

Elua Group WattDepot-CLI Code Review

A. Review the Build

A1. Download the system, build it, and run any automated quality assurance checks (use "ant -f verify.build.xml")

The build was successful.

B. Review System Usage

B1. Run the system. Exercise as much of its functionality as possible.

I initially tried running it from the jar file and had some errors. So I tried running it within Eclipse and it was alright. But a huge problem arose and for some reason only the command 'help' and 'list summary' worked and the rest didn't, especially 'quit'. After they released a second distribution file that only fixed the non-executing jar file, I tried running from it and was successful. So this review focuses on the second release. After running all the commands, I found all of them worked perfectly except for 'list total carbon/energy generated' command. The command does not correspond to the command format found on after calling help.

B2. Try to break the system by providing it with unexpected input.

The program crashes after calling 'list summary' without the source.

C. Review the JavaDocs

C1. Does the System Summary (provided in an overview.html file) provide a high-level description of the purpose of the system? Does it explain how each of the packages in the system relates to each other? Is the first sentence self-contained?

The resulting Javadoc overview provided a concise description.

C2. Do the Package Summaries (provided in package.html files) provide a high-level description of the purpose of the package? Do they explain how the classes in the package related to each other? Is the first sentence self-contained?

They provided a sufficient explanation.

C3. Does the Class Summaries (provided at the top of each .java file) provide a high-level description of the purpose of the class? Does it provide sample code for clients of the class, if useful? Is the first sentence self-contained?

They could improve the summary by including a better Javadoc explanation of what each of class does and not just saying it's a class.

C4. Does the Method Summaries (provided before each method) explain, from a client-perspective, what the method does? Do they avoid giving away internal implementation details that could change? Do they document any side-effects of the method invocation? (Note that you can click on the method name to see the source code for the method, which is helpful to assessing the correctness and quality of the Javadoc.)

This group has avoided giving away implementation details.

C5. Please review Chapter 4 of the Elements of Java Style for additional JavaDoc best practices, and check for compliance.

They need better explanation of what each of their commands' class does and they can do so with Elements of Java Style #47 and #50.

D. Review the Names

D1. Do another pass through the JavaDocs, this time concentrating on the names of packages, classes, and methods. Are these names well chosen? Do they conform to the best practices in Elements of Java Style, Chapter 3? Can you propose better names?

They used meaningful names to for their variable.

D2. Once you have reviewed the names displayed in the JavaDocs, review the source code for internal names in the same way.

They also used meaningful variables names within their source code.

E. Review the testing.

E1. Run Emma or some other code coverage tool on the system ("ant -f emma.build.xml"). Look at the uncovered code in order to understand one aspect of the limitations of the testing.

The Elua group did pretty good test cases and considering the uncovered code, they are mostly the exceptions for the each of the classes.

E2. Review the test cases. Is each component of the system exercised by a corresponding test class? Do the test cases exercise more than "happy path" behaviors?

The tests were in their own class and they were more than "happy paths" such as comparing the class generated result vs. the client generated result in 'TestTotalGenerated'.

E3. Review the test output. Under default conditions, the test cases should not generate any output of their own. It output is desired for debugging purposes, it should be controlled by (for example) a System property.

Their outputs from the tests are controlled by the System property.

F. Review the package design

F1. Consider the set of packages in the system. Does this reflect a logical structure for the program? Are the contents of each package related to each other, or do some packages contain classes with widely divergent function? Can you think of a better package-level structure for the system?

They separated the main class and the commands in different packages.

G. Review the class design

G1. Examine its internal structure in terms of its instance variables and methods. Does the class accomplish a single, well-defined task? If not, suggest how it could be divided into two or more classes.

Their design followed the requirements of having each of the commands in their own class but they didn't have a separate class whose task is to parse the user input and pass proper arguments for the commands to process.

G2. Are the set of instance variables appropriate for this class? If not, suggest a better way to organize its internal state.

The instances were appropriate since their design have each of the commands in separated classes.

G3. Does the class present a useful, but minimal interface to clients? In other words, are methods made private whenever possible? If not, which methods should be made private in order to improve the quality of the class interface to its clients?

From their design, I think that they should've made all their commands private since the commands, separated into classes, should not be visible to the other commands because they don't concern each other.

H. Review the method design

H1. Does the method accomplish a single thing? If not, suggest how to divide it into two or more methods.

They have done a good job on reducing the commands to do a single task.

H2. Is the method simple and easy to understand? Is it overly long? Does it have an overly complicated internal structure (branching and looping)? If so, suggest how to refactor it into a more simple design.

The methods are generally straightforward and easy to understand thanks to their inline a comment that tells exactly their loop does and the different recurring effects.

H3. Does the method have a large number of side-effects? (Side effects are when the result of the method's operation is not reflected purely in its return value. Methods have side-effects when they alter the external environment through changing instance variables or other system state. All "void" methods express the results of their computation purely through side-effect.) In general, systems in which most methods have few or zero side-effects are easier to test, understand, and enhance. If a method has a large number of side-effects, try to think about ways to reduce them. (Note that this may involve a major redesign of the system in some cases.)

Their methods don't have side-effects since they are creating a string buffer which they passed as string.

I. Check for common look and feel

I1. Is the code implemented consistently throughout the system, or do different sections look like they were implemented by different people? If so, provide examples of places with inconsistencies.

The overall look and feel of the code is implemented consistently throughout the system.