to do the backup
3. Dispose of the GUI

As one can see, steps 4 and 5 have been eliminated. The GUI can now be eliminated by using the dispose() call, which will freely destroy the GUI and yet leave the other threads running. The application will terminate “naturally” when all the database backup threads have completed their tasks, unless, for some reason, the developer of the application has implemented some types of tactics to keep the threads alive.

In many cases, the process of making an application “**FastPak** friendly” in this regard is not complicated. Some examples will clarify this. All example source code is included in the “Developer” subdirectory under the “lib” subdirectory **FastPak's** home.

Example 1 – Modifying a simple, GUI based HelloWorld.java program for use with FastPak. The following screen capture illustrates a simple “HelloWorld” type program that has been modified to make it “**FastPak** friendly:”

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5
6 public class HelloWorld {
7     public static void main(String[] args) {
8         JFrame frame = new HelloWorldFrame();
9         frame.show();
10    }
11 }
12
13
14 class HelloWorldFrame extends JFrame {
15     public HelloWorldFrame() {
16         setTitle("Hello, World!");
17         setSize(300, 300);
18         addWindowListener(new WindowAdapter() {
19             public void windowClosing(WindowEvent e) {
20                 //System.exit(0); <----- Eliminated for FastPak "friendliness"
21                 dispose(); // <----- Added for FastPak "friendliness"
22             }
23         });
24
25         Container pane = getContentPane();
26         pane.add(new HelloWorldPanel());
27     }
28 }
29
30 class HelloWorldPanel extends JPanel {
31     public void paintComponent(Graphics g) {
32         super.paintComponent(g);
33         g.drawString("Hello, World!", 110, 130);
34     }
35 }
36
37
```

Figure 1. A simple “HelloWorld” application modified for “*FastPak* friendliness.”

The example above is about as basic as it gets. The application creates a JFrame which in turn adds a WindowListener, and then adds a new JPanel that writes out the string “Hello World!” to the JPanel. The critical changes that make this application “*FastPak* friendly” can be seen in lines 20 and 21. The original code in line 20 has been commented out, and replaced with a single call to dispose(). On Java releases preceding version 1.4, this application will never terminate unless line 21 is commented out and line 20 is uncommented. On Java releases of 1.4 and greater, this problem will work fine as illustrated above. In the latter case, there is no need to “blow up” the JVM via a System.exit() or Runtime.exit() call. The code for the friendly and hostile versions of this application may be found under the Developers library under the directory titled HelloWorld. In this directory there are two subdirectories called “friendly” and “hostile” which correspond to the friendly and hostile versions of the application respectively. All example code will also eventually be placed on the **SCSC** web site under the downloads section. The **SCSC** web site is: <http://www.scsc-online.com>.

Example 2 – Modifying a Multi- Windowed Application to be *FastPak* Friendly.

The preceding example is extraordinarily simple. To illustrate a much more complicated example a program called “FPWindowTester” was created that shows both the new, *FastPak* friendly coding techniques as well as the original *FastPak* hostile code. The source code may be found under *FastPak's* “lib/Developer” directory under the subdirectory titled “FPWindowTester” with the code for the friendly and hostile versions of the applications under directories “friendly” and “hostile” respectively.

Both of these versions have two class files that comprise the application named FPWindowTester.java and FPFrame.java. This program initially creates a window (JFrame), which has under its “File” menu three options that allow the application to create another window with identical functionality, delete itself, or delete all windows and terminate the program. Each new window subsequently opened has the same set of capabilities as that of the original. Readers should either open up some editing windows and review the code as this analysis proceeds or print them out prior to proceeding.

The following two code listings show the source code for the hostile versions of FPWindowTester. Since the changes needed to make the code *FastPak* friendly are relatively small compared to the size of the original application, the full source code for the friendly version will not be included in this text. Readers are asked to either read the actual source code from an editor or print it out and read it as the discussion progresses.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FPWindowTester extends WindowAdapter {
    public int windowCount = 0;
    private Point lastLocation = null;
```

```

private int maxX = 500;
private int maxY = 500;

//Constructor initializes and creates first instance of the window
public FPWindowTester() {
    Dimension screenSize =Toolkit.getDefaultToolkit().getScreenSize();
    maxX = screenSize.width - 50;
    maxY = screenSize.height - 50;
    makeNewWindow();
}

//This method makes a new window when activated
public void makeNewWindow() {
    JFrame frame = new FPFrame(this);
    windowCount++;
    if (lastLocation != null) {
        //Move the window over and down 40 pixels.
        lastLocation.translate(40, 40);
        if ((lastLocation.x > maxX) || (lastLocation.y > maxY)) {
            lastLocation.setLocation(0, 0);
        }
        frame.setLocation(lastLocation);
    } else {
        lastLocation = frame.getLocation();
    }
    frame.setVisible(true);
}

//This is the old way of quitting, which is to terminate the JVM
public void quit(JFrame frame) {
    if (quitConfirmed(frame)) {
        System.exit(0);
    }
}

//Dialog to confirm an operation to close all windows
private boolean quitConfirmed(JFrame frame) {
    String quitButton = "Quit";
    String cancelButton = "Cancel";
    Object[] options = {quitButton, cancelButton};
    int n = JOptionPane.showOptionDialog(frame,
        "Press \"Quit\" to close all windows and exit," +
        "\n or \"Cancel\" to Contine",
        "Close All Windows?",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        quitButton);
    if (n == JOptionPane.YES_OPTION) {

```

```

        return true;
    } else {
        return false;
    }
}

public void windowClosed(WindowEvent e) {
    windowCount--;
    if (windowCount <= 0) {
        System.exit(0); // <----Required for JVMs prior to 1.4
    }
}

//The main method
public static void main(String[] args) {
    FPWindowTester framework = new FPWindowTester();
}
}

```

Listing 1. Source code for “FastPak hostile” version of FPWindowTester.java

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class FPFrame extends JFrame {
    private Dimension size = new Dimension(300, 300);
    private FPWindowTester fpWindowTester = null;

    public FPFrame(FPWindowTester handle) {
        super("FPWindowTester Frame");
        fpWindowTester = handle;
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        addWindowListener(fpWindowTester);

        JMenu menu = new JMenu("File");
        JMenuItem item = null;

        //Action listener for closing a window
        item = new JMenuItem("Close This Window");
        item.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                FPFrame.this.setVisible(false);
            }
        });
    }
}

```

```

        FPFrame.this.dispose();
    }
});

menu.add(item);

//Add a new window
item = new JMenuItem("Make New Window");
item.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        fpWindowTester.makeNewWindow();
    }
});
menu.add(item);

//Terminate program
item = new JMenuItem("Close All Windows");
item.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        fpWindowTester.quit(FPFrame.this);
    }
});
menu.add(item);

JMenuBar menuBar = new JMenuBar();
menuBar.add(menu);
setJMenuBar(menuBar);
setSize(size);
}
}

```

Listing 2. Source code for “FastPak hostile” version of FPFramejava

The hostile version of the program in the file FPWindowTester.java (which also contains the main() method), initially calls its constructor, which in turn sets up some sizing parameters and then immediately calls the method makeNewWindow(). This method creates a JFrame by calling the constructor to the class FPFrame, contained in the source code file of the same name. FPFrame's constructor receives a handle to FPWindowTester itself. This is needed because this class will need to use some of that class's methods. FPFrame then contains the following code:

```

        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        addWindowListener(fpWindowTester);

```

This code tells the class that if the window is closed when a user clicks on the window control that close that specific window, instead of exiting the program, it should dispose of the window. This is necessary in this application, even though it's **FastPak** hostile, because had the DISPOSE_ON_CLOSE option been replaced with EXIT_ON_CLOSE then every time a window was closed the program would terminate. This is unacceptable behavior even with this application.

The rest of the code in FPFrame.java creates the JMenu item "File", and the pull down items and the actionListeners associated with the activities of closing itself, adding a new window, or destroying all existing windows and exiting the program. The important aspects of the program are the closing of the window itself and the termination of the program. About the only notable thing about the option to create a new window is that it uses the method makeNewWindow(), which will increment the window count. Aside from that focus will be on the options to close the existing window and close all windows and terminate the program.

The lines in the ActionListener associated with the "Close This Window" option have the following code:

```
public void actionPerformed(ActionEvent e) {
    FPFrame.this.setVisible(false);
    FPFrame.this.dispose();
}
```

This code tells the window that if a user closes the window manually via the menu item, it sets it's visibility to false (to make it "invisible") and then disposes of it. It should be noted that this code is needed because this is not the same thing as terminating the window by clicking on the windows own control to close the window as discussed in the preceding paragraph.

The lines in the ActionListener associated with the "Close All Windows" option have the following code:

```
public void actionPerformed(ActionEvent e) {
    fpWindowTester.quit(FPFrame.this);
}
```

This code launches the method quit() in the FPWindowTester class. If readers look at this class's source code, they will see it terminates the program by first calling a dialog for confirmation and if confirmed, it calls System.exit(0). This is one of the places that makes this code **FastPak** hostile, since System.exit() calls will take down the JVM, and hence **FastPak** itself. This method does not actually close windows as one might expect, it simply destroys the entire JVM.

Another event that hasn't been addressed so far is what will happen when, instead of terminating all windows as described in the previous paragraph, a user closes all windows manually. For example, say a user opens three windows and then closes them, one by one, by either selecting the "Close This Window" option from the menu, or by closing the window via it's associated window control. Since both of these options call the dispose() operation either directly (for the menu item) or via that associated with the DISPOSE_ON_CLOSE operation, for Java releases prior to 1.4, this leaves only one option when all the windows are disposed of: the program must, in some way, terminate the JVM. It does this is the following code section of FPWindowTester.java:

```
public void windowClosed(WindowEvent e) {
    windowCount--;
    if (windowCount <= 0) {
        System.exit(0); // <----Required for JVMs prior to 1.4
    }
}
```

}

This method gets called every time a window is closed. `System.exit(0)` is only called when the number of window is zero or less, meaning they've all been disposed of. Looking back at the `FPWindowTester` method `makeNewWindow()` we see that each time the window is created a window counter is incremented. When a window is closed, this method decrements that count. If that count is less than or equal to zero, a `System.exit(0)` is issued and the JVM shuts down. On JVMs prior to 1.4, this is needed otherwise the JVM would remain active. This is also a characteristic of a **FastPak** unfriendly program because it will, once again, shut down the JVM and hence terminate **FastPak** and all other programs it may have running underneath it.

As stated previously, for a program to be classified as **FastPak** friendly, it must be devoid of method calls that shut down the JVM. In the hostile version, this occurs when one of the existing windows has the option to close all the windows and terminate the program and when a user manually closes all the windows and there are none left, hence there is no reason for the program to keep running.

To address these issues, the **FastPak** friendly version of the program must do the following:

1. Track the creation of each window, and if and when a specific window is destroyed, the window must not only be disposed of but it must be removed from the tracking mechanism.
2. In the event that a user decides to simply close all existing windows and terminate the program, all existing window must be destroyed and removed from the tracking mechanism, and the program must terminate “naturally,” meaning it will see no more threads requiring servicing and hence terminate.

It may sound complicated, but this is actually done fairly easily. If the reader opens up and reads the source code associated with the **FastPak** friendly version of the program, the following paragraphs will illustrate how this can be done.

The first change made is the addition of a `Vector` class to the file `FPWindowTester.java` file. The `Vector`, called “windowVector” is used to store all active references to existing windows. If a window is destroyed, its associated element in the vector is removed and the vector is “trimmed” to its actual length. If the option to destroy all windows is selected by a user, then the vector will be used to cycle through and sequentially destroy all existing windows. When this is done, all threads associated with the application will be gone, and since this is JVM 1.4 (or greater) specific, the application will terminate naturally. The introduction of the vector is shown in the source code below:

```
//FastPak mod - Added for window index/Window storage
public Vector windowVector;
public FPWindowTester() {
    //FastPak mod below -Used for reference storage
    windowVector = new Vector(0, 1);
    Dimension screenSize =Toolkit.getDefaultToolkit().getScreenSize();
    maxX = screenSize.width - 50;
```

```

    maxY = screenSize.height - 50;
    makeNewWindow();
}

```

Every time a new window is created, it must be added at the tail end of the vector. This is done in the following code section of the makeNewWindow() method in FPWindowTester.java:

```

public void makeNewWindow() {
    FPFrame frame = new FPFrame(this);
    windowCount++;
    //Added for Window handling
    windowVector.add((Object)frame);
}

```

Every time a window is closed (via a dispose operation) its reference in the vector must be eliminated. This requires the introduction of a windowClosing() method and modification of the existing windowClosed() method, both in the file FPWindowTester.java. The windowClosing() method is added to extract the window handle from the closing window, remove it from the vector, and then trim the vector so its size reflects the actual number of active windows still in the vector. The windowClosed() method now has the System.exit(0) call completely removed and simply decrements window counter. The code for both of these is shown below:

```

public void windowClosing(WindowEvent e) {
    JFrame window = (JFrame)e.getWindow();
    windowVector.remove((Object>window);
    windowVector.trimToSize();
}

public void windowClosed(WindowEvent e) {
    windowCount--;
}

```

The final change made is to the quit() method. This method used to be nothing more than a method that interrogated a dialog for confirmation that this was really a desired option followed by a System.exit(0) call that shut the JVM down. In the **FastPak** friendly version, instead of simply and blindly killing every window that exists, they are accessed, one by one from the vector, and then sequentially disposed of. When this operation is done (on JVMs 1.4 and greater) all threads in the application are seen as being complete so the application terminates naturally. The code for this new version of the quit() method is shown below:

```

public void quit(JFrame frame) {
    if (quitConfirmed(frame)) {
        windowVector.trimToSize();
        for(int i = 0; i < windowVector.size(); i++) {
            JFrame toDispose = (JFrame>windowVector.get(i);
            toDispose.dispose();
        }
    }
}

```

```
}  
}
```

By making the changes described above the program is now a **FastPak** friendly program. It will not terminate the JVM, **FastPak**, or any other applications running under **FastPak** when the program is terminated.

As a final note regarding this characteristic of a **FastPak** friendly program, developers must be aware of the ways a `System.exit()`, `Runtime.exit()`, or `Runtime.halt()` method may manifest themselves and be hidden from developers. This is particularly important with GUI programming where the impact of a setting such as `DISPOSE_ON_CLOSE` and `EXIT_ON_CLOSE` will have radically different effects on **FastPak** when it's executing. Additionally, developers need to be aware of possible exit-type calls that may be hidden in methods or libraries they may be using.

Requirements 4 and 5 both involve the EDT. Some applications when run under their own standalone JVM can successfully block the EDT for their own use. In such an application the effects of blocking may or may not be visible to a user, and if they are visible, the developers of the application may feel the negative effects of blocking are little more than an inconvenience and hence ignore it.

For example, suppose a developer writes an application where the user presses a button and another window pops up. The pop up window could be some type of intensive graphics processing window (as in Requirement 4) or perhaps it could be a dialog that establishes a network connection to another machine and doesn't disappear until the connection is established (as in Requirement 5). In either case, if the window that comes up blocks the EDT and the user takes his mouse and drags the new window over part of the GUI that launched it (or any other GUI components that may already be present), as the window is moved over the other parts of the application, when the window is removed, instead of seeing components restore themselves to their original appearance, they just leave "grayed out" areas. Although this may not look good, some developers may deem it acceptable because the application may be developed in such a way that processing in the parent windows of the pop up window should not proceed until the pop up window is complete. Although this is acceptable for such an application running under its own dedicated JVM, the effects can be extremely detrimental under **FastPak**.

This document addresses **FastPak** running under JVMs 1.4 and 1.5. Using either of these JVMs, the EDT under **FastPak** is not a thread dedicated to a single application but rather a shared resource. All applications that need to use the EDT will use only the single EDT supplied by the JVM. As a result, if several Java applications are running under **FastPak** and one of them blocks the EDT, any other applications that need to use the EDT will block as well as long as the EDT is tied up. This means that their graphics processing will freeze until the blocking activity is complete.

With respect to Requirement 4, this usually only occurs when high speed graphics (such as those shown in the demo application Analyzer) are involved. The best way

to overcome this problem is to generate all code that assembles images outside of methods such as paint(), repaint(), update(), paintComponent(), etc. For example, a portion of the source code for Analyzer is shown below:

```

    public void paintComponent(Graphics g) {
        .
        .
        g.drawImage(bufferImage, 0, 0, this);
        paintCounter++;
        .
        .
    }

    public synchronized void run(){
        int frameCounter = 0;
        .
        .

        while(true) {
            .
            .
            bufferGraphics.clearRect(0, 0, bufferWidth, bufferHeight);
            paintBuffer(bufferGraphics);
        }
        repaint();
        .
        .
    }

```

In the code above, counter values and general management code has been eliminated. The drawn image shown on the screen of Analyzer is generated in the run() method when it makes the call paintBuffer(). The method paintBuffer() , which is not shown in the code example. Is a large and relatively time consuming method that interrogates the data arrays used to store the image data, creates the grid overlays, draws the image onto a buffer – in short it does just about everything but actually display the image. The important feature of paintBuffer() is that it is executed in the run() method, hence it's not on the EDT. Keep in mind, the thread associated with the run() method and the EDT are two completely separate threads running under the JVM. The only code that occurs in the EDT occurs when repaint() is called, which in turn calls paintComponent(). All paintComponent() does is take the image created by paintBuffer() and display it on screen.

In contrast, the following code uses the example above but in this case it's been rearranged so that the EDT is not only loaded, it's very heavily loaded:

```

    public void paintComponent(Graphics g) {
        .
        .
        bufferGraphics.clearRect(0, 0, bufferWidth, bufferHeight);
        paintBuffer(bufferGraphics);
    }

```

```

        g.drawImage(bufferImage, 0, 0, this);
        paintCounter++;
        .
        .
    }

    public synchronized void run() {
        int frameCounter = 0;

        while(true) {
            .
            .
            repaint();
            .
            .
        }
    }
}

```

In the example above, the extremely time consuming method `paintBuffer()` has been moved out of the `run ()` method (which is not on the EDT) and into the `paintComponent()` method, which is on the EDT. This means that all the time used to generate the image is now done on the EDT before it gets drawn. The end result, as far as **FastPak** is concerned, is a general degradation of the graphics performance of all other applications running under **FastPak** simultaneously.

With respect to Requirement 5, it's recommended that readers visit web sites or obtain documentation regarding `SwingWorker` and other related topics that involve unloading the EDT. This is really nothing more than good programming practice.

Unloading the EDT is not really a **FastPak** specific topic, since it's considered good Java design to do so. However, there are some caveats that **FastPak** developers and users should be aware of with respect to EDT. Problems that are worth making note of are identified in the following list:

- Graphics performance can vary drastically, very, very drastically, with operating system, processor speed, Java version, video card speed and memory, and system memory variations. This is, once again, not really a **FastPak** specific issue, but using **FastPak** may very well make potential problems more noticeable.
- With the above comments in mind, **FastPak** and any applications run underneath it should be tested on a range of machines that replicates the expected user base. Developers frequently have very “high end” machines that can mask potential end user problems.

Requirement 6 involves the use of applications using multiple `main()` methods (Yes! It is possible to do that in Java!). Applications of this type appear to be very, very rare, and the fact is **SCSC** simply hasn't even bothered testing this, thus **SCSC** won't stand

behind any use or attempts to use such applications.

Requirement 7 states that an application doesn't deliberately attempt to kill threads. This requirement relates primarily to very, very, old legacy applications that typically require either an old JVM or that a JVM be launched with flags forcing it to a previous version. **FastPak** does not support this activity by virtue of the fact that it's restricted to JVMs, JREs, and JDKs of 1.4 or higher. Additionally, users attempting to kill threads deliberately using extraordinary means may very well interfere, disrupt, or halt **FastPak** and/or any applications running under it. This will most often mean application making use of such classes as ThreadDeath. This is generally considered abnormal development.

Requirement 8 deals with applications using JNI libraries. Generally, if an application uses a specific JNI library, when that application is loaded under **FastPak**, the JNI library is typically loaded as well. If a second instance of the application is loaded an error will typically occur because the second instance will be attempting to load a library that's already been loaded. There are ways to circumvent this, but they are relatively complicated and doing so violates Requirement 1 for **FastPak** friendliness as stated above, thus they will not even be addressed in this document and they will not be supported by **SCSC**. On some platforms, users may find that if an application is launched, used, and then terminated, once the remnants of the application have been garbage collected the application will be usable again. This type of processing may be operating system, hardware, and JRE release specific and can't be guaranteed to work on all platforms. It also requires the end user to monitor the GUI or log files for a message that the original JNI application has been terminated and garbage collected, which can yield unpredictable results and potentially be quite time consuming. Multiple applications using differing JNI libraries can be launched provided each application has it's own unique library and the libraries don't include or reference other libraries that may result in a redundant JNI load.

Requirement 9 simply states that an application targeted for use with **FastPak** should not be known to throw exceptions or errors during its execution and should be well tested and as bug free as possible. When an application is loaded in **FastPak**, that application is not seen by **FastPak** as a unique application, it is seen as an integrated extension of the **FastPak** kernel. As a result, **FastPak** will see any exceptions and errors thrown by launched applications as internally generated, and the **FastPak** kernel must decide whether to try and handle them or propagate them back to the original application. In most cases the kernel properly propagates the exception back to the program and it will show up on the GUI screen or the log files as a thrown exception. However, in the rare case that the exception thrown just happens to occur in a region of the **FastPak** kernel where the kernel itself may be testing for such an exception, then the kernel may catch and erroneously process the exception. The results in this case are typically a **FastPak** lock-up. Applications that throw Errors will almost invariably bring **FastPak** down.

It's recognized that not all Java applications a user or developer may come across will be provided with source code, and thus it may be necessary to test the applications for **FastPak** compatibility. The following is a list of pointers to help users and

developers determine if an application is **FastPak** friendly:

Run two or more instances of the same application with the exact same startup parameters, which really means launch two or more instances from the same application profile. Users should, assuming the applications launch properly use the applications to their fullest extent by applying the same execution steps to all instances of the application, but use differing data when possible. Users should watch out for the following **FastPak** hostile characteristics:

1. The second launch of the application immediately starts showing logging errors about loading a library that's already loaded. This is an indication that the application uses JNI and only one instance of it can be run at a time. Depending on the hardware, operating system, and Java version, it may only be able to be run once during a **FastPak** session. See the section on Requirement 8 for details.
2. When one of the instances is terminated, all applications and **FastPak** itself terminate. This is an indication that the application is making direct or indirect use of `System.exit()`, `Runtime.exit()`, or `Runtime.halt()`. See the sections on Requirement 3 for details.
3. If the application is a GUI application and the opening and closing of the GUI components causes underlying components to be “grayed out” once exposed, either the EDT is being blocked by one or more of the applications or the number of applications running is simply too much for the hardware/OS combination to deal with. The latter will be seen on older operating systems and hardware configurations, and in such a case there may be nothing that can be done about the problem since it will likely exist with or without **FastPak**. See the sections on Requirements 4 and 5 for details.
4. **FastPak** or some of the applications “lock up” unexpectedly. This can be caused by throwing exceptions or errors, or the program could be attempting (and succeeding) in killing threads using abnormal programming tactics. See the sections on Requirements 7 and 9 for details.

In closing, it should be stated that an application that is not **FastPak** friendly should not be necessarily be excluded from **FastPak** use, it simply depends on how the applications are used. For example, there is a user that launches five applications with **FastPak's** auto-launch mode, keeps them up all day, and then terminates them by simply closing one of them. The applications in this case violate Requirements 2, 3, and 8, and yet because the set of launched applications are unique, the user is able to use them throughout the day, and then terminate all of them by closing a single one of them (they all violate Requirement 3). The user in this case could care less about **FastPak** friendliness. With the exceptions of Requirements 6, 7, and 9, **FastPak** friendliness is not a requirement, but rather a **strong recommendation** regarding the design of applications where **FastPak** will be used.

The **FastPak** Controller Thread

The **FastPak** Controller Thread allows an instance of **FastPak** to be remotely controlled by another application and allows the use of a custom GUI to replace the **FastPak** GUI. Configuration and administration of the *Controller Thread* is covered primarily in the **FastPak for Java Advanced Users Guide** as well as the **FastPak for Java Users Guide**. This section will not focus on administrative issues but rather give a high level overview of this thread before addressing the **FastPak** API itself.

The *Controller Thread* is a thread that, if started, runs in parallel to the **FastPak** kernel. This thread makes use of the **FastPak** API (discussed in the next section) which allows a user to either remotely or locally control a running instance of **FastPak** via a sockets interface. If a **FastPak** instance is run in daemon mode, users can use the **FastPak** API and the *Controller Thread* to create their own, custom GUI for use with **FastPak**. The *Controller Thread* essentially waits on a server socket for incoming messages, and when received, assuming the message(s) adhere to the **FastPak** protocol and are error free, formats the command and issues a request for processing to the **FastPak** kernel. The kernel processes the command and sends a message back to the *Controller Thread*, which then puts the message in a text format and sends it back to the application that sent it.

There is no requirement that the application that is going to communicate with **FastPak** via sockets and the *Controller Thread* be written in Java. However, the following requirements must be met for all applications to communicate with the *Controller Thread*, regardless of what programming language is used:

1. All applications communicating with the *Controller Thread* will be using client side TCP/IP sockets.
2. All messages that the *Controller Thread* processes are using ISO-8859-1. If an application is developed using Java and the **FastPak** API library, this will be hidden from the application.
3. All applications should send a request to **FastPak** and then wait for a response before proceeding with more requests.
4. Communications with the *Controller Thread* are inherently slow, and are limited strictly to the **FastPak** protocol. This is a necessity because **FastPak**, which is intended to run many applications simultaneously, including networked applications independent of the **FastPak** protocol, can not have its internal operation suspended or interrupted waiting for unnecessarily long data transfers associated with the *Controller Thread*. This is the way **FastPak** was deliberately designed.

The **FastPak** API is small and very easy to use if a developer is programming the *Controller Thread* using Java.

The *FastPak* API

The ***FastPak*** API is a small API and associated library (in jar format) that allows a developer to communicate with an instance of ***FastPak*** that has its *Controller Thread* running. The method `FastPakAPI(int, int, int, String)` creates the `FastPakAPI` object. All other methods, open a socket, send a command to the *Controller Thread*, obtain the response from the *Controller Thread* as a `String`, and close the socket. All responses are a single `String`, not arrays, but the response may be formatted with '\n' embedded throughout the `String` for multi line appearance. The methods that comprise the `FastPak` API are as follows:

Method name: `public FastPakAPI(int retry, int time, int port, String host)`

Method Parameters:

`int retry` – The number of times to retry establishing a link with a *Controller Thread* before giving up

`int time` – Maximum amount of time allocated for a retry in milliseconds.

`int Port` – The network port number assigned to the instance of `FastPak` being accessed.

`String Host` – The host name of the machine hosting the instance of ***FastPak*** being accessed.

Description:

This method instantiates the `FastPakAPI` object. It only needs to be called once unless it's assigned a null value and/or the hosting class or thread has terminated. It will initialize the underlying and hidden socket interface used to communicate with the *Controller Thread* as specified by the host name and the port ID. Each instance of ***FastPak*** on a machine must have a unique port ID. This is a public constructor and there are no return values from this method. All methods identified in this API below must access the object created by the constructor to communicate with the *Controller Thread*.

Example:

```
FastPakAPI fpa = new FastPakAPI(10, 1000, 20001, "Zebra");
```

This example creates a `FastPakAPI` object called "fpa" with a retry counter set at 10 attempts before failure, a timeout value of 1000 milliseconds, a port number set at 20001, and the machine hosting the ***FastPak*** has a host name of "Zebra."

Response:

There is no response for this constructor. It simply loads the parameters needed for communications with the *Controller Thread*.

NOTE: All methods described in the following method descriptions are all part of the FastPakAPI class, hence they will always be referenced to an existing FastPakAPI object. In the example above, this object is called "fpa." All of the example code provided in the following method descriptions will use the "fpa" and this can be changed as developers see fit.

Method name: public String runApp(String *appProfile*)

Method Parameters: String *appProfile*

Description: This method launches an application on a **FastPak** instance as defined by the FastPakAPI object. The String *appProfile* is the name of an application profile that exists on the targeted **FastPak** instance. This method will use the applications default profile settings with no command line parameter modifications of any sort.

Example: String retValue = fpa.runApp("ActiveLogo");

In this example, the profile called ActiveLogo is launched on the targeted instance of **FastPak**. The application will run with no additional, modified, or replaced command line parameters.

Response:

On success, the following String is returned:

---- Requested operation succeeded -----

On failure, a host of messages can be received, but the most common is as follows:

An error has occurred in the FastPak kernel.

The following errors have been reported:

******* FastPak ERROR detected *******

LOCATION: FastPakKernel

The application you attempted to load under FastPak could not be found.

The most probable causes for this are as follows:

- 1. The class path is not set correctly for FastPak**
- 2. The file name is incorrectly spelled.**
- 3. The application is in a package and is inappropriately prepended.**
- 4. The file has been erased.**
- 5. The path specified to the file is incorrect.**
- 6. The properties file associated with it has been deleted.**
- 7. The properties file incorrectly identifies the application.**

Please verify all of the above are correct and review the FastPak Users

Manual for further information.

Method name: public String runAppNewInput(String *appProfile*,
String *newInputParams*)

Method Parameters:
String *appProfile*
String *newInputParams*

Description:
This method launches the application associated with the profile *appProfile* with a new set of command line parameters. The new command line parameters are placed in the single String object *newInputParams*. The new input parameters completely replace any existing command line parameters that exist in the stored profile for that application launch only.

Example:
String retValue = fpa.runAppNewInputParameters("CarPics",
"imagePath=/U1/bob/FastPak/Data/SolarPics height=400 length=400")
;

In the example above, the command line parameters associated with the profile CarPics have been completely overridden so they now point at the directory associated with that of SolarPics, and in addition the height and length of the image displayed have been resized from non-existent settings (default settings in this case) to provide a display of 400 by 400 pixels.

Response:
On success, the following String is returned:
----- **Requested operation succeeded** -----

On failure, there may be a response, and there may not be, depending on the error. If the name of the application profile submitted is in error or it's associated binaries have been removed, renamed, etc. then FastPak will return a set of errors similar to those shown under the method description for runApp() above. If the parameters sent to the program are incorrect, then FastPak will generally not send a response since it has no way of internally interrogating how an application will respond to errors the application itself sees. If logging is on, any output data from the application will be dumped to the log files on the targeted **FastPak** instance.

Method name: public String runAppAppendedInput(String *appProfile*, String *appendedInputParams*)
String *appProfile*
String *appendedInputParams*;

Description:

This method takes the application profile defined by the variable *appProfile* and then appends the command line parameters defined in that profile (if any) with those contained in the String *appendedInputParams* for that application launch only.

Examples:

```
String retValue = fpa.runAppAppendedInput("ActiveLogo", "768");
```

In the example above, the application defined by the application profile ActiveLogo, which has no command line parameters defined in its profile, will now execute with the command line parameter of "768" which tells it to override its default setting of 600. This same task could have been done using the previously described method.

```
String retValue = fpa.runAppAppendedInputParameters(  
    "CarPics", 'height=400 length=400" );
```

In this case, the profile CarPics, which has a single command line parameter defining the directory where the car pictures are located, has, in addition, parameters that set the height and width of the image to 400 by 400 pixels, hence overriding its default settings.

Response:

On success, the following String is returned:

```
----- Requested operation succeeded -----
```

On failure, there may be a response, and there may not be, depending on the error. If the name of the application profile submitted is in error or it's associated binaries have been removed, renamed, etc. then **FastPak** will return a set of errors similar to those shown under the method description for runApp() above. If the parameters sent to the program are incorrect, then **FastPak** will generally not send a response since it has no way of internally interrogating how an application will respond to errors the application itself sees. If logging is on, any output data from the application will be dumped to the log files on the targeted **FastPak** instance.

Method name: public String showAvailable()

Method Parameters:
none

Description: This method displays a list of the application profiles currently defined on the targeted instance of **FastPak**.

Example:
String retValue = fpa.showAvailable();

Response:
This method will return a sentence describing the output followed by a list of the currently available applications as follows.

The following list identifies the applications that are currently configured for use with FastPak:

ActiveLogo
Analyzer
AppletApplication
CarPics
FourierAnalysis
LogoLib
RemoteAnalyzer
RemoteFourierAnalysis
RemoteSCSCLogo
SCSCLogo
SolarPics

Method name: public String activateLogging()

Method Parameters:

none

Description:

This turns on logging to disk on the targeted FastPak instance.

Example:

```
String response = fpa.activateLogging();
```

Response:

This method will return the name of the log file on the targeted **FastPak** instance.

For example:

Logging has been turned on.

Log file: /Users/smith/FastPak/log/log_Sat_02_03_2007-04-25-11.log

Developers need to be a little careful with redundantly calling this method since each time it is called it will close any pre-existing log files and start logging to the newest instance. The logs exist on the targeted **FastPak** instance.

Method name: public String deactivate Logging()

Method Parameters:

none

Description:

This method will deactivate the logging on the targeted FastPak instance and close the logging file.

Example:

```
String response = fpa.deactivateLogging();
```

Response:

FastPak will return a String similar to the following:

Logging has been turned off.

Log file: /Users/smith/FastPak/log/log_Sat_02_03_2007-04-32-56.log
has been closed.

Method name: public String listCurrentConfig()

Method Parameters:

none

Description:

This prints out a list of the configuration for the targeted FastPak instance.

Example:

```
String response = fpa.listCurrentConfig();
```

Response:

FastPak will return a String similar to that shown below:

FastPak is currently using the following settings:

Home directory for FastPak: /Users/smith/FastPak/

Port number for the FastPak controller thread: 20001

Current FastPak controller thread state: controller thread running

Current default user interface mode: Graphical User Interface

Method name: public String listThreads()

Method Parameters:

none

Description:

This method will list the current high level threads under the targeted instance of FastPak. Be aware that some applications have threads that can pop up, do a task, then disappear from this list.

Example:

```
String response = fpa.listThreads();
```

Response:

The response, which can vary from platform to platform, is a list of high level threads but should be somewhat similar to that shown below:

FastPak's currently running threads including FastPak main():

Name: Reference Handler, Group: system, Priority: 10, Daemon: true

Name: Finalizer, Group: system, Priority: 8, Daemon: true

Name: Signal Dispatcher, Group: system, Priority: 9, Daemon: true

Name: Java2D Disposer, Group: system, Priority: 10, Daemon: true

Name: AWT-AppKit, Group: main, Priority: 5, Daemon: true

Name: AWT-Shutdown, Group: main, Priority: 5, Daemon: false

Name: AWT-EventQueue-0, Group: main, Priority: 6, Daemon: false

Name: Thread-1, Group: main, Priority: 5, Daemon: false

Name: Thread-0, Group: main, Priority: 5, Daemon: false

Name: DestroyJavaVM, Group: main, Priority: 5, Daemon: false

Name: TimerQueue, Group: main, Priority: 5, Daemon: true

Name: Thread-4, Group: main, Priority: 5, Daemon: false

Method name: public String reloadApps()

Method Parameters:

none

Description:

This method reloads the application profiles on a targeted FastPak instance. This is useful if new applications and application profiles have been added while FastPak is running.

Example:

```
String response = fpa.reloadApps();
```

Response:

FastPak will return the following String message:

FastPak just reloaded the application configuration files.

Any changes you made should now be in effect.

Method name: public String deactivateApp(String *appProfile*)

Method Parameters:

String *appProfile*;

Description:

This method temporarily deactivates an application profile from being used. This deactivates a specific profile, and if other profiles are making use of an application that need to be deactivated as well, then access to a possibly problematic profile can still exist. If an application is problematic, then all profiles associated it should be removed using this method one by one. This method call does not permanently remove the profile from the list available application profiles, but rather removes it for that session of the targeted **FastPak** instance. A users on using the targeted FastPak instance will immediately find that this profile is no longer available for use.

Example:

```
String response = fpa.deactivateApp("ActiveLogo");
```

Response:

On success, the targeted **FastPak** instance will send the following response:

Delete command processed.

On failure, the only possible cause is that the name of the profile submitted is either incorrect or it's already been removed. In either case, the following message will be displayed:

Your attempt to remove an application from the active applications storage

area has failed because the key you provided has a was not found.

Either this

application has already been removed from the storage area or it was never

placed in the storage area. This operation has been suspended.

Method name: public String activateApp(String *appProfile*)

Method Parameters:

String *appProfile*

Description:

This method can reactivate a previously deactivated application profile or update an application profile that has been manually edited. When application profiles are edited via the **FastPak** GUI they are automatically reloaded, but administrators may find it necessary to manually perform this task at times on some application profiles. If the profile is already loaded, this method goes ahead and reloads it. If the application profile has been manually edited, then the new profile will overwrite the existing profile. If the profile hasn't changed it will still be reloaded, which has no real effect. If the profile had previously been deactivated, this method will reactivate it.

Example:

```
String response = fpa.activateApp("ActiveLogo");
```

Response:

On success, the targeted instance of **FastPak** will send the following message:

Add command processed.

On failure, the only possible cause is that the name of the profile submitted is either incorrect or it's already been removed. In either case, the following message will be displayed:

You attempted to add an application profile to the active applications storage area. The name you provided was not found. This operation has been suspended.

Method name: public String stopController()

Method Parameters:
none

Description:

This method will stop the *Controller Thread* on the targeted instance of **FastPak**. If this method is called, the targeted instance of **FastPak** will send back a response indicating that the *Controller Thread* is being shut down. After that, communications between the application using this call and the targeted instance of **FastPak** will cease, and any other attempts at making API calls to the targeted instance of **FastPak** will send socket timeout errors. The only way at this point to restart the *Controller Thread* on the targeted instance of **FastPak** will be for a user that has access to that instance to manually restart it, but only if they are using GUI or console mode. If the targeted instance of **FastPak** is in daemon mode, then since all communications to that instance have terminated, the only way to stop it from running or reestablish communications will be to use operating system commands to terminate that particular instance. **USE THIS METHOD WITH EXTREME CAUTION!**

Example:

```
String response = fpa.stopController();
```

Response:

The following is the most likely response that will be sent to an application:
The FastPak controller thread is stopping.

Method name: public String fastPakExit()

Method Parameters:
none

Description:

This method will issue an exit call to the targeted instance of **FastPak**. The exit call shuts down the *Controller Thread* on the targeted instance of **FastPak**, terminates user interfaces (**FastPak's** GUI or console modes), and puts the kernel into a state that allows running applications to continue running, with the **FastPak** kernel terminating when they are done. After this command is issued, the application making use of this call should not issue any more calls to the targeted instance of FastPak because Controller Thread termination is not instantaneous and additional calls to the targeted instance of FastPak may cause both the FastPak kernel and/or the application making use of this command to hang. **USE THIS METHOD WITH EXTREME CAUTION!**

Example:

```
String response = fpa.fastPakExit();
```

Response:

The targeted instance of FastPak will respond with the following message:

FastPak EXIT command received.

FastPak will shut down and exit when all applications and threads have finished running.

Method name: public String fastPakShutdown()

Method Parameters:

none

Description:

This method will immediately terminate the targeted instance of **FastPak**, the *Controller Thread*, as well as any applications currently running on that instance. This method is really intended for use in emergency conditions, such as an application running under a targeted instance of **FastPak** that goes into a “tight loop” making things like database calls or network communications that erroneously saturate other systems. When this method is called, the response from the targeted instance of **FastPak** can be erratic, and vary considerably based on the platform and speed of the machine hosting the targeted instance of **FastPak**. **USE THIS METHOD WITH EXTREME CAUTION!**

Example:

```
String response = fpa.fastPakShutdown();
```

Response:

Responses from the targeted instance of FastPak can vary considerably based on the platform and its speed, but the following message is not uncommon:

ERROR: Problems encountered reading response from FastPak.

DATA RECEIVED FOLLOWS:

null

Method name: public String fastPakHelp()

Method Parameters:

none

Description:

This method lists the commands available in the accessible portions of the **FastPak** protocol. This is pure **FastPak** Protocol which is not terribly user friendly. The use of this method is discouraged unless a developer is putting together a console that duplicates that of the **FastPak** console, which is also somewhat user unfriendly. It's recommended that developers review the demonstration application called Clc.java instead of attempting to make use of this call. **This call is not supported!**

Example:

```
String response = fpa.help();
```

Response:

The following is typical of a response from a targeted instance of FastPak, but it is very likely to change:

```
Welcome to FastPak, Version 2.10.55
```

```
FastPak Startup Information:
```

```
Operating System Name: Mac OS X
run <app_name> - Runs an app specified by the key <app_name>
run <app_name> input_params string1 string2 ... stringN
- Provide command line parameters, ignore those
provided in the stored app
run <app_name> append_params string1 string2 ... stringN
- Use the command line parameters in the properties
file, but append these words to it
show_available - Dumps the TreeMap of available apps in TreeMap
logging_on - Turns on the logging with a new log file
logging_off - Turns the logging off and closes the log file
dump_config - Dumps FastPak configuration settings to UI
dump_threads - dumps info about all threads currently running
reload_apps - Re-loads Application configuration property files
deactivate <app_name> - Removes the app specified by the <app_name> key
from the TreeMap
activate <app_name> - Reactivates the app specified by the key
<app_name> that had been deactivated.
start_controller - start the controller, error if it's already up
stop_controller - stop the controller
exit - exit FastPak gracefully (stops when threads end)
shutdown - shutdown FastPak immediately via System.exit(0)
help - show this list
```

The **FastPak** Protocol

The **FastPak** protocol is the command sequence used by both the **FastPak Controller Thread** and the **FastPak** API to control operations on a running instance of **FastPak**. If a user has used **FastPak** in its console mode, then they've already been exposed to the formatting of the protocol, since the console mode uses a duplicate of the protocol to control its associated **FastPak** instance. It should be noted that using the **FastPak** protocol directly without using the **FastPak** API is not supported.

This section is being provided for developers that wish to create an application in a non- Java language, such as C or C++. To do so, however, the application must meet the following requirements:

1. The *Controller Thread* must be running on targeted instances of **FastPak** for the non- Java application to be of any use.
2. The Strings must be formatted using ISO-8859- 1 format

Developers will note that all elements of the **FastPak** protocol have a corresponding method and response as defined in the **FastPak** API (above). Because of this, instead of re- writing the same text, this section will instead define the protocol element by name and identify its corresponding API entry for reference. All API entries are nothing more than wrappers around the **FastPak** protocol.

All commands sent to the **FastPak Controller Thread** are the binary equivalent of a single Java String, and all responses are a single String as well. Buffers sent and received between a targeted **FastPak** instance's *Controller Thread* and the non- Java application must make use data in this format and convert it accordingly.

For the protocol definitions below, all examples will use a < > set to define variable input. Anything without such notation is a mandatory field. For example, looking at the protocol element “run <appProfile> append_params <param1 param2paramN>” below, the protocol command is “run”, <appProfile> would be filled in with the name of an application profile, “input_params” is a necessary modifier to the “run” command to tell the *Controller Thread* to parse for additional parameters after the modifier and accept them as additional parameters to the application defined by the application profile.

Using the example above, if a developer created a type- def byte buffer called ISOStringBuffer, and a function called sendFastPakMsg(char *) that uses that return type, then the code for this implementation may look as follows:

```
.  
. ISOStringBuffer retVal;  
.  
.
```

```

.
retVal = sendFastPakMsg("run CarPics append_params length=400
height=400");
.
.

```

The p-code function `sendFastPakMsg()` issues a “run” command with appended input parameters to the application profile for “CarPics,” with the appended input parameters being “length=400 height=400.”

For a much simpler example, if the protocol element “show_available” is used with the same method above, the code might look like this:

```

.
.
ISOStringBuffer retVal;
.
.
.
retVal = sendFastPakMsg("show_available");
.
.

```

In the examples shown below, the text is what would be put into the function that will be used to communicate with the *Controller Thread*.

Protocol Element: run <appProfile>

Corresponding API entry: String runApp(String appProfile)

Example: “run ActiveLogo”

Protocol Element: run <appProfile> input_params <param1 param2paramN>

Corresponding API entry: String runAppNewInput(String appProfile, String newInputParams)

Example: “run SCSCLogo input_params 480”

Protocol Element: run <appProfile> append_params <param1 param2
.....paramN>

Corresponding API entry: String runAppAppendedInput(String appProfile, String
appendedInputParams)

Example: “run CarPics append_params height=400 length=500”

Protocol Element: show_available

Corresponding API entry: String showAvailable()

Example: “show_available”

Protocol Element: logging_on

Corresponding API entry: String activateLogging()

Example: “logging_on”

Protocol Element: logging_off

Corresponding API entry: String deactivateLogging()

Example: “logging_off”

Protocol Element: dump_config

Corresponding API entry: listCurrentConfig()

Example: “dump_config”

Protocol Element: dump_threads

Corresponding API entry: String listThreads()

Example: “dump_threads”

Protocol Element: reload_apps

Corresponding API entry: String reloadApps()

Example: “reload_apps”

Protocol Element: deactivate <appProfile>

Corresponding API entry: String deactivateApp(String appProfile)

Example: “deactivate SCSCLogo”

Protocol Element: activate <appProfile>

Corresponding API entry: String activateApp(String appProfile)

Example: “activate CarPics”

Protocol Element: stop_controller

Corresponding API entry: String stopController()

Example: “stop_controller”

Protocol Element: exit

Corresponding API entry: String fastPakExit()

Example: “exit”

Protocol Element: shutdown

Corresponding API entry: String fastPakShutdown()

Example: “shutdown”

Protocol Element: help

Corresponding API entry: String help()

Example: “help”

Developing a General Controller for FastPak

Developing a general controller for a targeted instance of **FastPak** is generally straight forward. So much so, in fact, that instead of going through the details, an example will be presented. The source code for the example may be found under the “lib” subdirectory of the **FastPak** base installation directory under its Developer/CommandLineController subdirectory. The the application consists of the file “Clc.java,” which contains the main() method, and “CmdLineController.java” which does most of the work.

Building the applicator is very straight forward:

1. Change to the directory containing the command line controller.
2. Copy or link the FastPakAPI.jar file from the “lib” subdirectory under **FastPak's** base directory to this directory.
3. Compile the application with the following command:

```
javac -classpath .:FastPakAPI.jar Clc.java
```

To test this application after compiling it, do the following using **FastPak** in GUI mode:

1. Start an instance of **FastPak** on the machine.
2. Activate the *Controller Thread* by clicking on the “Control” menu item and then select “Start Controller Thread” option. If an error comes up stating that it's already running, then that instance of **FastPak** already has the thread running.
3. Open up a terminal and switch to the lib/Developer/CommandLineController directory.
4. Launch the command line controller by typing in “java - classpath .:FastPakAPI.jar Clc localhost 20001” This assumes that the default port ID of 20001 hasn't been changed, but if it has then change the number.
5. Follow the menus and launch some applications. Once it's verified the application is running properly, go to step 6.
6. Get a **FastPak** instance running on a remote machine and get the host name or IP address of that machine as well as its port number if these values aren't known.
7. Stop the previously running instance of the command line controller.
8. Restart the command line controller using step 4, but this time substitute the new host name/IP address and the port number.
9. Run some commands using the command line controllers screens. If the remote machine is not visible, then if applications are being launched, have someone contact you to verify the applications are indeed launching on that machine. Some commands that do nothing more than obtain information about the system will not require a second users presence.

The name of the command line controllers primary file is “Clc.java” and the file the does the real work is called CmdLineController.java. The listings of both of these modules are shown below:

```

public class Clc {
    public static void main(String args[]) throws Exception {
        if(args.length < 2) {
            System.out.println("Usage: java Clc hostname port");
            System.exit(0);
        }
        String hostname = args[0];
        int portNum = Integer.parseInt(args[1]);
        CmdLineController clc = new CmdLineController(hostname, portNum);
        clc.start();
    }
}

```

Listing 3. The source code for Clc.java

```

import java.io.*;
import java.lang.*;
import java.text.*;
import com.scsc_online.fastpak.api.*;

public class CmdLineController extends Thread {
    private String commandNumber;
    private String dummy;
    private InputStreamReader isr;
    private BufferedReader br;
    private int option = 0;
    private String hostname;
    private int portNum;
    CmdLineController(String host, int port) {
        hostname = host;
        portNum = port;
    }
    public void run() {
        String appName = null;
        String appParams = null;
        String response = null;
        //set retry at 20, interval at 1 sec (1000 msec)
        int retry = 20;
        int timeout = 1000; //1 sec
        //Create the FastPakAPI instance
        FastPakAPI fpa = new FastPakAPI(retry, timeout, portNum, hostname);
        //Set up command line I/O
        commandNumber = new String();
        dummy = new String(); //discarded input data
        isr=new InputStreamReader(System.in);
        br=new BufferedReader(isr);
        option = 0;
        while (true) { //Start processing network test commands
            System.out.println("\n Enter your FastPak option by number from the list
below: \n");
            System.out.println(" 1 Run a configured app as configured");
            System.out.println(" 2 Run a configured app with new input parameters");
            System.out.println(" 3 Run a configured app with appended input

```

```

parameters.");
    System.out.println(" 4 List configured applications on target machine.");
    System.out.println(" 5 Activate logging on target machine.");
    System.out.println(" 6 Deactivate logging on target machine.");
    System.out.println(" 7 List config on target machine.");
    System.out.println(" 8 List high level active threads on target
machine.");
    System.out.println(" 9 Reload application profiles on target machine.");
    System.out.println(" 10 Deactivate app on target machine.");
    System.out.println(" 11 Activate app on target machine.");
    System.out.println(" 12 Stop controller thread on target machine.");
    System.out.println(" 13 Execute \"exit\" command on target machine.");
    System.out.println(" 14 Execute \"shutdown\" command on target
machine.");
    System.out.println(" 15 Terminate this session.");
    System.out.print("\n Enter the option number you wish to execute: ");
    try {
        commandNumber=br.readLine();
    }
    catch(IOException e) {
        System.out.println("Error reading input!");
    }
    option = Integer.parseInt(commandNumber);
    if((option < 1) || (option > 15)) {
        System.out.println(" ***** Invalid input received. Please use numbers
from 1 through 15!!*****");
        System.out.println(" HIT THE RETURN KEY TO CONTINUE!!");
        try{dummy = br.readLine();}catch(IOException ioe){}
    }
    else {
        switch(option) {
            case 1:
                appName = getAppName("run as configured: ");
                response = fpa.runApp(appName);
                break;
            case 2:
                appName = getAppName("run with new input parameters: ");
                appParams = getAppParams("new");
                response = fpa.runAppNewInput(appName, appParams);
                break;
            case 3:
                appName = getAppName("run with appended input parameters: ");
                appParams = getAppParams("appended");
                response = fpa.runAppAppendedInput(appName, appParams);
                break;
            case 4:
                response = fpa.showAvailable();
                break;
            case 5:
                response = fpa.activateLogging();
                break;
            case 6:
                response = fpa.deactivateLogging();
                break;
            case 7:

```

```

        response = fpa.listCurrentConfig();
        break;
    case 8:
        response = fpa.listThreads();
        break;
    case 9:
        response = fpa.reloadApps();
        break;
    case 10:
        appName = getAppName("deactivate: ");
        response = fpa.deactivateApp(appName);
        break;
    case 11:
        appName = getAppName("activate: ");
        response = fpa.activateApp(appName);
        break;
    case 12:
        response = fpa.stopController();
        break;
    case 13:
        response = fpa.fastPakExit();
        break;
    case 14:
        response = fpa.fastPakShutdown();
        break;
    case 15:
        return;
    }
}
System.out.println(response);
}

private String getAppName(String type) {
    String displayString = " Enter the name of the app. profile you wish to ";
    String retVal = "";
    boolean accepted = false;
    while (!accepted) {
        try {
            System.out.print(displayString + type);
            retVal = br.readLine();
            accepted = true;
        }
        catch (IOException ioe) {
            System.out.println("Input error processing app name, please retry.");
        }
    }
    return retVal;
}

private String getAppParams(String paramType) {
    boolean accepted = false;
    String retVal = "";
    while (!accepted) {
        try {

```

```

        System.out.print("Enter the "+paramType+" input parameters: ");
        retVal = br.readLine();
        retVal=retVal.trim();
        accepted = true;
    }
    catch(IOException ioe) {
        System.out.println("Input error processing argument list, please retry.");
    }
}
return retVal;
}
}

```

Listing 4. The source code for CmdLineController.java

In the listings above, it should be clear to the reader that Clc.java simply starts the main() method and receives some command line parameters, which, after conversion of one to an int, passes then entries to the constructor for CmdLineController and then starts the thread. In the CmcLineController's run() method the FastPakAPI instance called "fpa" is created, and then the program enters an interactive console like session.

Developing a GUI Controller for *FastPak*

If the *FastPak* instance on a machine is put into daemon mode, it requires that a developer provide a suitable interface for the daemon if there is to be any control over it. Although this can be done from a remote machine, this can also be provided as an opportunity for a developer to replace *FastPak's* default GUI with a new, custom interface.

In this section, which will rely heavily on code examples, a GUI is developed that can be run right alongside *FastPak* or underneath *FastPak* as an auto launched daemon process. This example is non sophisticated. As the code will soon reveal, the primary steps in the operation of are as follows:

1. Obtain the hostname and port number from the command line parameters.
2. Launch the class LoadableGUIFrame.
3. Inside the LoadableGUIFrame() method, the FastPakAPI instance is created as fpa.
4. Use the FastPakAPI call showAvailable() to get a listing of the available FastPak profiles.
5. Parse out header Strings from the response and trim them. This is seen in the code snippet:

```
retVal = retVal.replaceAll("The following list identifies the
applications that are currently\n", "");
retVal = retVal.replaceAll(" configured for use with
FastPak:\n", "");
retVal = retVal.trim();
String[] appList = retVal.split("\n"); //create application list
for(int i = 0; i < appList.length; i++) //trim up strings
    appList[i] = appList[i].trim();
```

6. Create a very basic GUI application. In this case it should be noted that the windowClosing method calls System.exit(0). If this instance is run under FastPak, this won't matter since the method also makes an API call to terminated FastPak prior to that, but if it's launched as a separate instance, then this will need to be completely shut down anyway.
7. Using the appList shown in item 5 above, a pull down list is created, and an actionPerformed() method has the code to launch a FastPak profile with the FastPak API method runApp(). It should be noted that all responses sent from the Controller Thread to this GUI are basically trashed as the return value to runApp() does nothing with them.

The GUI in this example is very simple and intended as an example only. It could, however, with sufficient "tweaking," be of great use if the applications being run meet the following requirements:

- The applications have been heavily tested, errors are not expected or will not happen.

- The configuration in the profile is set and needs no user modification.
- There is no need for any of the other “bells and whistles” associate with the FastPak GUI.

In short, the example is a very, very basic launcher. The code may be found under FastPak's lib/Developer directory under the subdirectory called “CustomGUI.”The code for the application is shown below:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LoadableGUI{
    public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println("Usage: java -classpath...");
            System.exit(0);
        }
        String hostname = args[0];
        int portNum = Integer.parseInt(args[1]);
        JFrame frame = new LoadableGUIFrame(hostname, portNum);
        frame.show();
    }
}
```

Listing 5. The source code for LoadableGUI.java.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.scsc_online.fastpak.api.*;

class LoadableGUIFrame extends JFrame implements ActionListener {
    private JComboBox fpCommand;
    private FastPakAPI fpa;
    private String hostname;
    private int port;
    private String retVal;
    private String dummyRetBuf = "";

    public LoadableGUIFrame(String host, int portNum) {
        //Assign host and port numbers
        hostname = host;
```

```

port = portNum;
//Open up FastPak, get app list message, extract list from message
fpa = new FastPakAPI(10, 1000, port, hostname);
retVal = fpa.showAvailable();
retVal = retVal.replaceAll("The following list identifies the
applications that are currently\n", "");
retVal = retVal.replaceAll(" configured for use with FastPak:\n", "");
retVal = retVal.trim();
String[] appList = retVal.split("\n"); //create application list
for(int i = 0; i < appList.length; i++) //trim up strings
    appList[i] = appList[i].trim();
//Set up general GUI stuff
setTitle("LoadableGUI FastPak Controller");
setSize(300,100);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        dispose();
        dummyRetBuf = fpa.fastPakShutdown();
        System.exit(0);
    }
} );

//Set up the combo box, add the apps to the list
fpCommand = new JComboBox();
fpCommand.setEditable(false);
for(int i =0; i < appList.length; i++)
    fpCommand.addItem(appList[i]);
fpCommand.addActionListener(this);

JPanel p = new JPanel();
p.add(fpCommand);
getContentPane().add(p, "Center");
}

public void actionPerformed(ActionEvent evt) {
    JComboBox source = (JComboBox)evt.getSource();
    String app = (String)source.getSelectedItem();
    dummyRetBuf = fpa.runApp(app);
}
}

```

Listing 7. The source code for LoadableGUIFrame.java.

FastPak Security Issues

As identified in the document **Security Warnings Regarding FastPak for Java**, **FastPak** is **NOT** a secure product. There are, however, several ways that a developer may be able to circumvent some security issues.

With respect to the *Controller Thread*, a developer could overcome some, if not all security problems by doing the following:

1. On each machine running an instance of **FastPak**, install a fire wall (if not already installed) and assign a port ID to the **FastPak** instance running on that machine that is blocked from outside access, meaning it has localhost access only.
2. Develop an interface to the *Controller Thread* that has secure, possibly even encrypted access to the outside world that uses a different port than that of the installed **FastPak** instance. This interface (application) is to be installed and run on the same machine with the running instance of **FastPak** in question. The firewall will need to be adjusted accordingly.
3. Have applications that attempt to remotely access the *Controller Thread* do so via the secure interface and associated port.
4. When the application receives input remotely, it will decrypt it (if needed), and perform any other needed measures on the incoming message to ensure that it is properly secured.
5. Allow the interface to communicate with the *Controller Thread* via the port that is blocked from outside access using localhost as the network interface. When the activity is complete, receive responses from the *Controller Thread*, and essentially reverse the security process to send them back to the originating machine.

DO NOT ALLOW AN UNPROTECTED CONTROLLER THREAD TO BE GIVEN UNPROTECTED ACCESS TO THE OUTSIDE WORLD UNDER ANY CIRCUMSTANCES!!

With respect to applications running under **FastPak** that need security protection, a possible solution is to have **FastPak** load an application utilizing the SecurityManager class. Methods associated with this class are static, hence they will be applied **GLOBALLY** to all applications running under a **FastPak** instance. Developers need to be aware of the following:

1. One and only one instance of an application implementing a SecurityManager instance should be used since attempts to run additional managers will throw exceptions unless handled properly.
2. Applications running under **FastPak** can be bundled into several groups with each group associated with a specific **FastPak** instance. This means that a group of applications that may pose no security risks whatsoever could be bundled to run under one **FastPak** instance, while those requiring security could be bundled to run under a **FastPak** instance where a suitable degree of security can be ensured.

With that said, all developers should not take the advice just provided as being of expert value. SCSC is absolutely NOT a specialist in security. The intent of the product is NOT to be a web component with unlimited and/or uncontrolled access to the outside world, but rather a tool intended to be used underneath an environment that is already secure.