Best Practices for High Quality Code

Information

This page offers some best practices and standards for code which should help ensure it is more maintainable and higher quality. Some tips will help ensure you have better performance and scalability and others will increase the change tolerance. It's not good enough to just get something working. It has to be maintainable.

Automated testing

- Code should include automated testing to ensure stability, reliability, change tolerance, and correct operation
  - We recommend that code should have unit testing with at least 50% overall code coverage
    - Function coverage should be 90% or higher
  - Unit testing should always run during Sakai compiliation

Practice Test Driven Development

- Create the Class/API -> Write the Test -> Program the Implementation -> Run the Test
- Forces you to use your own method (and hopefully check if it is intuitive to use)
- Requires an immediate check against the javadoc (API)
- Makes the developer think about how the method will work and what it does BEFORE they write any code

Always create and include unit tests when writing services

- Unit testing should be added to test methods
  - This is most appropriate for testing methods which have no dependencies on external services (e.g. utils)
  - If methods make heavy use of other services then it may be better to simply write an integration test
  - Mocks can be used but are not a substitute for full integration tests
  - **Example:** EntityReferenceTest
  - Sample unit test:

```
public void testCountAsInString() {
   // positive test (e.g. test getting something back)
   int count = Stuff.countAsInString("aaronz");
   assertEquals(2, count);

   // negative test (e.g. test getting nothing back)
   assertEquals(0, Stuff.countAsInString("xxxxxxx"));

   // exception test (test that the right exception is generated)
   try {
      Stuff.countAsInString("");
      fail("This should have died");
   } catch (IllegalArgumentException e) {
      assertNotNull(e.getMessage());
   }
}
```

Add integration testing for critical code

- Integration testing ensures components are operating together correctly
    - Integration/validation/load testing should be easy (and possible) to run in Sakai
- DB integration tests allow the developer to check if their data layer is working correctly
    - These can be run in eclipse and during the maven build
    - These are structured like standard Junit tests but use AbstractTransactionalSpringContextTests to handle injections
    - Example: EntityBrokerDaoImplTest
- Logic integration tests allow the developer to test their logic layer/service with the DAO underneath it
    - These can be run in eclipse and during the maven build
    - These are structured like standard Junit tests but use AbstractTransactionalSpringContextTests to handle injections
    - Example: EntityBrokerImplTest
- Full integration tests can be generated and executed using test-harness or test-runner
    - These tests are executed within a running Sakai Component Manager (possibly a full running Sakai system)
    - These are structured like standard Junit tests but use SpringTestCase or SakaiTestCase to handle injections and provide access to some helper methods
    - Example: Test Runner

Team/Pair Programming

- Team or pair programming involves one driver (on the computer) and one or more co-pilots (observer/advisor)
    - Encourages discussion of code design decisions and best practices
    - The observer catches syntax and more importantly logical errors
    - Multiplies knowledge of the codebase and encourages sharing of expertise
    - Discourages the use of poor practices and short-cuts (laziness)

Exception Handling

- Appropriate exception handling is critical for identifying errors and finding problems in code

Never swallow exceptions or leave catch blocks empty

- Leaving empty catch blocks is a bad idea in almost all cases
    - The notable exception to this is the try->catch->finally->try->catch blocks in JBDC and things like that
- This is typically a result of "throwy" interfaces

```
public void setThing(String thing) {
   try {
      savedThing = Long.parseLong(thing);
   } catch (Throwable t) {
   }
}
```

    - Does this do what the developer calling this would expect if an exception occurs?
        - No, there would be no warning that the value was not set, it would appear to have worked when it had actually failed
    - At LEAST put in a log statement (warn level) but it would be better to actually rethrow this exception

Don't throw java.lang.Exception

- java.lang.Exception should not be used in interfaces (throws Exception) or to indicate an error occurred
    - Use an appropriate type of exception (IllegalArgumentException) or create one by extending RuntimeException
    - Always provide a message with the exception with as much information as would be needed to understand the failure

Don't catch Throwable

- java.lang.Throwable should not be caught since it is the superclass of all errors and exceptions in the Java
    - includes java.lang.Error (indicates serious problems that a reasonable application should not try to catch) and OutOfMemoryError (among others)

Maintain the stack when re-throwing an exception

- Pass on the cause of the original exception when rethrowing an exception which was caught

```
try {
   newC = (T) Array.newInstance(componentType, 0);
} catch (Exception e) {
   throw new IllegalArgumentException("Cannot construct array of type: " + componentType + " for: " +
beanClass, e);
}
```

Minimize use of "throws" (checked exceptions) in method definitions

- Don't make it hard on developers who are using your interfaces

```
public void badInterface(String thing) throws
        NullPointerException, ActivationException,
        AlreadyBoundException, BadStringOperationException,
        InstantiationException, InvalidApplicationException,
        UnsupportedFlavorException;
```

- These are known as checked exceptions
- This leads to developers capturing all the exceptions and writing empty catch blocks because they do not know what to do with all these exceptions or possibly plan to come back to it later and forget
- From Tim McCune (definition of checked and unchecked exceptions):
  _Checked exceptions are exceptions that must be declared in the throws clause of a method. They extend Exception and are intended to be an "in your face" type of exception. A checked exception indicates an expected problem that can occur during normal system operation. Some examples are problems communicating with external systems, and problems with user input. ... Often, the correct response to a checked exception is to try again later, or to prompt the user to modify his input.

Unchecked exceptions are exceptions that do not need to be declared in a throws clause. They extend RuntimeException. An unchecked exception indicates an unexpected problem that is probably due to a bug in the code. The most common example is a NullPointerException. There are many core exceptions in the JDK that are checked exceptions but really shouldn't be, such as IllegalAccessException and NoSuchMethodException. An unchecked exception probably shouldn't be retried, and the correct response is usually to do nothing, and let it bubble up out of your method and through the execution stack._

- Throw RuntimeExceptions instead (unchecked exceptions)
    - These are typically used to indicate programmer failures and are the way to guide the programmer so they can fix the code that is calling your methods/interfaces
    - Don't forget to document them in the Javadoc API though (using @exception)
- Rule of thumb: If a developer can reasonably be expected to recover from an exception, make it a checked exception (throws). If they cannot do anything to recover then make it an unchecked exception (runtime).

Use a static code review tool like PMD

- Code review tools can be used to look for common issues in your code. Many of these can be setup to run during your maven builds
    - Fix the issues that are identified by the static code review
- You can look at the Sakai code review here: http://qa1-nl.sakaiproject.org/codereview/bug_dashboard/
    - Contact Alan Berg (a.m.berg@uva.nl) to get your code added to the review

Keep It Simple Stupid

- Simple code is elegant and can be harder to write but easier to maintain and understand

- Complex code is hard to read
- Overly clever code is hard to understand
    - For example: Putting too much on a single line is hard to read, hard to debug, and makes reading stacktraces more difficult

```
if (service.addThing(users.getUser(item.userId),
        sites.getCurrentLocation().getSite().id, Format.getformattedtext(texturl))
        == previousCheck.getResult().convertToInteger()) {
                // do something here, but when is anyone's guess :-(
}
```

Use appropriate logging levels

- Use of appropriate logging levels helps admins and developers deal with log messages appropriately
    - When logging things that are informative for normal usage, use INFO level
    - If a failure occurs which is not a problem then use WARN level, but do NOT use warn to indicate that something failed!
    - That is what ERROR and FATAL log levels are for (so use them please)
    - Using log levels that are inconsistent will confuse and upset the production team AND developers

Minimizing dependencies

Don't depend on implementations

- Always use the API/interface when it is available and do not cast objects to implementation types
    - The possible exception to this is if you are using the implementation inside your own project

Use IDs instead of objects when possible

- Try to use IDs instead of heavy objects in interfaces
    - This reduces the exposure of dependencies and allows for simple communication of data
    - This also makes it easier to work with webservices and webapps
    - Example: **String findUserEmailById(String id)** would be better than **String findUserEmail(User user)**
        - This allows a developer to find the email address easily if they have the user object OR if they only have the id of the user

Interface design

Keep interfaces small (one interface per logical piece)

- Interfaces which have to be implemented by developers should be as small as possible
    - Smaller interfaces are easier to implement and understand

Use an extension model

- Break up the interfaces into logical pieces which can be added together to extend the capabilities of the code being implemented
    - This **capabilities** style is more flexible and easier to work with for implementors
    - It keeps developers from implementing classes and leaving methods unimplemented which they do not understand or need

Include detailed Javadocs with examples

- Good documentation for APIs is absolutely critical for making the interface usable and understandable
    - Use the javadocs standard style
    - Specify valid and invalid inputs
    - Specify exactly what is returned (especially if nulls are returned sometimes)
    - Specify the Exceptions that are returned if they are something the developer might need in order to work with the interface
        - Don't bother indicating that NullPointerException is returned if a null is provided, but you might indicate that IllegalArgumentException is returned when a supplied value is invalid

General Best Practices

- Documents and defines best practices for programming in Sakai (though many of these are generally good pratices to follow for Java in general).

Minimal usage of synchronized collections (Hashtable, Vector, etc.)

The use of synchronized collections (Vector, Hashtable) should be a last resort and only used when absolutely necessary. These should primarily be used for thread safety. However, the cost of writes AND reads are expensive when using these (easily 10-100x more than using the basic unsychronized versions). The cost of synchronization can become even more severe in a highly concurrent environment like Sakai, especially when it is deployed on a multi-core server.
When thread safety is not needed (99% of the time in Sakai) an ArrayList, HashSet, or HashMap should be used instead.
If thread safety is needed together with the performance of lock-free code, the best choice will usually be ConcurrentHashMap. CopyOnWriteArrayList and CopyOnWriteArraySet are other classes that could be useful in very specialised situations.
More info on concurrent collections here: http://www.ibm.com/developerworks/java/library/j-jtp07233.html

Use StringBuilder for appending strings when appropriate

Strings can be appended in many ways in java. The most common are to simply add the strings together, use StringBuilder, or use StringBuffer. These are all appropriate at different times.

1. Adding Strings
   This is appropriate when all the strings are appended in a single statement.

   ```
   String newVal = string1 + " " + string2 + " more stuff " + blah.toString() + ":" + thing.getValue().
   title;
   ```

   You might hear people telling you that you should be using a StringBuffer/StringBuilder here but that is silly because that is what the compiler converts this into anyway. No reason to make your code longer and uglier.
2. Using StringBuilder
   This is appropriate when you are appending strings in multiple statements. I would personally not bother unless there are 3+ strings to append.

   ```
   StringBuilder sb = new StringBuilder();
   for (int i=0; i < thing.size(); i++) {
     sb.append(thing.get(i));
     sb.append(":");
   }
   String newVal = sb.toString();
   ```

3. Using StringBuffer
   In Java 1.5-level code and above, any explicit use of StringBuffer should be replaced with the lock-free equivalent StringBuilder. There is almost never a good reason to use StringBuffer. This is a synchronized object and therefore very slow to use unless you actually need the synchronization. In general, you should never use this over StringBuilder unless you have a very good reason (and need thread safety).

Minimize API dependencies

Interfaces in Sakai services (and in general) should be written to minimize dependencies on other packages when possible. This means not requiring a User object when the identity of a user is needed, but instead requiring the userID. When using an id instead of the actual object you should clearly identify what is expected in the javadocs like so:

```
    /**
     * @param assignGroupId the id of an {@link EvalAssignGroup} object to remove
     * @param userId the internal user id (not username or eid)
     */
    public void deleteAssignGroup(Long assignGroupId, String userId);
```

Use numeric autogenerated ids in database tables

There are many ways to generate IDs for database tables, but one of the simplest, fastest, and safest is to allow the database to do it for you using numeric increamenting. This is supported in every database and is easy to do with hibernate. This also can reduce locking issues in MySQL and creates relatively simple looking IDs. Here are some helpful links:
HSQLDB Identity MYSQL autoincrement ORACLE sequence

- If you want to ensure that you also have globally unique ids for your data, there are 2 ways to make these globally safe
  1. Append prefixes like: edu.rutgers.sakai.evaluation.answer.123, (system-prefix).(app-prefix).(table-prefix).id
  2. Store an eid (GUID, or whatever you like) in the tables along with the autogened id and use that externally

Compare strings/constants by putting them on the left side of the comparison

You can avoid many NullPointerExceptions by simply always placing hardcoded string and constants on the left side of an object comparison. This works well for any object constants (not just strings).
For example, this code can generate a NullPointerException if **myString** is null (bad):

```
if (myString.equals("hardcodedstring")) {
    // do something
}
```

This example code can never generate a NullPointerException (good):

```
if ("hardcodedstring".equals(myString)) {
    // do something
}
```

Cleanup user submitted strings before storing them (i.e. do not trust input)

You can avoid issues related to XSS (Cross Site Scripting) by simply cleaning up the strings that are submitted to your services. This should be done in all systems and can be handled in the webapp (tool) or in the services (probably more ideal). In Sakai this can be done using the utils from the kernel but external jars are fine as well if you have one you are comfortable with.
For example, this code takes a user submitted string and returns the cleaned up version. It would be called any time there is any user data being input (that includes data from web form radio buttons and pulldowns).

```
String cleaned = FormattedText.processFormattedText(userSubmittedString, new StringBuffer());
```

Ideally a method that does not require creating a StringBuffer would be available but that is not currently the case.

References

- http://www.squarebox.co.uk/download/javatips.html
- Exception Handling
    - http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
    - http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html
    - http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html
- http://c2.com/xp/CodeSmell.html
- http://en.wikipedia.org/wiki/Pair_programming
    - http://xpairtise.sourceforge.net/
- http://www.refactoring.com/catalog/
- http://en.wikipedia.org/wiki/JUnit
    - http://en.wikipedia.org/wiki/Unit_testing
    - http://en.wikipedia.org/wiki/Integration_testing
- http://en.wikipedia.org/wiki/Software_documentation
    - http://java.sun.com/j2se/javadoc/
    - http://java.sun.com/j2se/javadoc/writingdoccomments/