Purpose: The SQA team works throughout the duration of the product development to ensure that project standards are met, to test new features and components that are added to the workflow, and track any faults or errors that may arise.

1. Fault Tracking (Joel)

a. Fault list

i. **Description:**The fault list is a single document that will track and record errors, faults, and failures within the project both past and present. Faults will each be flagged in various ways to signify their status and prevalence to the project. All faults will fall into either performance faults or design faults.

ii. **Performance Faults:** These are errors, bugs, or failures that have arisen somewhere in the code implementation of the project. These faults typically appear at some point during runtime. A performance fault is addressed within the code base by replicating the bug, patching the problem, and testing that the same or other bugs do not arise.

iii. **Design Faults:** Design faults are issues that have been flagged with the planning and direction of the project. They are present before the code implementation, but often only realized once the coding begins. These faults are addressed by updating the requirements to work around the design issue.

b. Fault flagging

i. **Description:** Here is a standard for fault flagging so that the team can easily maintain faults and determine their importance. A fault will have the following format:
Fault num, Status, Category, Criticality level, Location (document, line num), Description

ii. **Fault num:** Simple fault counter for easy reference within the fault doc. Current fault num = 1 + previous fault num.

iii. **Status:** Denotes how far along the faults is in being addressed. The status flags are:
  1. **TBD:** Has had no attention, only identified in the code walkthrough
  2. **FIXED:** Fault has been patched in the code or relevant document, but has not been tested or reviewed rigorously.
  3. **TESTED:** Fault has been patched, tested, and has passed all review.

iv. **Category:** Whether the fault is a performance or design fault.
  1. **PERF:** Performance fault
  2. **DESIGN:** Design fault

v. **Criticality level:** Scale from 1 (minor bug) - 5 (breaking error) indicating how crucial the fault is and thus its importance.

vi. **Location:** Specifies the file and line number(s) that are relevant to the fault.

vii. **Description:** Relevant information about the fault, how to replicate, and any other notable observations about the issue.

2. Testing

a. Non-Execution-Based Testing (Abi)

Description: This section is the analysis of our code base in much detail without actually running the software. In this phase the code would be carefully read through by the SQA team, and compared to the documentation relevant to the component of the code under review. Non execution based testing in the cellular automata simulator will be used for various software verifications including checking that the backend simulator engine is intuitive and has no obvious errors, assessing the object oriented GUI design for the gird and controls panels, and locating conceptual faults.

i. Walkthroughs

Description: Prior to inspecting the code, the SQA team will select either a participant-driven or a document-driven approach to fault finding. The selection will depend on factors such as the complexity of the code, the experience level of the reviewers, and the need for structured guidance during the walkthrough.

1. **Participant-Driven**: Members of the SQA team independently review the code and identify faults prior to the walkthrough meeting. Individuals will compile a list of suspected faults to be addressed by the group members responsible for the code. This approach is best suited when the SQA team is already familiar with the document's contents and can effectively identify errors independently.
2. **Document-Driven**: The group members responsible for the code present it to the SQA team, systematically explaining each section. SQA team members are encouraged to interrupt for clarification or to highlight potential faults. This approach is preferred for more complex code, as it ensures a more thorough review and structured discussion.

ii. Inspections

Description: Inspections are a structured review process where the SQA team evaluates the code for faults before it is finalized. The steps the SQA team will follow for inspecting the code are outlined below.

1. **Overview:** Before distributing the code for review, one of the responsible group members will provide the SQA team with a brief overview of its purpose, goals, and expected outcomes. This ensures the SQA team can conduct their inspection with a clear understanding of the code.
2. **Preparation:** To prepare for the inspection, SQA team members will review previous inspection records and compile

a list of common faults. This helps streamline the identification and reporting of errors, improving efficiency.

3. **Fault finding:** The SQA team will walk through the code using one of the methods described above. After the inspection, the team will compile a report detailing the identified faults and return it to the responsible group members.
4. **Rework:** The designated group members will use the fault-finding report to correct all identified issues within the code.
5. **Follow-up:** The SQA team will follow up with the responsible group members to verify that all faults have been adequately addressed.

b. Execution-Based Testing (Joel)

Description: The purpose of the execution based testing is to check that the expected results from each software component indeed matches the outcome. It cannot be an exhaustive search for all bugs since that would require an infinite amount of processing and time, but rather uses a small sample of test cases to provide confidence that each component of the code will not fail. Execution based testing in the CA simulator will be used to verify that the major components like the state grid, control widgets, and rule setting work as expected, and testing the backend with a few rules and initial conditions to ensure that the simulator results match commonly known results.

i. Utility

Description: How is the user's experience with the CA simulator? These are some heuristics for testing how usable the product is.

1. How easy is the state grid to interact with? Is it hard for users to select a specific cell within the grid?
2. Do all the controls and options relate to the purpose of the tab that they are under? Is it clear what each widget does?
3. How difficult is rule setting? Since this is the crux of the project, rule setting must be simple and intuitive?
4. Is the appearance of the project modern and aesthetically pleasing?
5. If the user makes an accidental mistake, how simple is it for them to undo that mistake.

ii. Reliability

Description: How often do failures arise in the software and how much difficulty will the user have should an error arise? Will they be able to work around it or will they need to call tech support? Highly inconvenient errors should be avoided as much as possible. Here are some questions to consider how reliable the CA simulator is.

1. What is the most problematic aspect of the simulator? Does the state grid cause problems? Is it the backend? Do the widgets always do what they are intended to do?
2. How well does importing / exporting simulations work?
3. Are the statistics accurate to the simulation? Consider things like integers overflows or division by zero.
4. How frequently do issues arise when using the program?
5. Will turning it off and on again fix most problems?
6. Does interacting with each widget produce the same expected results every time? Are there any surprises?
7. Test the start, stop, pause, and step buttons that they do what they are supposed to given various stages of the simulation. Ie. we cannot step through the simulation if it has no rule.

iii. Robustness

Description: Robustness tests assess the product's ability to handle edge cases and inputs on the very extremes of its capabilities. A robust product should rarely ever crash or freeze in any situation. The CA simulator must handle invalid inputs before they get into the backend and cause a crash.

1. If the computation demand gets heavy, what does the program do? Does it crash? Do the steps remain synced or do they get out of phase?
2. Are overflows and out of range values handled properly.
3. Do all the mathematical elements of the backend handle invalid inputs and respond accordingly?
4. Do we have error handling and a system to communicate invalid inputs to the user?
5. Has the product been tested on the highest possible settings and the least capable supported devices?
6. Do we have memory leaks that will slowly crash the application?
7. If the simulation begins to run slowly how can we handle more user requests as typically happens when a software product is not performing well?

iv. Performance

Description: One of the core aspects of a good user experience is how well the application performs. Is it fast and responsive or does it feel like it was developed in the year 2000? The CA simulator is a computation based application and therefore may be more demanding than other standard desktop apps. These are the benchmarks we will use to assess the application's performance.

1. Resource testing, record memory, and cpu usage of the application in runtime.
2. Simulation steps per second. What is our max throughput?
3. Rule complexity. What is the most sophisticated CA rule we can handle?

4. Graphics requirements. How well does the application run on integrated graphics? Can we use the presence of a graphics card to accelerate our simulations?
5. Latency between widget interaction and response.
6. How many tasks are typically on our runtime queue? What is the max queue size and do we ever hit that?

v. Correctness

Description: This focuses on verifying that our output is indeed correct for a given input. Since our application simulates cellular automata we must ensure that our results match known results and that each rule gets evaluated properly.

1. Test 1D simulations in each different class of Wolfram cellular automata and compare to results others have obtained.
2. Test 2D simulation with Conway's game of life.
3. Test that each whenever a widget is interacted with the backend variable changes to reflect what is shown in the UI.
4. Verify that the statistics for the CA are correct at various points within the simulation.
5. Check that the displayed state grid matches what is in the backend.