Hmi

1.1 Exception handling guidelines

1. Use checked exceptions for exceptions that the client can take useful actions upon. Use RunTimeException and its subclasses otherwise ([1], item 58, 59).

2. Store information in the exceptions so that the client can use this information in recovering from them (for example: the AudioUnitPlayException contains the failing AudioUnit) ([1], item 63).

3. A chain of exceptions can be used to translate a low-level exception into a higher level one ([1] item 61). When this technique is used, always include the original exception in the higher level exception (using initCause), so that it stack trace can be used in debugging.

4. Don't ignore exceptions ([1], item 65). Either catch them and act upon them or throw them. If an exception can't happen, but has to be caught anyways, throw an AssertionError. If the occurrence of an exception doesn't influence the progress of the client (for example, a file not properly closing after reading all information from it), at least log it.

1.2 Hmi Logging setup

All logging within HMI code is done through the Simple Logging Facade for Java (SLF4J, http://www.slf4j.org/). This facade requires the slf4j-api.jar in the classpath. The output of SLF4J can be redirected to the logger of the clients choice, by adding the appropriate jars in the classpath. To allow this, the Hmi framework code itself should not put any of these logging jars in its classpath. For the HmiEnvironment demos, logback (http://logback.qos.ch/, from the developer of log4j which is no longer actively updated) is set up as the output logger.

1.2.1 Logging levels

SLF4J defines the following levels of priority ordered from low to high:

or severity or whatever is a good term here?

- trace: finer-grained than debug, discouraged, could be used for as for extra filtering/redirection as an 'extra' debug level
- debug: fine-grained informational events that are most useful to debug an application.
- info: informational messages that highlight the progress of the application at coarse-grained level.
- warn: designates potentially harmful situations. The application can continue running after these.
- error: error events that will presumably lead the application to abort.

There is a fatal level mentioned in the javadoc, but it is unclear to me what it is for, since the Logger can't log at fatal level. See . Perhaps it is only used in redirection from other loggers? See below for ref, todonotes apperently doesn't allow me to put it into the comment itself..

http://www.slf4j.org/faq.html#fatal

The level of a log message is selected by the function of the Logger that is used for logging (`logger.debug("message")`, `logger.error("error")`, etc.).

1.2.2 Parameterized logging

SLF4J's parameterized logging is a printf-like logging style:

```
LOGGER.debug("Hello {}", name);
```

This avoids the unnecessary performance overhead of string concatenation

```
LOGGER.debug("Hello "+name);
```

or code bloat

```
if (logger.isDebugEnabled()) {

      LOGGER.debug("Hello " + name);
}
```

if the log statement is *not* executed. If three or more parameters are needed, the log statement has to be called with an Object array. For example:

```
LOGGER.debug("Hello {} {} and {}", new Object[]{name1,name2,name3});
```

The `logger.isDebugEnabled()` version is still recommended if some expensive operation is needed to create the log String:

```
if (logger.isDebugEnabled()) {

      LOGGER.debug("Expensive result: {}", getExpensiveResult());
}
```

1.2.3 Logging Exceptions

SLF4J can log the Exception trace using:
`logger.error("Exception message", exception)`

1.2.4 Hierarchical logging

Like most loggers SLF4J allows one to set the log level, log destination and log format based on the name of the logger. See Figure 1.1 for an example configuration that does this. The naming scheme allows setting up level, destination and format for a group of loggers. For example, one could redirect the logging of loggers hmi.elckerlyc.PegBoard, hmi.elckerlyc.animationengine.GazeMU and hmi.elckerlyc.animationengine.AnimationPlanPlanner
by redirecting hmi.elckerlyc, or one could redirect only hmi.elckerlyc.animationengine.GazeMU and hmi.elckerlyc.animationengine.AnimationPlanPlanner by redirecting hmi.elckerlyc.animationengine.

See the logback manual http://logback.qos.ch/manual/architecture.html for a more thorough discussion on this topic.

Recommended logging naming scheme.

All loggers should be named after the class (+package) they log in, to allow redirection for specific packages. For example:

```
private static Logger logger=
LoggerFactory.getLogger(ElckerlycRealizer.class.getName());
```

1.2.5 Logback configuration and tools

The Logback configuration xml is selected by adding the run time argument

```
-Dlogback.configurationFile=logconfig.xml
```

Several example configurations are set up in the Shared/repository/logbackconfig directory. See also Figure 1.1 for an example.

The Lilith Logback event viewer (http://freshmeat.net/projects/lilith-viewer) can be used to search and filter log events written in Logback's XML format. It can also capture Logback output in real-time using a socket connection.

1.2.6 Redirecting input from other loggers

Input from other loggers (Jakarta Commons Logging, Log4j, java.util.logging) can be redirected to SLF4J (see http://www.slf4j.org/legacy.html). This is done by placing the appropriate jars in the classpath. Redirecting java.util.logging requires call to

```
SLF4JBridgeHandler.install();
```

This installs SLF4J as an additional logging handler. To get rid of the existing java.util.logging handlers use:

```
<configuration debug="true">
  <appender name="FILE" class="ch.qos.logback.core.FileAppender">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
      <layout class="ch.qos.logback.classic.html.HTMLLayout">
        <pattern>%relative%thread%mdc%level%logger%msg</pattern>
      </layout>
    </encoder>
    <file>test.html</file>
  </appender>

    <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
     </pattern>
    </encoder>
  </appender>


  <root level="INFO">
    <appender-ref ref="FILE" />
  </root>

  <logger name="hmi.elckerlyc" level="DEBUG">
    <appender-ref ref="STDOUT" />
  </logger>
</configuration>
```

Figure 1.1: Logback configuration example. Send all output to with level $\geq$ INFO to the test.html file. In addition, log all output from hmi.elckerlyc with level $\geq$ DEBUG to the stdout

```
Logger root = Logger.getLogger("");
for (Handler h:root.getHandlers())
{
    root.removeHandler(h);
}
```

There is a serious performance impact of redirecting java.util.logging over SLF4J, a 60 fold increase is reported for disabled logging statements and a 20% overhead for enabled log statements. So, only use this redirection if the third party software you want to redirect logs from software which has few log statements at performance critical places (this holds for odejava).

1.3 Unit Testing

State based testing:
1. Set system under test (SUT) in some state.
2. Call some functions etc on SUT
3. Check if SUT ended up in desired state.

Some guidelines for unit testing

1. Test cases should be simple. Reduce complexity in setup with generic setup functions and/or @Before. Reduce checking complexity and clarity with custom asserts. Don't use conditional branches in tests cases.
2. Unit tests should be stand alone. If possible, don't use files, the network, databases and don't share data between tests. Mockups/stubs/nullobjects can help here.

1.3.1 Use assertions that communicate the failure as cleary as possible

Test code smell: your test is full of System.out.println's. Use assertions that communicate failure in a better way. For example:

```
assertTrue(expected==actual)
```

Does not communicate the values of expected and actual if an error occurs. One solution is to use the string description of the assert.

```
assertTrue(expected + "<>" + actual,expected==actual)
```

But, ofcourse we're lazy and we don't want to write error messages with each assert. Custom asserts in JUnit or other libraries can help us. For example:

```
assertEquals(expected,actual)
```

Will give you information on actual and expected values if the assert fails. You can write your own asserts for custom data types. Some HMI specific asserts (for example to assert Quat4f or Vec3f equality) are stored in HmiTestUtil.

Hamcrest

See http://code.google.com/p/hamcrest/. Javadoc is available from http://www.herwinvanwelbergen.nl/hamcrest-javadoc/.

I added a more or less exhaustive list of hamcrest stuff I use below, since their documentation sucks.

Assert that there are 3 items in someList, prints the content of the list if this is not the case:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.collection.IsCollectionWithSize.hasSize;
...
assertThat(someCollection,hasSize(3));
```

Asserts that $x > 4$:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.number.OrderingComparison.greaterThan;
...
assertThat(x,greaterThan(4));
```

Asserts that x instanceof X:

```
import static org.hamcrest.MatcherAssert.assertThat;
...
assertThat(x, instanceOf(X.class));
```

Asserts that someList has items item1, item2, item3 in any order. someList may contain other items.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.hasItems;
...
assertThat(someList,hasItems(item1,item2,item3));
```

Asserts that someList has items item1, item2, item3 exactly in that order. someList may not contain any other items.

```
import static org.hamcrest.MatcherAssert.assertThat;
import org.hamcrest.collection.*;
...
assertThat(someList,IsIterableContainingInOrder.contains("item1","item2","item3"));
```

Asserts that someList has items item1, item2, item3 in any order. someList may not contain any other items.

```
import static org.hamcrest.MatcherAssert.assertThat;
import org.hamcrest.collection.*;
...
assertThat(someList,IsIterableContainingInOrder.contains("item1","item2","item3"));
```

Asserts that someCollection has size 1 (does not show list items if this is not the case)

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.collection.IsCollectionWithSize.hasSize;
...
assertThat(someCollection,hasSize(1));
```

Asserts that someCollection is an empty list of String (and shows the list items if this is not the case)

```
import static org.hamcrest.MatcherAssert.assertThat;
import org.hamcrest.Matchers;
...
assertThat(someCollection,Matchers.<String>empty());
```

1.3.2 Mocking, stubbing, behavior based testing

JMockit

http://code.google.com/p/jmockit/

1.3.3 Test hierarchy, test superclasses

1.3.4 Parameterized tests

1.3.5 Make your stuff testable

Dependency injection!

This:

```
class Foo
{
private final Bar bar;
public Foo()
{
bar = new Bar();
}
}
```

Is less testable than this:

```
class Foo
{
private final Bar bar;
public Foo(Bar b)
{
bar = b;
}
}
```

Because in the first case you cannot easily mock bar, which makes it hard to test Foo in isolation. The second form is called dependency injection (you inject the dependency to bar into the constructor). If you insist on having a convience no-argument constructor, you can use something like:

```
class Foo
{
private final Bar bar;
public Foo(Bar b)
{
bar = b;
}

public Foo()
{
this(new Bar());
}
}
```

Bibliography

[1] Joshua Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.