**GridWorld Testing and Analysis Report**

**Vipul Kohli** and **Abdullah Albadri**

**Abstract**  
GridWorld is a set of Java packages that form a GUI and API to create grid-based characters and programs that provided the material for this testing project. Black-box checklist testing was applied to the several functions of the Location class based on the documentation provided by its author. Data flow testing was applied to the color-changing algorithms of the ChameleonCritter. The movements of different bugs and the boundaries of the UnboundedGrid and BoundedGrid allowed for different types of boundary testing in which several defects were discovered. Finite-state machine modeling and testing was applied to the menus of the GUI to distinguish the most important features and test their paths accordingly.

**Black-box Checklist Testing: Location**

**Artifacts.** See Appendix A for documentation on the Location class from 2 different authors and the original Location.java source code file which includes licensing, packaging, and authorship details.  
**Background.** The Location class takes in both a row and column coordinate upon instantiation and contains the associated getter and comparison functions. These functions allow for arithmetic operations with a single Location object or two Location objects. However, the Location class itself has no internal concept of a grid and boundaries therein, so it can handle all signed integer coordinates within the four byte range.

**Problem.** The purpose of this testing approach is to verify the meaning of the documentation and the interrelationships among the functions in the Location class so that client programmers and other Java classes can use the Location class appropriately without confusion. Upon verifying the meaning and interrelationships of the functions as easily understandable, validation across several test inputs and test oracle outputs demonstrates consistency proving that the functions are behaving according to their specifications. Client programmers encountering the Location class naturally simply read the documentation or the function signatures and immediately apply test inputs to verify the Location class's compatibility with their application, so the Location class must be ready for such usage. This testing approach will be successful if the behavior observed by each function is consistent and easily understood with the documentation and each function's signature.  
**Strategy and Activities.** The first priority was verifying the behavior of each function with a single test case and test oracle output based on the documentation and each function’s signature, which is defined by its name and parameter list. This was done in a stepwise manner by manually providing an input to each function and observing the output. At the beginning, this manual approach was taken rather than automated approach in order to use a tracer bullet approach to verify the inputs and outputs are being provided in the expected format. For example, if a particular input to a function call results in an error, exception, or other unreasonable output, further analysis can be applied to either understand the concept of the function or reformat the input. This approach is the typical approach a client programmer would take to learn the semantics of the Location class.  
After verifying the reasonable behavior of each function, validation for consistency can be automated to verify expected behavior across several test inputs. This is done by providing an ordered function list and an ordered collection of oracle outputs to a tester program, and the program will return either true or false for a match against the oracle output. This is a typical situation for the developer who knows fully about how each function should behave but wants to ensure the behavior for different sensitized input combinations.  
**Results and Analysis.**  The oracle values are compared to each function's returned value in Table A1. Upon first reading of the documentation and applying of the manual stepwise checklist test, some unexpected values were returned particularly in the getDirectionToward and compareTo functions. All other functions matched the oracle value, appearing defect-free with this test run. By further investigation into the unexpected returned values, there were some vague descriptions in the documentation and naming of functions. The getDirectionToward returns the nearest 45-degree compass direction from the current Location to the target Location. In the documentation provided by the author, as shown on page A2, the description simply says closest compass direction. Upon first reading, I was thinking "This function is going to return an integer angle with 0 being north doing arithmetic and return an arctan," but the author intended "compass" to refer to the integer angles of north, northeast, east, southeast, south, southwest, west, and northwest which have compass directions as multiples of 45. This approach completely ignores trigonometry making the documentation and function signature somewhat misleading.

Similarly the compareTo function also had vague documentation. I was expecting the Location’s compareTo to return a magnitude difference just as compareTo functions are defined other Java classes such as the String class. For example, (3,0) compared to (1,3) should return 2, but it only returned 1. The reason became apparent when (3,0) was compared to (6,0) and the result was -1. Apparently, The author intended the compareTo function to only decipher a less-than, equal-to, or greater-than replationship as -1, 0, and 1 without considering magnitude. With this experience, black-box checklist testing helps find confusing terminology in function signatures and documentation descriptions.  
 Checklist testing also helps identify useful features that are "missing" in the current release. For example, if a client programmer wanted to convert from Location's row and column orientation to an x and y orientation, the functionality is not yet available. Such functionality can then be put on the list for a future implementation of the Location class. Client programmers will need to keep the functionality limitation in mind should they need it for their applications.

Once the manual stepwise testing was completed, automation can be used to run multiple test runs to validate the Location class across several input combinations. An excerpt of the automation code is provided in the Appendix on page A3. Once all the concepts were clearly defined from the initial stepwise manual checklist testing, all the results from the automation were true against the test oracle showing that the Location class behaves defect-free.

**Lessons Learned.** Documentation and function signatures as provided by a developer can be either incomplete or unclear, and black-box checklist testing is an effective method to identify the specifics of these functions to still be able to use them reliably as a client programmer. As a client programmer, checklist testing also allows me to determine what a particular third-party software has and lacks in order to find or write the code necessary to optimize for my particular purpose. As a developer of a software product, automated black-box checklist testing effectively validates that all of the software's functions are consistently behaving as intended for a variety of inputs.  
**Followup Actions.** The documentation shall be revised to more clearly state the significance of the parameter and return values, especially of the compareTo and getDirectionToward functions. Perhaps these will be renamed to getComparisonOf and getPrincipalDirectionToward to be more exact respect to their return values. Clients shall be warned of the row, column orientation opposed to the x,y orientation. Future releases shall have an XY conversion mechanism. The automated checklist test shall be saved for future use in case that additional software may "break" the Location class.

**Black-box Finite State Machine testing**

**Artifacts.** See Appendix B for figures showing samples of boundary testing

**Background.**

**Problem.**

**Strategy and Activities.** Finite state machine is not only consider a testing technique, but also consider as an assistance which leads to zoom out the software to get some subtest cases to the Grid and the actors as well Figure which consider detecting the interior functions. We made FSM with Markov chain to the Grid World Figure. We built Markov OP Figure which helped us to identify the most important areas in the system from a user perspective. Finally we got a functional OP to the system in order to test the frequently used regions.We have gone deeply to test the boundedGrid and draw an actor as well which consider the engine of this software in order to compare Finite State Machine testing results with boundary testing technique which will be covered subsequently.

**Results.** After building the Finite State Machine models and the test cases we have sensitized all paths and we found two defects:

1. The first problem in the state boundedGrid when the user starts to set the boundaries by entering values greater than 8\*8, the grid will be less than the expected size.
2. The second problem when the user moves an actor in the sub Finite State Machine draw an actor and the new location is greater than grid size, it will be a logic error that result in the move to state.
3. The last problem when we use setGrid function, we get an exception.

**Result Analysis.** In the Finite State Machine Testing some defects were caused by logical exceptions, and the other were caused by inaccurate coding.

**Lessons Learned.**

**Followup Actions.**

**Black-box Boundary testing: Grid and Bug**

**Artifacts.** See Appendix C for figures showing samples of boundary testing

**Background**. As we mentioned above boundaries play an important role in the GridWorld. Boundaries must have a serious consideration among the rest of our testing techniques by test all the boundaries to verify inputs in the Grid boundaries. So, boundary testing is necessary for the most important features based on the functional OP. The most important areas in the GridWorld are BoundedGrid and draw an actor which we have done different boundary testing strategies such as Extreme Point Combination, Weak N\*1 and Weak 1\*1.

**Problem.**

**Strategy and Activities.** We drew an actor inside the grid by drawing a Box Bug and tested it by EPC testing strategy. So, EPC worked with the Box Bug and it covered all the 16 extreme points and the Box bug pass the test. It can be clearly seen that the box bug made a square shape which can be tested by EPC without any problem. Thus, EPC covered all the boundaries. In addition, we drew another actor as a Spiral which has a continues movement and tested it by EPC as well. However, EPC didn’t cover all the boundaries of the Spiral. As result, the test is failed because of its movement.

Once we have used N\*1 strategy by testing the spiral and it passed the test by adding 6 ON points and three OF points we covered all the Spiral's boundaries.

Additionally, we have used weak N\*1 strategy in one dimensional box bug. Thus Weak N\*1 detected a boundary shifting. This strategy leads to caught the shifting witch happened in the external ON point.

**Results.** Boundary testing covered a few defects by using different techniques. In the boundary testing we tried to test the complicated areas which tread as the artery of the software.

1. We found boundary shifting problem when we used Weak N\*1 technique.
2. When the user sets boundary size to "0" the GridWorld will be crashed.

**Result Analysis.** Most of the defects detected throughout Boundary Testing have caused by entering invalid integers, and we detected a boundary shifting problem.

**Lessons Learned.**

**Followup Actions.**

**Data Dependency testing - Vipul**