Experiment 6 – Recovery Oriented Programming

Imagine you have a service and it crashes under certain circumstances. There are two ways to deal with it. Either you fix the problem or you restart your service after it has crashed.

Fact is that large software systems are build from COTS (commercial off the shelf) components. But as another matter of fact those components might not be as reliable as you want them to be. Very often you will not be able to change them because of a lack of understanding or because you do not have the source code of those components. So the solution for this is quite simple: an automatic restart of a crashed service.

Contents

Introduction

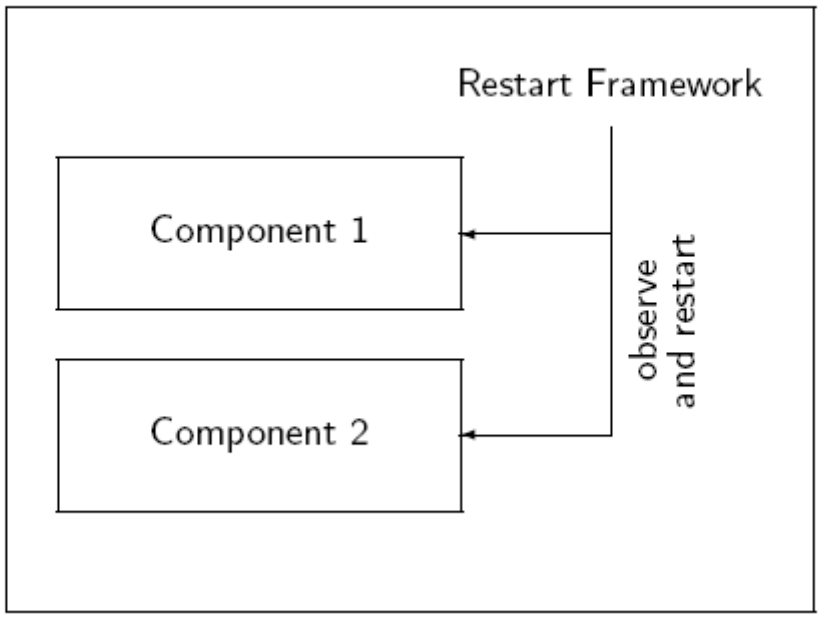
The problem with the automated restart is that the restart might take some time. In the meantime your service is unavailable. This might lead to a pretty inefficient runtime behavior since you will have to wait for the restart to be completed. So it is quite obvious which two improvements could be thought of. One is to restart only the crashed components and not the whole system and the other one is to restart **before** the crash becomes visible (to your customers).

Restart Framework

You have some code to use separate components. You will need to integrate the restart framework in this code. Its purpose is to observe the state of your components and to restart them if necessary.

You do not need to restart your whole application in the presence of a failed component. To save time it is enough to restart the failed component. But you have to respect the dependency between the components.

Architecture:



Restart of a certain component

The restart of a component running in its own thread is more difficult than that of a component without an own thread.

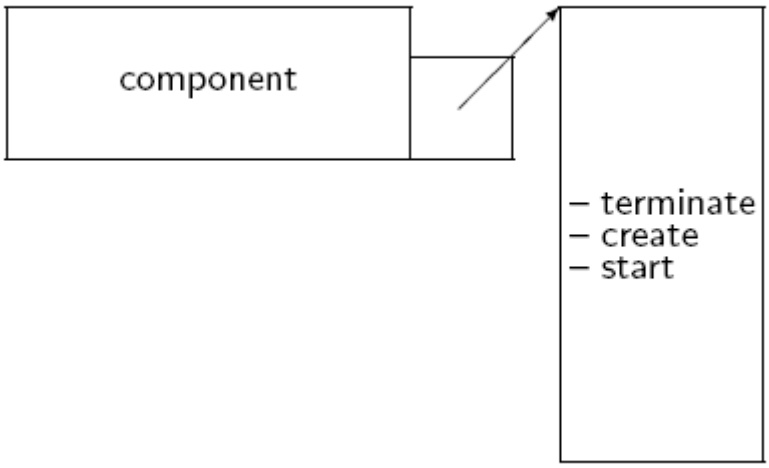
If the component has **no** own thread you do:

- free component's resources
- create new instance
- inform the other components about this new instance

If the component does have its own thread proceed as follows: in **another thread** different the component's thread:

- stop the component's thread
- free component's resources
- create new instance
- start thread of new instance
- inform the other components about this new instance

See the lower picture. The rectangles represent threads. The arrow represents the spawning of a new thread.



Your Task

You are provided with a tiny HTTP server. Unfortunately you will discover some bugs. So your task is to make the server more reliable. Find out the condition under which a certain component of the server crashes and add the ability to restart this component to the HTTP server. Please be aware that we do not give you the source code of our components! You will have to sub-class our components and provide a new main program!

Our HTTP Server consists of two components:

`HttpWorker`

- parses HTTP requests
- builds the corresponding HTTP response
- sends it to the client
- does **not** have its own thread (runs in the server thread)

`HttpServer`

- listens for clients requests
- accepts them and
- passes them to the **`HttpWorker`**
- does have its own thread

You will have to sub-class these two components. Both call virtual methods that you can overwrite to adjust its behavior. Read the javadoc documentation for them!

A simple main program

The following main program starts the server.

```
public class SimpleMain {
    public static void main (String[] args)
        throws IOException, HttpException
    {
        // parse command line arguments
        HttpConfig config = new HttpConfig (args);
        // creating the worker
        HttpWorker worker = new HttpWorker (config);
```

```
        // creating the server
        HttpServer server = new HttpServer (config);
        // connectiong the worker to the server
        server.setWorker (worker);
        // start the server
        server.start ();
    }
}
```

You will have to extend this main program by initializing your own sub-classes of `HttpWorker` and `HttpServer`.

Hints

Read the `javadocs` of our components (`doc/index.html`)!

To find the bugs start the server with `%>ant run`. Use a browser (URL: http://localhost:8080/ and observe its behavior (log messages).

The bugs are deterministic. To simulate the non-robust behavior in presence of depletion of an finite resource like file handles.

To deal with the bugs you must keep in mind that there are two bugs but you will find the second bug **after** dealing with the first one.

Print out all status information that you can acquire in your subclasses. You can find both bugs with a single run, if you retry the page loading after the first bug occurred and if you pay attention to the messages.

You will not be able to start a new HTTPServer if the last one is still running.

Familiarize yourself with threaded programming in Java. Read the Javadoc about class `Thread`.

Organizational Remarks

Deadline

Solve this week's experiment at home or in the lab. Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified day we will regard your solution as missing so you will not receive your certificate.

Directory Structure

We require a specific directory structure for your solutions.

```
roc/
  example/SimpleMain.java   - the main class
  doc/                      - java doc about server components
  build.xml                 - Ant build file (starts server)
  test/                     - web server root directory
  lib/tiny-httpd.jar        - our HTTP components
```

Hints for Automatic Check

The check will fail if you do not stick to `example/SimpleMain` as main program. If you want to define additional class files, add them under `example/`. Do not forget to add them to your SVN work directory with svn add before committing.

Out checks will fail, If you restart the components more often as necessary. So try to find the exact error condition.