Delft University of Technology

Final Report

Testing framework for AATOM

CSE2000 Software Project

Authors:

Patrik Barták, Andrej Erdelský, Matej Havelka
Amar Mesic, Vincent van Oudenhoven, Marco van Veen

Client: Alexei Sharpanskykh
TU Coach: Soham Chakraborty
TA: Giuseppe Di Giuseppe Deininger

June 13, 2021

Preface

This report about the outcomes of this project was composed by six TU Delft Computer Science & Engineering students with academic experience in software engineering. In the final quarter of our second year, we embarked on the Software Project, where in a period of over two months, we developed a testing framework for an agent-based simulator. The simulator, namely AATOM (Agent-based Airport Terminal Operations Model), is a highly valuable airport terminal simulator used for optimising and analysing airports, but was virtually untested. During our work, we fully tested it, provided it with an extensible testing suite, and proposed an improved software architecture. The goal is that all future developers of AATOM can now easily verify and validate AATOM so that it can reliably be used to make real-world predictions.

This report will discuss AATOM, the testing framework, and its implications. It is assumed that readers have a basic understanding of agent-based simulators or simulation models. An index of important terminology can be found in the report to ensure there are no misunderstandings or ambiguity. It is not necessary to have an understanding of the object-oriented programming language Java or software testing, but it can be beneficial. An introduction to relevant software testing principles and software testing for agent-based simulation models can be found in Section 2.3.
Readers interested in the structure of the testing framework can look to Chapter 6, and the requirements our product had to meet can be found in Appendix A. An introduction to AATOM can be found in the Chapter 2.

We would like to thank our client representatives, Alexei Sharpanskykh and Sahand Mohammadi Ziabari, who provided us with valuable feedback and freedom in how we implemented the testing framework. We would also like to thank MSc students, Didier v.d. Horst and Klemens Koestler, for providing us with valuable insight into how AATOM was extended and used. Lastly, we would like to thank our teaching assistant and coach, Giuseppe Deininger and Soham Chakraboty for guiding us through our process.

Delft, June 18 2021
Amar Mešić
Andrej Erdelský
Matej Havelka
Marco Van Veen
Patrik Barták
Vincent van Oudenhoven

Summary

Airport travel has grown to become one of the main methods of international travel, which means airports are getting bigger, more sophisticated, and are seeing more traffic year by year. To tackle the challenge of creating and maintaining airports which must handle ever-growing passengers and needs, software programming solutions are now being sought after. One such program, the Airport Terminal Operations Model simulator (AATOM), was developed at TU Delft. Airports have now started to use AATOM to verify that certain airport procedures would work efficiently, but there currently exists no framework to verify or validate that AATOM works as expected. These two methods of quality control are extremely important for complex computer programs that have real-world implications. On top of that, agent-based simulators are notoriously difficult to test. Given this momentous task, in our project we developed a testing framework that is complete with tests encompassing multiple layers of abstraction, and extensible such that expansions of AATOM could further be verified for correctness.

The aim of this report is to present the process of developing the testing framework, as well as the end product. First we give an overview of the AATOM simulator, who its internal and external stakeholders are, and what it is capable of. Then we follow with the main drawbacks of AATOM, and explain how they set the stage for the problem we aimed to solve in this project. We then describe the main goals of our project in a general setting. We also explain the fundamental principles of software testing which are most relevant to the development of a testing framework for AATOM, and this is complemented with our initial research findings of existing software testing techniques for agent-based simulation models. After the problem has been introduced, we move on to the requirements that were set for the testing framework, how they were prioritised, and how we measured their completion. Further on, we explain our work processes and development methodology as a programming team.

Afterwards, we introduce the testing framework (and its three layers of abstraction: micro-, meso-, and macro-), and we discuss its ethical implications. Additionally, we go into each of the three layers of our testing framework, and discuss the scope and the problems that were faced, and how we solved them. We created a test suite framework, met our testing completeness metrics, and added test builders & utilities. We also present a proposal for a new and improved AATOM architecture that places the autonomous agents at the core of the program, and reduces overall complexity. We end with a discussion on the testing framework and future improvements to our testing framework that we recommend. We were successful in completing our prioritised requirements, and we found and recorded plenty bugs in AATOM with our tests. Through our testing framework, future developers can conveniently generate new tests to ensure AATOM is an insightful simulator.

Contents

1 Introduction

Agent-based simulation models have become increasingly popular among researchers in the last decades, and the development of robust testing techniques for such models can even lead to them becoming a revolution in modelling (Bankes, 2002). Robust testing can achieve this by verifying that these simulation models behave as expected, indicating that these models can adequately simulate real-life scenarios. The Agent-based Airport Terminal Operations Model simulator (AATOM) is an example of such a simulation model which aims to accurately model entire airports by simulating the buildings and all the people inside, including their interactions with each other and the environment. However, the AATOM simulation suffers from a severe lack of testing, which reduces the confidence in the simulation results that it provides.

The main goal of this report is to discuss the choices made when creating the robust testing framework for the AATOM simulation in order to increase the confidence in its results. Additionally, an architecture proposal for the AATOM simulator will be presented and discussed in order to improve the extensibility of the simulator. The testing framework will be developed using the generic testing framework presented by Gürcan et al. (2013), which splits the simulation model into *micro*-, *meso*-, and *macro*-levels. First, the individual agents will be tested in the *micro*-level. Once the this level has been thoroughly tested, interactions between groups of agents can be tested in the *meso*-level in order to verify that the interactions between different types of agents have been implemented correctly. Finally, the system as a whole can be tested through the *macro*-level tests. However, the testing of emergent behaviours in the *meso*- and *macro*-levels is limited to a certain degree, since it is hard to identify the expected outcomes in many scenarios due to our lack of expertise in the field of aerospace engineering.

The rest of the report will be structured as follows: Chapter 2 will present a detailed analysis of the AATOM simulation and the challenge of testing it. The requirements of the testing framework for AATOM will then be given in Chapter 3. Next, Chapter 4 will describe the development methodology used during the project. Chapter 5 will discuss ethical concerns regarding the project and give a detailed description of the design of the testing framework. How the testing framework has been implemented will be shown in Chapter 6. Section 6.3 will present our proposal for a better software architecture for the AATOM simulator itself. A discussion of both the testing framework and the architecture proposal is given Chapter 7. Finally, the conclusions will be presented in Chapter 8.

2 Problem Analysis and Approach

This section, specifically in 2.2, will describe the problems that are addressed by this project and why they are important to solve. Furthermore, in 2.4 an overview of the approach at solving these problems is presented. Lastly, in 2.5 the stakeholders of this project are outlined.

2.1 Introduction to AATOM

AATOM was developed in 2018 by Stef Janssen for his PhD thesis Janssen et al. (2019). It was developed after it was realized that air travel was growing to become the largest mode of international transportation, and at the heart of air travel lie airports. As airports become more complex due to the growing number of passengers, new innovations would need to be made, but conventional techniques have been nearly exhausted. Janssen realized that, "human behavior plays an essential role in modern airport operations", which meant that the next step of innovation in airports would have to be by simulating human behaviour through a simulator. The simulator would have to treat humans as autonomous agents, which means the only logical type of simulator this could be is an agent-based simulator. Agent-based simulators are already widely known and popular, and many existing simulation frameworks exist. However, this was not considered satisfactory by Janssen, who required that the environment be able to interact with the autonomous agents.

No existing framework currently exists that supports such a configuration, but this would be integral to an airport simulator, where the true subject under analysis were not the agents, but the airport itself. This created a unique situation; intelligent autonomous agents need to be simulated in a model of an airport, so that the efficiency (flow) and effectiveness of the airport could be analyzed. And on top of that, the agents not only interact between themselves, but also with their environment. Such an agent-based application or framework does not currently exist. This is where the idea of the Agent-based Airport Terminal Operations Model simulator was conceived.

A program like AATOM would have serious consequences for the airline industry. Currently there are airport traffic simulators in use, but these are not agent-based nor open-source, so they cannot offer insights to the same extent that AATOM could. With AATOM, an airport could add its own model to AATOM, and run the simulation with AATOM's interactive human agents. After the model is calibrated, any airport could start experimenting with new layouts, and discover a new model which is more efficient or more secure. Thus now models could be hypothesized, created, and verified much quicker, *and* at a much lower cost. Fortunately, this is not only hypothetical. Master students at TU Delft have been working with Schiphol Airport to investigate a more secure, faster flowing security checkpoint using an extension of AATOM. A UML diagram of their current extension can be found in Appendix B. This shows that AATOM is already being used to improve travel systems in real-life scenarios. In fact, currently, there are multiple extensions of AATOM under development, and this means there is increasing reliance on the fact that AATOM is sound and correct. This is where AATOM, under growing pressure, is slowly starting to crack, as two main issues have been bubbling up and coming to the attention of the Faculty of Aerospace Engineering. Firstly, it is completely devoid of any (automated) testing framework, and it is lacking in the extensibility it claimed to have. While AATOM has great potential, these issues need to be addressed promptly in order for AATOM to realize its full potential and become a reliable airport simulator.

2.2 Problem Overview of AATOM

The problems present in AATOM that this project aims at solving are grouped into the following six categories:

1. **Testing is limited to units of functionality**
   In order to be able to properly verify the model it's crucial that the System Under Test (SUT) is tested at various levels - building confidence with each. The tests present for AATOM do not satisfy this and thus do not provide the desired verification.

2. **Testing is sparse**
   Numerous sections of the code are not tested at all. In order to build the desired trust in the SUT each part of the code should be at least subject to unit tests. This makes the entire system more robust and prevents human bias from directing what should and shouldn't be tested. Furthermore, in order to be able to obtain confidence from a higher level test the individual components making up that test should already be tested individually. This further motivates a complete test suite.

3. **Testing standard is not defined**
   AATOM is a project used by many different people, each with their own distinct programming background. This means that without a testing standard in place tests will either be written and designed inconsistently or be neglected altogether. It should be clear what the testing requirements are and how to meet them.

4. **AATOM is not built using an existing ABM framework**
   No existing open-source framework exists that is able to easily implement both human behaviour and airport specific elements. Furthermore, a high degree of flexibility is required for the definition of agents, which was deemed restrictive in the existing frameworks. (Janssen et al., 2019) AATOM is

tailored specifically to its use case, which means it cannot leverage testing procedures designed for existing frameworks.

5. **Lack of flexibility makes functional extensions difficult**
   Two Master's students, Klemens Koestler and Didier van der Horst, that use AATOM for their theses rewrote significant portions of the codebase in order to be able to define their own custom security checkpoints. Not only is this a lot of work that is not directly related to their extension, it also means that projects that relied on the previous codebase would no longer work on this modified version. This clearly violates the open-closed principle and means that the current codebase is not *open to extensions* and thus can't remain *closed* while providing the required functionality, e.g., of a custom security checkpoint (Meyer, 1988).

6. **Lack of modularity complicates maintenance and testing**
   As mentioned above, extensions to AATOM have shown to require modification of the existing codebase. This creates a challenge for the test suite in place as existing tests may require adjustments and/or redesign. Existing tests should not have to change and should continue to pass when extensions are made to AATOM.

In order to solve the above problems the following four goals have been set:

1. **Design and develop a testing framework for AATOM**
   The testing framework should provide verification of the source code. Additionally it should be designed such that extensions to AATOM can easily be tested without having to change any of the existing tests.

2. **Test AATOM through the testing framework**
   A bottom up approach will be used to test the entire system. For the agent this would entail testing the individual components of the agent, the agent as a whole, multiple agents, and finally agents interacting in a realistic environment. Specifics are provided in Section 5.2.

3. **Facilitate use and maintenance of test framework**
   When extensions are added or new cases considered there will be a need to adjust and/or extend the test suite, this will be facilitated through guides and explanatory documentation. Furthermore, Continuous Integration (CI) could be setup on the AATOM GitHub in order to provide up-to-date testing metrics and further automate the testing procedure for further development.

4. **Deliver an architecture proposal for AATOM**
   A thorough analysis of the current architecture and its trade-offs will be made with the help of UML diagrams and case studies of how researchers have used AATOM in the past. Ultimately we aim to design an architecture prioritizing modularity, extensibility, and testability that is open to the (expected) extensions.

2.3 Introduction to Relevant Software Testing Techniques and Principles

Given that the wider goal of this project is to test a software program, it would be nice to introduce some important traditional software testing concepts. First of all, it is important to note that when we mention software testing, we refer to automated software testing. The main key is that with automated testing, while we still create the tests, they are run by a computer, meaning they can be run with no human effort any number of times. A large number of automated tests make up a test suite, and we divided our test suites according to the layers of the testing framework. Automated testing, can further be divided into different layers.

The first and most important is of these layers is unit testing. Unit testing tests a program through its individual units. What we can regard as a unit are the methods of a class, as they are the smallest pieces of code with their own indivisible functionality. Unit testing should comprise the biggest number of tests since units form the basis, and at Google 80% of the testing budget is spent on unit testing (Winters et al., 2020). For unit testing to be done effectively, they must have certain properties. Firstly, they must be specification-based, meaning that the tests assert or check that the program output matches the specifications. For this to be done effectively, testers *must* have access to the specifications of the program. Next, they should be structural, meaning that a large portion of the structure is explored. In a program with many conditional branches, a good test suite covers most branches, but well written code must also have a reasonable number of branches.

The next layer above unit testing tests the integration of different components, and it is called integration testing. In traditional software testing, integration tests test that different technologies and platforms can co-operate, such as checking that an application is able to store data in a database. To test these interactions, mocks are used very commonly, which are objects that externally act like a class, but internally they do very simple things (for example, a web service can be mocked to return an empty .html file plainly for testing purposes). Another very important feature that the source code needs to satisfy in order for both integration tests and unit tests to be doable is dependency injections, which is when a complex object can be given to a unit rather than a unit making the object for itself. Why is this needed? If dependencies can be injected, this allows us to work with mock dependencies, which makes testing much more manageable. In the context of AATOM, integration testing was about ensuring that the different *layers* of an agent could be integrated.

Lastly, it is also necessary to ensure the quality of tests. We already mentioned two properties (specification-based & structural) that indicate good test quality, but another highly effective method is through mutation testing. Mutation testing checks other properties of tests by slightly changing the source code itself, this is called mutating. Afterwards, tests are run on these mutations to see if the mutations cause the tests to fail. The proportion of mutations that are detected by a failing test determine the mutation score. A high mutation score correlates strongly with high quality tests.

Traditional software testing techniques and principles are useful when testing a program, and we discussed in what ways we can quantify the quality of tests as well as what properties the source code itself needs to have to be testable. Since unit and integration testing comprise the majority of all software testing, it is important to understand the foundations. However, agent-based simulations are in no way traditional software, and we will have to devise additional software testing techniques in order to properly test AATOM.

2.4 Solution Research and Approach

After considering the traditional java automated unit testing frameworks, and exploring two testing frameworks designed for agent-based simulation models, we decided on a unique approach that would combine the strengths and abilities of each. Our solution comprises of a combination of automated testing frameworks used within the scope of the Generic Testing Frameworks for agent-based models.

In this project, all development had been limited to Java, for which we decided on the tools JUnit, Mockito, and AssertJ. These three libraries/frameworks have been well established in the area of unit and integration testing. JUnit and AssertJ are the most versatile, and are used specifically for automating unit testing. Mockito, which provides mocking capabilities, specializes in integration testing. Mocking has served a useful purpose for testing entity-entity interactions by mocking one and observing the other. With the huge complexity of the program mocking was also very useful in lower scopes, where modules would be mocked internally. Combined with unit testing, we can test agent-agent and agent-environment interactions, which in the context of agent-based systems would be considered *micro-level* testing.

The generic testing framework by Gürcan et al., 2013 outlines a clear, bottom-up approach to testing agent based models. The model can be classified into three testable levels: *micro*, *meso*, and *macro*. The micro-level aims to deal with the most basic elements, in our case agents, individually. This would then also include mocking since mocking tests how an individual agent interacts with mocks. Next up, the meso-level is introduced to bridge the gap between micro- and macro-, which both deal with opposite extremes. The meso-level tests interactions (communication protocol) within small groups of agents, known as "sub-societies", which together may try to achieve a goal in the macro-level. It should also test whether long-term behaviours are achieved if necessary. The macro-level looks at the simulator as a whole, and tests it from the end-user's perspective. Mainly, it checks that the simulator sufficiently models the real world given the correct data, as well as *bad-weather* cases, to ensure it does not break given malicious or erroneous data. Stress testing also falls in the macro-level.

2.5 Internal and External Stakeholders

The testing framework that we are developing will have an impact on anybody who will work with expanding and studying AATOM in the future. Our sole direct stakeholder is our client at the faculty of Aerospace Engineering, who has asked us to develop this testing framework to catch problems in the design and to verify the correctness of the model. The client is seeking a testing framework in order to offer students a better starting point for their research in addition to improving the trust in the obtained results from using AATOM.

Closely related to the staff are our indirect stakeholders: Master's and PhD students who are currently working on expanding the AATOM simulator, as well as students who plan to expand it for their research projects and theses in the future. Most students working to expand the simulator have not successfully managed to use a testing framework to check their work for correctness due to the complexity of testing an ABSM, and this is compounded by the fact that the existing program lacks modularity, which has made it hard to test in the first place. This results in ad hoc solutions every time a research project is started, which impedes the output of research and insight into airports and the field of air transport. This project would benefit students who will as a result be able to focus more on the insights provided by the simulator in order to optimize airports.

3 Requirements of the Testing Framework of AATOM

This chapter explains the approach taken to write down requirements that this project entailed. First, we explain the elicitation methods that were used to fully understand the requirements of this project. Secondly, there will be further specification of the actual requirements. Lastly the explanation of the completion metrics of requirements, describing what needed to be done, such that they were marked as finished.

3.1 Requirements Elicitation

To begin working on a project it is required to correctly assess and distribute work that needs to be done. To correctly assess these problems, it is needed to first properly define what the problems are. As developers may not necessarily have the same vision of the project as the client, project requirements need to be explicitly settled with the client, so a well-defined problem space is found. For this to take place, the team can take multiple approaches. Before starting to work on the testing framework of AATOM, we have decided to first meet with the client, where we were presented with the overall overview of the AATOM repository. From this first meeting, we received the code-base of AATOM, and conducted further research into the subject of agent-based simulations. Most importantly, we established contact with master students that have expanded upon AATOM.

As these master students have worked with AATOM and are using it for their master thesis, they are the major stakeholders in this project, as our testing framework should serve students such as them to verify the simulation. That is why we have met with these master students and interviewed them on the basis of their expectations of AATOM and their view of its testing needs. We have discussed the problems they have encountered when trying to expand on AATOM such that we can use these observations in further improvements of the code-base.

These meetings with the client and the master students have also given us an insight into the inner workings of AATOM and some of its fundamental flaws, as described in section 2.2. These flaws have helped us analyze the architecture and its problems, which further lead to more concrete requirements.

Finally, the research provided by the client has provided insight into what a testing framework of an Agent-based simulator should consist of. This insight has further solidified the requirements as we were able to provide more descriptive and actionable requirements based on research, rather than rephrasing the requirements expressed by the client.

3.2 Requirements Specification

After obtaining a clear idea of what our actual mission in this project was, correctly categorizing requirements was the next logical step. To achieve a coherent and logical classification, we opted to use a primary division into functional and non-functional requirements along with a priority based ranking system.

The majority of requirements reside in the functional category, meaning they define what actually has to be done to achieve our project goals. This project could get derailed quite easily because of its abstract scope, therefore knowing which requirements had to be tackled first was crucial. Because of this we used the MoSCoW method to separate requirements into priority sub-groups based on the needs of our client and their feasibility during the allocated duration of the project. The Must-Have requirements represent a set of features that are vital to the success of our project and therefore must be completed. For example, the fact that we have to test AATOM on three distinct conceptual levels (the micro-, meso-, and macro-levels) is one of these requirements, along with affirming that each non-agent-based component in the model will have to be tested. The Should-Haves are designated to contain requirements that are not crucial goals, but are still worthwhile additions that we think are would benefit the quality of our product greatly. These include for example the creation of a standard testing format that can be easily paired with a tutorial that would explain this format to new potential users, in order for them to familiarize themselves with our code more easily. The Could-Haves are extra requirements that could prove worthy if and only if time allows it. Refactoring sections of the code-base that need it most based on our proposal is one these requirements. Lastly, the Won't-Haves represent functionalities that are good to look into when thinking about the future development of AATOM, but are outside of the scope of our current work on the project. For example, joining all the extensions of AATOM into one cohesive repository is one of them.

On the other hand, the non-functional requirements define other operational requirements of the system, and which tools and paradigms should be used to implement the system. Although these restrictions do not describe any features of our final output, they provide agreed-upon conventions and prevent scope creep issues, making them vital for the project. Some of these include the kinds of testing frameworks we should be using, the fact that our newly created testing framework will only take into account the original code-base of AATOM and will not encompass any of its extensions and that our refactor proposal should have extensibility, maintainability, and usability as its conceptual pillars.

3.3 Requirements Completion Metrics

With a list of requirements completed, it is important to come up with a proper metric to determine whether a requirement is fulfilled or not. To tackle this problem we have made sure to define clear definitions of completeness for requirements.

As most of our requirements are based around the testing of software, our first goal was to properly define the metrics that we would take into consideration while testing. These metrics should reflect the quality of the tests we write, and a certain threshold should be put in place such that it is easily verifiable whether something is tested or not. In the end, we have decided to take the full branch and condition coverage as the main indicator for unit tests, setting the threshold at 85%. Another important aspect to take into account is the test quality, rather than just its pure coverage. Therefore we have chosen to consider the mutation score to properly address the quality of our tests. The threshold of the mutation score was set at 80% per package. These thresholds have permitted us to clearly define when a package is considered to be fully tested, thus allowing a numerical evaluation of the completion of a testing requirement.

A secondary objective of this project is to improve the overall structure of AATOM. Requirements connected to this goal, are quite complex to define, as these improvements are not measurable in an objective manner. Therefore the chosen approach to these issues ended up being a discussion with the client on whether they consider the proposal of the new architecture adequate. This, unfortunately, means that there is no good numerical metric that was setup to define the completeness of the software architecture proposal requirement, and thus needs to be dealt with in a trial-by-error manner. Of course there are still certain possible sub-requirements that can be decided upon before meeting with the client, for example the structure of the proposal, or numerical analyses like time analysis or space analysis.

The completion of the requirements is dependant on what the client imagines behind the requirements. It is important to capture these visions as closely as possible, when writing down the results of requirements engineering, however it is as important to check near the end with the client whether everything is still how they envisioned it. That is why near the end of the project a meeting is to take place, to properly go over the requirements and mark them as complete, so a definitive confirmation is given by the client themselves.

4 Development Methodology

4.1 Process

Scrum. We have decided that we will be creating sprints for each week, where we decide on what our plans are for the upcoming week and what issues we need to cover. Each sprint will be represented by a milestone on GitLab so it is easy to keep track of our process. Sprints will be created on the start of each week during a meeting with the entire group based on issues found on GitLab. We strive to meet up with the client at least every two weeks to present our progress. On every Friday we will hold a reflection session as a team, where we will reflect on whether we have managed to satisfy the goals we have set up for ourselves. The Scrum Master is responsible to ensure that the sprints keep up with our project schedule. The role of the enforcer is to ensure that everyone finishes their part of the sprint on time for others to review.

In order to ensure the quality of our work and reduce the chances of mistakes and malpractices, we have made the following agreements over our development process:

1. Each issue should be worked on in individual branches off of the development branch. The naming of these branches should be done using GitLab's automatic naming, which is based on the issue name and number.
2. Issue branches are to be submitted through a merge request back into the development branch once deemed complete by the assignee. These merge requests should also be made using GitLab's functionality provided in the issue view, ideally using "Create merge request and branch". This will create a "Draft" merge request. Once work is ready for review the "Draft" label should be removed by the issue assignee.
3. Two approvals of group members not directly involved in the issue are required before the branch can be merged into development.
4. All methods and tests developed should have descriptive Javadoc.
5. Each newly added code has to abide to specific code rules determined by the GitLab pipeline. The pipeline checks for the style of the code, and its correctness.

Reviewing. We have decided to assign each other a reviewer. This decision was made to ensure that no-one works alone and everyone has someone to turn to when a discussion about an issue is needed. At the start of each week while creating the sprint, we will also create a randomized reviewing circle where everyone is assigned to someone as a reviewer.

4.2 Tools

GitLab. We will use GitLab to track our process and our code. The `Issues` tracking provided by GitLab will be utilized as a list of task we need to complete and the Milestone feature will be used to represent sprints. GitLab supports the creation of a back-log with its `Requirements` feature, a list where it is possible to write down all necessary requirements collected by the group.

IntelliJ. We have agreed to IntelliJ IDE provided by JetBrains as it makes it easier for the group to work in a similar environment. We will also be using the IntelliJ plugin CodeWithMe which provides the team with the possibility to work on the same version of the code at the same time.

Discord. We will use Discord for group meetings and group communication. We have chosen Discord as everyone in the group is already familiar with it and no-one in the group has encountered major technical difficulties with this platform.

Mattermost. Mattermost will serve as our main communication channel with the TA. This also comes with general channels used by the course staff for announcements or discussion between teams.

Microsoft Teams. As Microsoft Teams has been integrated with the TU Delft accounts, we find it easiest to use Microsoft Teams for professional meetings outside of our group, like the client or the coach. Microsoft Teams also provides a calendar in its interface which we will use to track our meeting schedule.

4.3 Schedule

The schedule can be found in Figure 8 of Appendix G.

5 Designing an Extensible Testing Framework

In this chapter the design of the testing framework will be discussed. This will be done by first taking a deeper look into the most prominent ethical aspect regarding the testing framework. After the main ethical issue has been discussed, a design for the testing framework will be represented which will also take into account some of the ethical concerns from the first section.

5.1 Ethical Implications of the Testing Framework

A significant ethical issue in the area of software testing is accountability. According to McGrath and Whitty (2018), accountability can be defined as "the liability to ensure that [a task] is satisfactorily done" (p. 687). In the context of software engineering, accountability refers to the extent to which one can be blamed for the consequences of a system failure. It should be noted that this differs from responsibility, which is "the obligation to satisfactorily perform a task" (p. 687).

A famous case of a critical system failure due to a software bug is that of the Mars Climate Orbiter (MCO). At one point during the mission, contact with the MCO was suddenly lost and could not be re-established. After some investigations were conducted, it was concluded that the crash was due to missing conversions from imperial units to metric units (NASA Mishap Investigation Board, 1999). Included in the Phase I Report of the Mars Climate Orbiter Mishap Investigation Board is the assessment that "There was inadequate training of the MCO team on the importance of an acceptable approach to end to end testing of the small forces ground software." (p. 24). This raises questions such as to what extent may the MCO team be held accountable for the mishap, and to what extent this accountability falls on the management of the MCO project.

Similarly to the MCO project, inadequate testing can result in a failure to discover critical issues in the AATOM model. Given that real clients, such as Schiphol airport, make use of AATOM, such issues may impact airport processes in the real world. Schiphol conducts field experiments inside the airport based on promising AATOM results, which may lead to wasted resources when these are based on incorrect results.

For this reason, accountability issues also arise in the testing of AATOM. The main stakeholders who may be held accountable for such issues are the engineers at Schiphol, the creators of the AATOM model, or us, the creators of the testing framework. Below we discuss each in more detail.

Firstly, Schiphol itself could be held accountable to some degree in case field experiments lead to wasted resources. Although some of these failed field experiments were set up due to promising, but potentially incorrect, AATOM simulation results, Schiphol itself decided to conduct these experiments even after considering the benefits of successful experiments against the drawbacks of failed experiments. So, it could be argued that Schiphol also bears some of the accountability for making the decision to conduct field experiments based on AATOM results. Additionally, security experts at Schiphol are equipped with the knowledge to evaluate if potential changes to the airport may pose a danger to passenger safety, and could therefore bear some accountability for approving such changes.

On the other hand, by arguing that AATOM or its implementation do not correctly simulate real airport processes, the creators of AATOM could potentially be held accountable for issues arising from it. As the authors of the underlying model, they are both accountable and responsible for its correctness. Furthermore, because AATOM has received interest from clients such as airports, the creators now have a larger responsibility for making sure that the model behaves correctly, since it could have an impact on real, security-critical airport processes. While the severity of the consequences increases the level of responsibility, their level of accountability does not change, since they still bear the same liability for the results produced by AATOM.

However, it could also be argued that such problems, especially when it comes to the actual implementation of the model, should have been discovered through thorough testing. This would imply that some of the accountability can also fall on us, since we are creating the testing framework which verifies the implementation of the model. It is important to note that, while we are verifying the model, we are not validating it because we do not possess the required domain knowledge in the field of air transport & operations. Thus, if it appears that the implementation of the model is incorrect, then a part of the accountability falls on us for not testing the software thoroughly enough. In case the implementation of the model appears to be correct, but the simulation results are still incorrect, the creators of the model should be able to be held accountable for a certain part. Still, we are responsible for bringing up any issues in the model if we do find them, as we have the responsibility to report such issues. These could range from incorrect assumptions to obvious (negative) biases in the model.

The discussions above attempt to establish general guidelines as to who can be held accountable in certain cases. However, determining who should be held accountable for problems, and especially to which degree, is difficult. Many different factors play a role, and it is not easy to determine the extent to which one's actions contributed to the occurring problems. Even though determining accountability is a difficult matter, it is still of importance to consider this issue in this case. When designing and implementing the testing framework we should make sure the entire model is tested thoroughly, especially the more important modules of the software. Doing this properly should help reduce the chances of issues occurring in the future, and thus thus lower the chances of such accountability questions being raised.

5.2 Testing Framework

The design of testing framework takes into account the primary objective of thorough testing at multiple levels of abstraction. This section provides a more in-depth look at how this will be approached. At the lowest level, we are applying unit testing using JUnit. In the case of the agent, testing targets various parts of the 3-layered human agent architecture outlined in Figure 7 of Appendix F. Specifically, we have identified two levels of testing specific to individual agents:

1. Testing individual agent modules using unit testing.
2. Testing interactions between the layers and modules using integration testing.

Our main testing strategy makes use of the *micro*-, *meso*-, and *macro*-levels categorisation detailed by Gürcan et al. (2013) and the layers above are considered part of the micro-level. In the context of AATOM, these levels can also be summarised as follows:

1. Micro: testing individual agents and their behaviour. This includes testing of sub-agent modules and layers as described by the two levels above, and the agent as a whole.
2. Meso: these tests span the gap between micro and macro levels - leading to a wide variety of test cases. The primary focus is to test subsections of AATOM which, when grouped together, support the macro level tests. Components that are not under test should be mocked. Examples include:
   (a) A passenger and an operator agent interacting at the security checkpoint.
   (b) A small group of passenger agents queuing.
   (c) A small group of passengers in the waiting room with a limited number of available seats.

   A visual of these levels for a specific simulation instance is given in Figure 5 of Appendix D.
3. Macro: multiple agents in a real (non-mocked) environment. These tests are focused on full airport terminal and agent simulations. The number of agents used for these tests is expected to be larger than that of the meso-level tests.

6 Implementation

This chapter goes into the details of the products delivered, namely the testing framework and the architecture proposal. The testing framework will be broken down into it's levels proceeded by how these levels combine to create trust in AATOM. The architecture proposal section breaks down the existing problems and the considered solutions followed by an overview of the architecture proposal itself.

6.1 Testing Framework

This section provides a detailed overview of the implemented testing framework for AATOM. First, a look is taken at the implementation of the micro-level - discussing the approach of the creation of unit tests and integration tests in addition to the tools used to complete this layer of testing. Once having understood the micro-level, the meso-level is presented together with examples of tested properties and how it was assured that the tests are independent. The third subsection deals with the implementation of the macro-level. Namely, what kinds of properties are tested and how the tests differ from those of the meso-level.

6.1.1 Micro-Level

The micro-level contains the lowest level tests, namely tests that are concerned with small pieces of the codebase. This level serves as the foundation for the testing framework and is the first step in building up the verification of AATOM's model.

There is no available specification for the specific class interfaces for AATOM and thus we resorted to creating "concepts" derived from our understanding of AATOM and the code itself. For example, given a method said to be responsible for getting the current activity of an agent a test is written assuring that upon setting the current activity of an agent proceeded by that method call we indeed obtain the originally set activity. Similarly a test is written to assert that if no activity is set for the agent that the result of the method under test is null. Alternatively, when the expected functionality of a method is not clear we turned to the AATOM research papers or those it referenced in order to understand what the method should be doing. If that still didn't provide enough clarity we communicated with our client to resolve the uncertainties. For example, one of the movement modules, HelbingMovementModule, has particularly complex methods which implement the social force model proposed by Helbing et al. (2000). In order to test these methods effectively we wrote tests based on the social force model and thus verified that the methods in AATOM coincide with it. All in all, the micro-level tests were written based on concepts formed from both the code and the available literature in order to avoid simply writing tests based on the exact implementation of the methods.

Before getting into the specifics of the tests written for the micro-level it is beneficial to understand the limited testability of the original AATOM codebase. The factors that limited testability for the micro-level together with their found solutions can be grouped into the following:

1. Many cases classes had private variables without any "getters". This means that these variables can't be accessed by the respective test class. This makes it impossible to verify whether variables are set as expected. In order to circumvent this problem public getters were introduced.
2. Lack of dependency injection. This means that methods use hard coded objects resulting in the inability to mock these objects or pass a known instance in order to exercise a particular functionality. Adjusting these methods to allow for injection was deemed as altering functionality and thus was not done. Thus tests that would have ideally checked for equality of objects instead only verify the existence.
3. Classes may depend on instances of a lot of other classes. For example the EmotionModule has instances of the MovementModule, ObservationModule, Map, AatomHumanAgent, PlanningModule, DecisionMakingModule and the AnalysisModule. In order to isolate tests to only test the functionality of the MovementModule all of the other class instances are mocks with mocked behavior.

The above factors slowed down the testing of the micro-level as extra work was required. It must be noted that care was taken to make sure that none of the changes alter the functionality of the model - e.g. introducing getters was deemed fine while injecting dependencies was not.

The developed test suite for the micro-level consists of two kinds of tests: unit-tests and integration tests. Their definition is described in (INSERT SECTION). Similar to how the micro-level provides the foundation for the meso-level so do the unit-tests for the integration tests. The unit tests are limited to testing logic of a single class, examples of the kinds of verifications include:

1. Constructors and initialization methods should set variables correctly.
2. Getters and setters should get and set the correct fields, respectively.
3. Methods should behave in the way that the concept is defined. For example given the concept that when an agent is in the waiting area it should be in a relaxed state we expect that the method getIntensity of the BasicPassengerEmotion returns zero, representing a relaxed state, when an agent is at the waiting area.
4. That methods are called on dependencies with the expected frequency. For example when an agent is updated it should cascade a single update call to it's respective strategic model, tactical model, and operational model.

Particular importance was given to making sure that the way tests are written is consistent and clearly defined. Not only is this important to assure for the tests we write but also to facilitate the building upon and maintenance of the test suite. Each test's name is of the form "outputIfInputTest". This achieves two goals. Firstly, each name clearly describes the intent of the test. Secondly, test writers are forced to focus their tests on specific input-output cases and thus naturally prevents highly complex tests. All assertions for the tests should either use JUnit or AssertJ. The reason for the acceptance of both is that JUnit is Another convention followed is that all of a class' dependencies should be mocked using Mockito. By mocking dependencies the functionality is explicitly defined within the test suite and does not change if changes are made to the mocked class. Furthermore, tests for class A won't start failing if a bug is introduced into class B, as desired from a unit test. When it is necessary to test a method on a wide range of inputs parameterized tests should be used. This means that a single test method is evaluated for each element of a provided a list of parameters, for example pairs of inputs and expected outputs. This removes the need for numerous tests with identical logic making it easier to write and read the resulting tests. Finally in order quantify the quality of the written tests each class' test suite should satisfy the following metrics:

1. 85% branch coverage as determined by JaCoCo[1].
2. 80% mutation score as determined by PITest[2].

6.1.2 Meso-Level

The meso-level builds on top of the micro-level. Meso-level tests go beyond a single agent and are focused on properties of the simulation. This means that unlike for the micro-level tests, the tests of this level require simulations to be set up and ran. We have developed the SimulationEnvironmentBuilder, seen in (INSERT FIGURE), in order to facilitate this. This class allows the tester to create simulations in a variety of ways. For example one can use a pre-defined "default" entrance area, supply the coordinates of it, or directly pass an instance of the entrance area. The same principle applies to other areas such as checkpoints, check-ins, gate areas, etc. All in all this class greatly simplifies the creation of specific simulators. That being said the simulator alone isn't enough as simulation level tests require the knowledge of what has happened in the simulator. This has been achieved by creating a "log" or a "trace" of the states of the simulator at each time step. The log is created through the AgentLogger class which is to be instantiated through the AgentLoggerBuilder, (INSERT FIGURE). The builder pattern was used for the logger as different tests require different pieces of information to be logged. For example, for tests relating to the flights one may want to have the hash of flight that each agent desires to board while this is not relevant when testing the behavior of agents in a restaurant facility. The builder makes it easy to attach log components in a modular manner and minimizes the strain of writing and reading from disk. There are also a multitude of useful methods for the creation of meso-level tests defined in MesoUtility. These methods include the setting up of a simple simulation with a single method call and data extraction methods leveraging Tablesaw[3] for log processing. By leveraging the SimulationEnvironmentBuilder, AgentLoggerBuilder and MesoUtility the tester is able to focus solely on testing the property of interest and greatly improves the consistency between meso-level tests.

Following the procedure of building up confidence in the model, we have built up the meso-level tests as well. At the lowest level each area apart from the one of interest for the test is mocked. For example when evaluating the behavior of an agent at a checkpoint all other areas except the checkpoint are mocked and thus the results of the test depend solely on the behavior of the checkpoint. Building upon this: the same simulation configuration is instantiated yet all mocked areas are replaced by real instances - the same property is tested but now in a more complex environment. This way we can gain confidence in each individual area. Furthermore, the cause of failing tests becomes much more apparent. Returning to the checkpoint example: if the test for the simulation with mocks passes while the one for the simulation without mocks doesn't it would be self evident that it is not the checkpoint that is not behaving as expected and instead is something else. As noted above the SimulationEnvironmentBuilder has methods that accept instances of areas directly making the injection of mocks trivial. Examples of the kinds of properties tested at the meso-level are:

1. Agents that are not checked in go through the check in area.
2. Each seat is only sat on by one agent at a time.
3. Agents should end up at the gate of the flight that they have.

[1]https://www.jacoco.org/
[2]https://pitest.org/
[3]https://github.com/jtablesaw/tablesaw

These properties verify that multiple agents in a simulator behave as expected for specific situations and can be used together in order to verify that the agents behave as expected throughout the simulation as a whole.

6.1.3 Macro-Level

The macro level is the most outer layer of the testing framework that tests the full simulation in its process, because of this property, it is the layer that is the easiest to grasp for students that were not involved with software testing before. This makes it important to make sure that the process of writing new tests is as easy as possible. Because of this, while we were creating properties for the macro level to test out, the consideration of what exactly would future users like was on the top of our list. Based on previous interviews and extensions of AATOM created by other students, specifications were established about how to approach testing the macro level, as most students tend to check whether their model follows certain distribution for certain activities. For example an important test would be to assert that during their simulation the check-in activity follows a specific distribution. This has been our core approach, when creating test for the macro level, the ability to extend it easily and assert specific time distributions for specific simulations, while checking the simulation as a whole with some meso-level assertions, but with all systems fully inplace.

To obtain extensibility and ease of use of the framework on the macro level, we have provided the users with a framework to assert that certain properties follow a specific distribution. With the distribution obtained from running the simulation and analyzing the logs, we were able to construct a sample distribution, which is then compared to the expected distribution. To compare two distributions the Kolmogorov Smirnov test is used (Dodge, 2008), which permits us to effectively compare whether a sample distribution follows a specific, pre-defined distribution. With these comparisons, we are able to create assertions for specific properties, like check-in times, and test whether these properties do hold for a specific simulation. With these assertions it is possible to verify whether the implementation matches the expected behaviour.

To properly test all the properties of the macro level, we have created a list of expected behaviours, as seen in (REFERENCE T.B.D.). The majority of these were tested with the aforementioned method of comparing distributions. Some of these properties were tested in a similar way like the properties of the meso-level, that is with extracting specific information from the logger and checking whether it follows certain conceptions. When a new test is created a simulation is built using the Simulation Environment Builder which UML is in (REFERENCE T.B.D.). The simulation is run with the custom Logger that can be created using the Agent Logger Builder (REFERENCE T.B.D.) and lastly to extract data from the created log a Log Analyzer (REFERENCE T.B.D.) is used to allow complete extendable and customizable nature of these tests.

With all of this framework setup, we have decided to create 4 sub-parts of the Macro level to test. The first part is the Distributions tests, where we test specific distributions in the model, like the time needed for a passenger to drop of their luggage. Secondly, we have created a list of universal properties, that should hold after the end of any simulation (REFERENCE LIST T.B.D.). We run multiple simulations that differ very little from one another and we expect this set of properties to still hold. The differences between the simulations may include things such as slightly changing the positions of areas, the sizes of areas, or the rotations. Thirdly, there are certain tests called stress tests, which are aimed to test how the simulation runs in extreme circumstances. Examples of such tests might be adding too many agents to a simple airport, or restaurant being unreasonably massive. Lastly, there are tests that check the parameters. These tests are aimed to ensure that all valid parameters run the simulation normally, while all invalid parameters throw an exception in an organized manner. These four layers of tests have permitted us to create a robust framework for tests, which is easily extendable thanks to the aforementioned classes we have created.

6.2 Code Quality assurance

To ensure that our code is of the best possible quality we have taken several steps to automate the process of quality assurance. First, we have added Gradle to the repository, which originally used Ant. This was due to our experience with Gradle and our inexperience with Ant. Gradle also supplied us with plugins we could use to test our code quality. As plugins, we have used checkstyle and PMD to perform static analysis of the code that we wrote (only the regarding testing framework, rather than the entire repository), which provided important feedback about our code. PMD was used to check the most important coding violations as well as proper documentation, as that was our number one goal. Checkstyle was used for better code quality, where we used custom rules to check specific qualities in our code.

During the creation of the testing framework, we have ran into multiple problems where classes were not testable and required small changes such that we can test them efficiently and reliably. These changes consisted mostly of adding getter and setter methods, or changing method access rights. Even though we have manually made sure that the simulation was not affected by these changes, we still needed an automatic, unbiased way to assure the correctness of the simulation. Our first addition was to the pipeline that would run the main simulation for one hour of the simulation time and assert that no exceptions were thrown, assuring that no changes would break the simulation. However, this was not enough to assert the correctness of the functionality. Therefore, when we tested the macro level, we created the tests in such a way that we could easily test the functionality of the simulation, as we could check the results of the simulation before it was changed and after. If the results were significant, this would indicate a major change in the simulation itself. For example, we could setup a test to assert that the time to get to the gate area in a specific simulation is following a distribution we observed in the original implementation, and if that is no longer true, we could assume that some of our changes led to a deeper change in the simulation.

6.3 Architecture Proposal

When it comes to the software architecture proposal, firstly the problems with the current architecture will be discussed. Following this, we will summarize our proposal of the new architecture in greater detail.

6.3.1 Architecture Analysis

The architecture of AATOM can be analyzed in two different way; one is based on the architecture proposed in Janssen et al. (2019), and the other one is the actual code-base, to see how it has been implemented. Unfortunately, these two do not correspond to each other, which makes the analysis of the architecture harder, as no proper documentation exists for the current implementation. First, the architecture proposes an architecture that would be close to the layered architecture, where agents are formed by three layers which communicate between each other in an independent manner. The proposed layered architecture can be seen in F and D. This architecture divides the agent into 3 layers based on the Agent-based modelling principles. Upon interaction, all agent observations are pushed into the operational module, which then get interpreted in the interpretation module. Based on those interpretations, agent forms certain beliefs, which than turn into goals and activities. These goals and activities then get processed by the reasoning module to create solid plans which the navigation module can follow and feed the actuation module with specific actions that the agent should take. This makes sure that the agent forms beliefs based on its surroundings, and with some if its goals it will act in the interest of achieving these goals. This covers the agent formation, however based on our findings in extensions of AATOM, most extensions create new environments for existing agents. This does not have a specified architecture, so we had to only base our opinions on the implemented model. This creates the major issue of the environment not being extensible, as a lot of logic corresponding to activities linked to these environments can be found in the environments themselves. This makes students that try to extend AATOM further worsen the situation and create a shortcut such that their extension works, which resulted in a lot of code duplication from creating classes that are similar just slightly different (like Restaurant and Restaurant2) or hard coding specific behaviours into classes which were intended to be a general class.

When creating the proposal for the new architecture we have split it up based on different levels. The first level is the overall change of the passenger, that is how the layers communicate with one another and how the agent decides how to act. Secondly we created an architecture for each separate layer, such that we could make these layers more extendable and generalised. Lastly we have split the individual modules and environment as its separate part, where we could apply some design patterns to improve modularity and complexity of classes.

6.3.2 Final Architecture Proposal

For the architecture of an agent, we have decided to modify the proposed architecture in Janssen et al. (2019). In the implementation we have noticed the major problems being the distribution of information. Each layer stores everything and modules communicate with one another with no restraint. That is why we propose a layered architecture, where a packet will be sent across the layers when an agent needs to be updated, with all the necessary information. First, the packet arrives in the operational layer, which adds new information and then proceeds to pass the packet to the tactical layer. The tactical layer then repeats that process and sends it to the strategic layer. Thanks to the packet, every layer gets all of the information it requires and only accesses information it needs. A UML diagram can be viewed in (REFERENCE T.B.D.), which better illustrates the new architecture. The layered property also permits to add new layers, or change one layer in particular without changing the entire implementation.

The individual modules needed to be changed such that they correspond to the layered architecture. This means getting rid of all dependencies that are not found on the same layer. Furthermore, the Activity module and the Goal module are conceptually equivalent in the current implementation, so we have decided to update the goal module to be formed by goals which can be formed by more goals or activities. In the current implementation there are no beliefs, interpretations or perceptions. This would have to change such that the layers can operate independently, as agents should be in full control of their actions rather than being forced to do a specific action based on the current activity. For example, the choice of a serving desk at check-in requires an agent to be assigned to the desk, rather than the agent finding an empty check-in desk and approaching it to check-in.

The environment has multiple changes that need to be added. Firstly, the inheritance should be used in a more modular way, rather than having an abstract class that has only one child. Secondly, there are multiple environments that are formed by other environments, for example a checkpoint is formed by a belt, detector, chairs, etc. This could be reflected better as a Composite design pattern as shown in (REFERENCE T.B.D.). Lastly, the environment, but also a lot of other parts of the codebase, can be replaced with external frameworks such as the Apache 3 framework for math distributions. These changes are made to increase modularity and decrease complexity of AATOM, as any classes that do not directly influence AATOM should be external.

7 Discussions

Throughout the project, decisions were made that determined the direction and final outcome of the AATOM testing framework. The following chapter discusses and justifies these decisions. It also includes a discussion on the quality and usefulness of the implemented tests, and recommendations for future improvements of the simulator, the testing framework, and the development process to be used for AATOM.

7.1 AATOM testing framework

The testing framework we developed consists of three levels: the micro-, meso-, and macro-level. Using such an approach was beneficial as it allows for finding bugs which manifest themselves in different levels of the code. The micro-level tests find bugs that occur within individual classes of the code and the agents as a whole, and may potentially explain bugs found in higher levels. The meso-level tests have shown to find bugs which were not captured in the micro-level. Such bugs are often related to issues happening when larger components interact with each other. Finally, the macro-level testing has found issues concerning overall system properties when running entire simulations, which are not captured in the meso-level tests since those generally focus on system properties at a smaller scale. A more in-depth discussion of each layer will be presented below.

The micro-level tests make sure the individual agents work as expected by testing the individual classes and the agents as a whole. We made sure to meet our testing criteria regarding the line & branch coverage, and the mutation scores. The mutation scores ensure that our test cases are of high quality. This is due to the fact that it shows the test cases are capable of detecting small mistakes or changes in the source code, indicating that the test cases will only pass on the correct implementation. Using this metric was useful, as the large amount of micro-level tests we created have helped us identify numerous problems occurring in the individual components of the agents. Such issues were related to hard-coded locations for agents to go to, which are only applicable to a single specific airport, or classes transitioning into invalid states, introducing the risk of runtime exceptions. Additionally, the micro-level testing highlighted numerous code smells such as highly coupled classes and overly complex methods. Finding these code smells proved to be useful when designing the architecture proposal for the new version of AATOM.

The meso-level of the testing framework deals with the properties of the interactions between agents with each other, and with the the environment. In order to help with testing these properties, a LoggerBuilder and and SimulationEnvironmentBuilder were created, as mentioned in Chapter 6.1.2. The addition of these two builders proved to be very helpful with creating new meso-level tests in a structured manner. The included documentation for using these two tools will allow future developers of AATOM to easily familiarise themselves with them and use them for maintaining or extending the testing framework. The meso-level tests created uncovered more bugs in the model. Some of these bugs were related to issues already found in the micro-level testing. One such example is the fact that agents will choose a very distant check-in desk when a nearby desk only has one passenger currently being served. However, we also encountered new, unexpected bugs occurring in these small simulations, such as passengers dropping off and instantly picking up their luggage before it reaches the luggage collection point. The fact that we discovered such new bugs in the meso-level indicates that the different levels of the testing framework indeed manage to uncover unexpected higher-level bugs.

The macro-level of the framework contains tests dealing with overall system properties and model calibration. The tests regarding model calibration check whether simulation results represent the expected outcomes according to real data. Even though we cannot actually execute these tests ourselves, since we do not know what the expected results should be, we decided to include the ability to test the simulation results against expected results into our framework. We believe that such tests can prove to be a useful addition to the testing framework, such that future developers can easily test some model calibrations. The stress tests included in the framework show that AATOM is capable of handling larger and more complex scenarios if necessary. This can prove to be especially useful for extensions of AATOM which aim to simulate larger airports or using even more complex agents within the simulations.

Regarding the testing framework as a whole, we find that it has achieved our goal of being maintainable and extensible. Everything in the framework is clearly documented, providing users who are unfamiliar with it with enough information to work with it. Furthermore, the testing framework follows a clear and logical structure where each level has their own designated folders. This makes it clear to future developers where to add new tests for their extensions, and they can use the supplied documentation to help them with creating new tests for their extensions within each level. Additionally, the builders we supplied will also aid the users in conveniently creating these tests.

7.2 Recommendations for further improvements to AATOM

The AATOM project can be improved from a number of different perspectives. Firstly, there is the architecture of the simulator itself. This has already been discussed in detail in Chapter 6.3, as it was in fact a deliverable proposed to the client. For this reason, it will not be discussed in this section. Secondly, there is the development process for AATOM. This mainly concerns recommendations to maintain the implemented testing suite, but also includes guidelines to ensure that the quality of the simulator code does not decrease. Lastly, there is the testing framework itself. Here, we will provide recommendations with regards to the further improvement of the testing framework. The following section explains these perspectives in more detail, and describes what changes we believe would contribute to the simulator.

The quality of the development process of a codebase is a contributing factor to the quality of the resulting code. It is therefore worthwhile to discuss problems with AATOM's original development process, and propose a better process. No automated testing was used when developing AATOM, which we believe is the leading cause of bugs found in the code. It is important that newly added classes are unit tested before they are considered functional, and this should be verified using code coverage. It is recommended to use the coverage requirements that were used for the testing framework in Chapter 6.1.1. Meso-level tests should be created whenever new functionality is added that involves multiple agents interacting with their environment. Macro-level tests should be added whenever new functionality has not yet been tested in the context of the entire simulator. It can also be used to verify that the output distributions of the simulator are within expected boundaries. For constructing these tests, the appropriate Builder classes from Chapter 6.1.2 should be used. Information is logged through either the agent or the environment, and then used to assert properties.

Lastly, we recommend further extensions to the testing framework of AATOM. As mentioned in Chapter 6.1.3, we have implemented a way to collect distributions that emerge from the simulation and compare them against expected distributions. This is done in order to enable model validation. Given that we were not given access to expected distributions in specific airports, we were not able to actually perform the validation step. This could be done in the future to ensure that the model is similar to the real-life data generating process it is modelling.

8 Conclusions

The main purpose of this report is to analyze and discuss the choices made when creating the robust testing framework for the AATOM simulation in order to increase confidence in its results. This testing framework is to be a permanent addition to the default AATOM implementation, continuously integrated into its repository, testing it thoroughly on multiple conceptual levels, derived from our research into agent-based model testing practices Gürcan et al. (2013); Padgham et al. (2013). An additional goal of this report is to propose a plan to improve and refactor the existing AATOM codebase architecture, aiming to make it more modular, extendable and testable, by utilizing tried and tested software engineering methods, practices, and to patterns. This proposal is made to make the repository more maintainable, easier to use and to facilitate future expansion.

8.1 Testing Framework

We have managed to create a testing framework that succeeds at testing AATOM as demanded while fulfilling the success criteria we set for ourselves. It does so on multiple conceptually distinct levels using a bottom-up approach, more specifically segmented into micro, meso, and macro-level test suites. Unit and integration tests of each individual class and component of AATOM comprise the micro-level, creating the majority of the tests of the framework. These tests helped find bugs that occur in individual classes of the code and in the agents themselves, bugs that were also potentially the cause of behavioral bugs in the higher levels. The meso level tests were much more complex in nature since they tested small-scale local properties of the actual simulation. These tests in turn found bugs that could not be caught when only testing the individual components themselves. Finally, the macro-level tested high scale global properties of the simulation, assuring the correct functioning of the agent-based simulator as a whole.

8.2 Architecture Proposal

The new architecture proposal consists of multiple levels of refactoring, which increase the modularity and decrease the complexity of AATOM. Firstly, we created a new architecture for each separate functional layer of the agent, such that these layers are more generalized and extendable. Secondly, the communication between the agent layers has been overhauled using a packet system. The way that agents decide how to act and perceive their environment has been modified in order to decrease coupling between classes. Lastly, the environment has been split from the individual modules as its separate part, such that we can apply design patterns. In addition to these changes, the proposal declares that any classes that have known implementations should be replaced by external libraries, effectively preventing redundant work by not having to reinvent the wheel by using well know Java libraries.

8.3 Recommendations

After analyzing and developing a testing framework for AATOM, as well as proposing an architectural redesign for its codebase, this report recommends three things. Firstly, to standardize the development process of AATOM to include our testing framework for the benefit of future expansion and current functionality. After analyzing the codebase and working with it daily, it was clear that most of the structural issues found and the bugs encountered were a by-product of an ”ad hoc” development process devoid of any thorough and educated testing. Thus, standardizing this process by testing every new addition on multiple conceptual levels by extending the testing framework provided by this project would be a great help in alleviating current and future issues. Secondly, the documentation for the AATOM codebase is lacking in many aspects, therefore updating and curating it more strictly would be worthwhile for developers working with it. A substantial amount of functionality is documented with either lacking descriptions or sometimes even false ones, making working with AATOM an unnecessary hassle since some parts don't adhere to software engineering conventions. Lastly, the AATOM architecture is flawed in multiple fundamental aspects making the simulator unwieldy from a modularity and extensibility standpoint. Utilizing the architecture proposal we created, most of these structural issues would be eliminated.

9 Glossary

Agent-based simulator/model. A model that simulates autonomous agents that interact with each other in order to predict large-scale outcomes of a system.
Extensible. A program that can have additional features and functionality add alongside the core of the application without requiring a heavy deal of work to change the original codebase.
Cyclomatic Complexity (CC). The number of unique simple branches that can be reached within a single method. A program with no conditional statements has a CC of 1, and each new possible branch adds +1 to the CC. In general, methods with a CC of 10 or more are difficult to maintain and 20 or more should be refactored or redesigned.
Mocking. objects that externally act like a class, but internally they do very simple things (for example, a web service can be mocked to return an empty .html file plainly for testing purposes)
Validation. Ensure that a system functions according to the needs of the end user.
Verification. Ensure that a system is functioning as it is supposed to, according to its specifications, i.e.,

what the programmer intended it to do.
Testing Framework. A set of code that provides the functionality to easily create tests for a program, and that contains tests of its own.
Unit testing. Testing a program through its individual units. What we can regard as a unit are the methods of a class, as they are the smallest pieces of code with their own indivisible functionality.
Integration Testing. In the context of AATOM, integration testing was about ensuring that the different *layers* of an agent could be integrated.

References

Bankes, S. C. (2002). “Agent-based modeling: A revolution?”. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7199–7200.

Curtis, S., Best, A., and Manocha, D. (2013). Menge. Available at `http://gamma.cs.unc.edu/Menge/`.

Dodge, Y. (2008). *Kolmogorov–Smirnov Test*. Springer New York, New York, NY.

Gürcan, Ö., Dikenelli, O., and Bernon, C. (2013). “A generic testing framework for agent-based simulation models”. *Journal of Simulation*, 7:183–201.

Helbing, D., Farkas, I., and Vicsek, T. (2000). “Simulating dynamical features of escape panic”. *Nature*.

Janssen, S., Sharpanskykh, A., Curran, R., and Langendoen, K. (2019). “AATOM: An agent based airport terminal operations model simulator”. *Simulation Series*. Held at the *51st Summer Simulation Conference, SummerSim ’19*; Conference date: 22-07-2019 Through 24-07-2019.

McGrath, S. and Whitty, J. (2018). “Accountability and responsibility defined”. *International Journal of Managing Projects in Business*, 11.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc., USA, 1st edition.

NASA Mishap Investigation Board (1999). *Mars Climate Orbiter Mishap Investigation Board - Phase I Report*. Retrieved from `https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf` [Accessed 8 May 2021].

Ozko, J., Tatara, E., and Collier, N. (2020). The repast suite. Available at `https://repast.github.io/repast_simphony.html`.

Padgham, L., Zhang, Z., Thangarajah, J., and Miller, T. (2013). “Model-based test oracle generation for automated unit testing of agent systems”. *IEEE Transactions on Software Engineering*, 39(9):1230–1244.

Winters, T., Manshreck, T., and Wright, H. (2020). *Software Engineering at Google: Lessons Learned from Programming Over Time*. O’Reilly Media.

Appendices

A Requirements

A.1 Non-Functional

1. The testing framework will use JUnit and Mockito testing frameworks.
2. The testing framework will be based mainly on the default AATOM implementation, and not on any subsequent extensions.
3. The testing framework will utilize an agent-based simulation testing paradigm as explained in Section 5.2.
4. The testing framework will be made intrinsically scalable and extensible, such that further developments on the AATOM codebase can be easily tested, without the framework becoming obsolete.
5. The project will be created in a manner such that it can become part of the AATOM repository (which is open-source) after the project.
6. The project will improve the scalability and usability of the entire codebase.
7. The testing framework will not change any of the functionalities of the AATOM simulator.
8. The architecture proposal will make use of well established software engineering design patterns.
9. The testing framework will be modular enough to ensure its function and reliability in newly created simulated environments.

A.2 Functional

A.2.1 Must Have

1. Create a list of expected behaviors and undesirable behaviors expected from the codebase.
2. The testing framework will test all non-agent-based components of the model, such as walls, with unit testing and integration testing if applicable.
3. The testing framework will test the agent models on a micro-level, meaning it will test the basic functionality of an agent by itself, based on agent-based unit testing detailed in Padgham et al. (2013). Further explanation of the micro-level can be found in Section 5.2.
4. The testing framework will test the agent models on a meso-level, meaning it will test the interaction of agents in specific subgroups. Further explanation of the meso-level can be found in Section 5.2.
5. The testing framework will test the agent models on a macro-level, meaning it will test the complex interaction of multiple agents on an environment-wide scale. Further explanation of the macro-level can be found in Section 5.2.
6. The testing framework will work on all simulation environments provided in the project.
7. Proposal of a new software architecture that improves the overall modularity and extensibility of the project as a whole.

A.2.2 Should Have

1. Create an automatic builder for basic unit tests, such that when new code is added the user is not required to write entire JUnit tests, rather just provide input variables and expected output.
2. Create a standardized test format that can be easily expanded.
3. Create documentation explaining how to expand the testing for users that will add new code to AATOM.
4. Fix newly found bugs in the AATOM software found by the newly implemented Testing framework. Certain bugs might require further knowledge of Airport Simulations, which would fall outside of our scope. Such bugs would be mentioned in the report, however, they might not be fixed.

A.2.3 Could Have

1. Refactor the codebase based on our Software Architecture proposal.
2. Replace the Ant builder system with Gradle for easier and more standardized use.
3. Integrate Math libraries into the codebase instead of custom made distribution classes.
4. Perform a time-analysis on the codebase to determine the bottlenecks of the system and propose subsequent changes.

A.2.4 Won't Have

1. Create an Airport-Builder GUI which would permit users to create new airports from the GUI, rather than hard-coding the precise positions.
2. Join all extensions of AATOM into the basic repository.

B UML Diagram of AATOM Extension Used by Schiphol Airport

AATOM on its own provides a great pool of functionalities but not a great realm of possibilities. Didier v.d. Horst and Klemens Koestler wanted to use AATOM to test different types of security checkpoint, but AATOM was expectedly not able to do so. So instead, AATOM was extended to allow for customizable security checkpoints. Extending AATOM was not the easiest task so a lot of new classes were created with the aim of allowing more modular object creation. This project shows both the potential of AATOM and the drawbacks of its inextensible design.

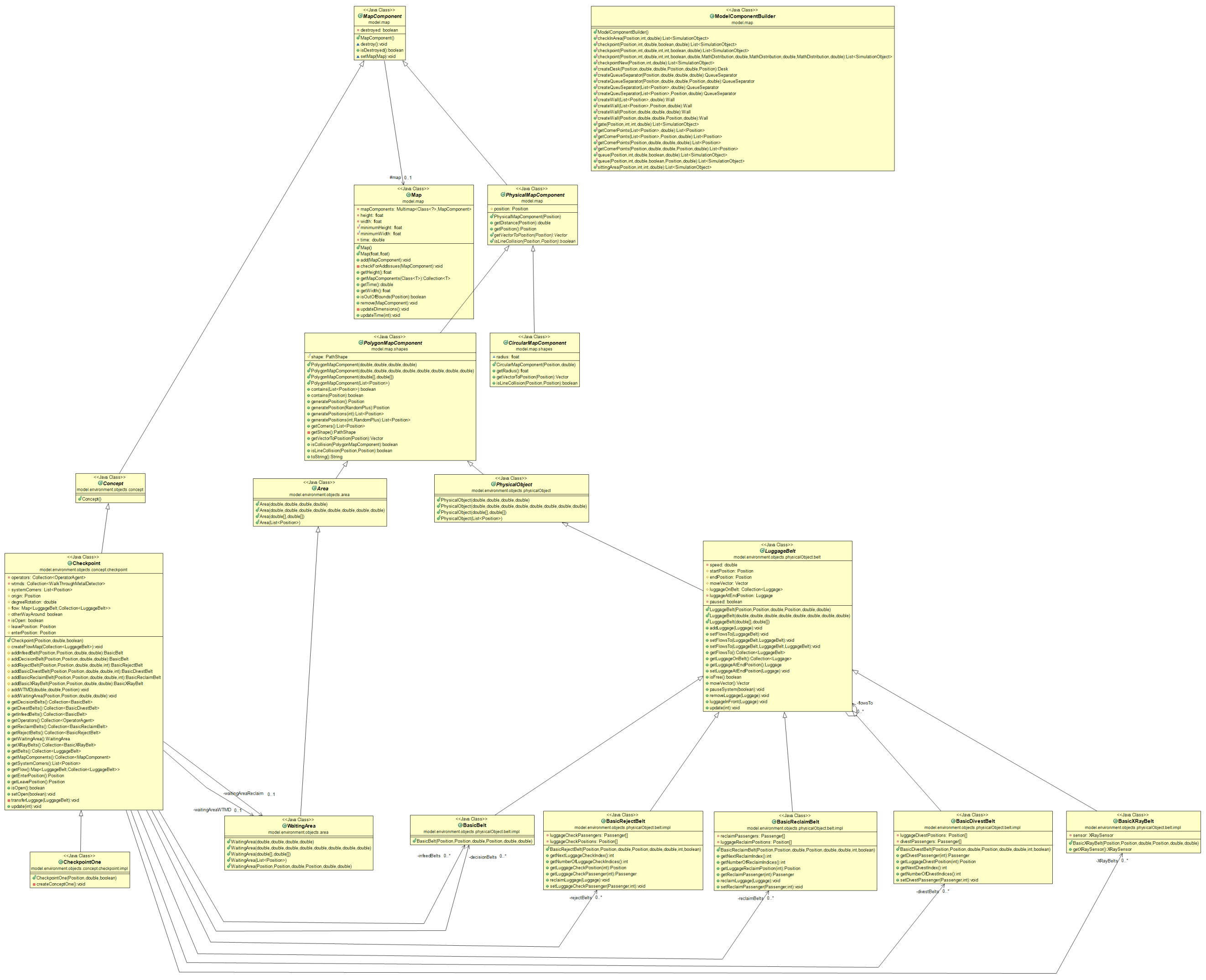


Figure 1: UML diagram of a new security checkpoint implemented by Didier v.d. Horst and Klemens Koestler.

C Agent-Based Simulation Model Frameworks vs. AATOM

The main feature that AATOM has that the other simulators (Repast & Menge) lack are: an interactive environment and different agent types. In AATOM, agents interact with different checkpoints in the terminal, which requires an extra step of complexity. This is missing in the other simulation models.

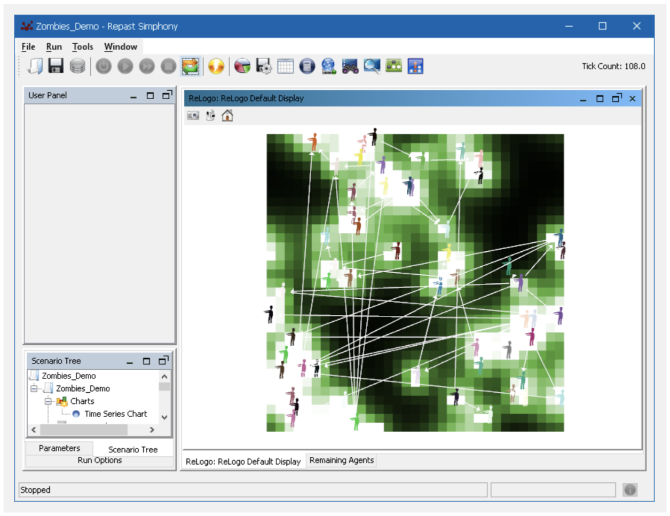


Figure 2: Visual representation of some Repast simulation presented in Ozko et al. (2020). Retrieved from `https://repast.github.io/docs/RepastReference/figures/relogo_zombies_run.png`.

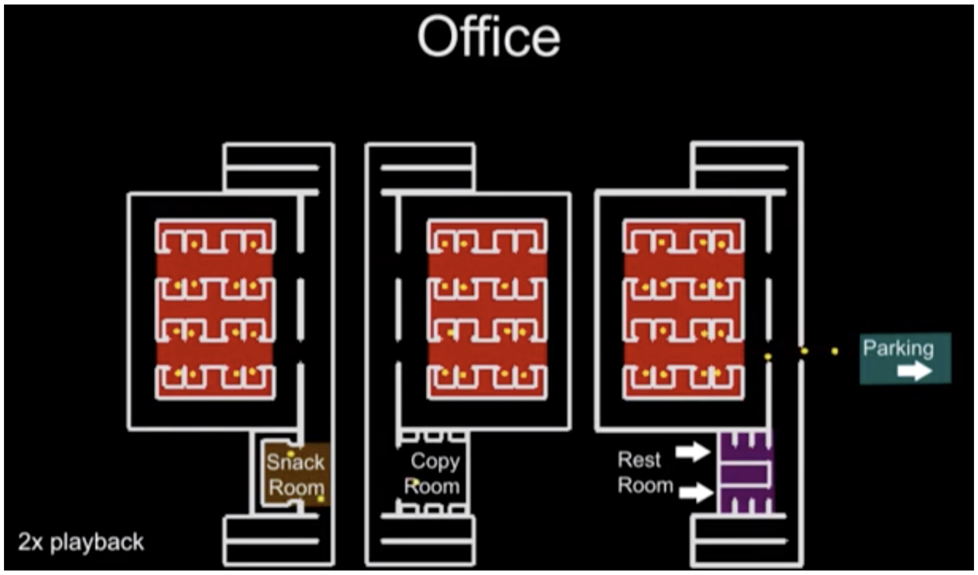


Figure 3: Screenshot of a visualised Menge simulation presented in Curtis et al. (2013). Retrieved from `https://www.youtube.com/watch?v=ofQdaNBFDp8&ab_channel=MengeCrowdSim`

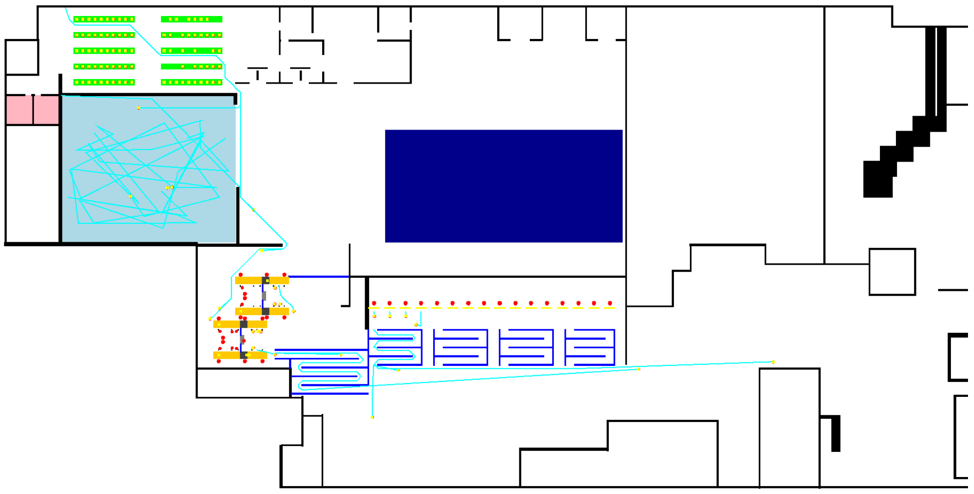


Figure 4: Visual example of an AATOM simulation instance showing the environment and the agents with their walking patterns.

D Envisioned Levels of Scope for AATOM

From micro to macro. The purple circles represent the micro-layer. This includes one agents (optionally with a mock agent). Above that in red is the meso-layer. This represents a group of two or more agents interacting with each other. At the very top is the macro layer in light blue, which encapsulates everything. Any changes to the environment as a whole are captured here.

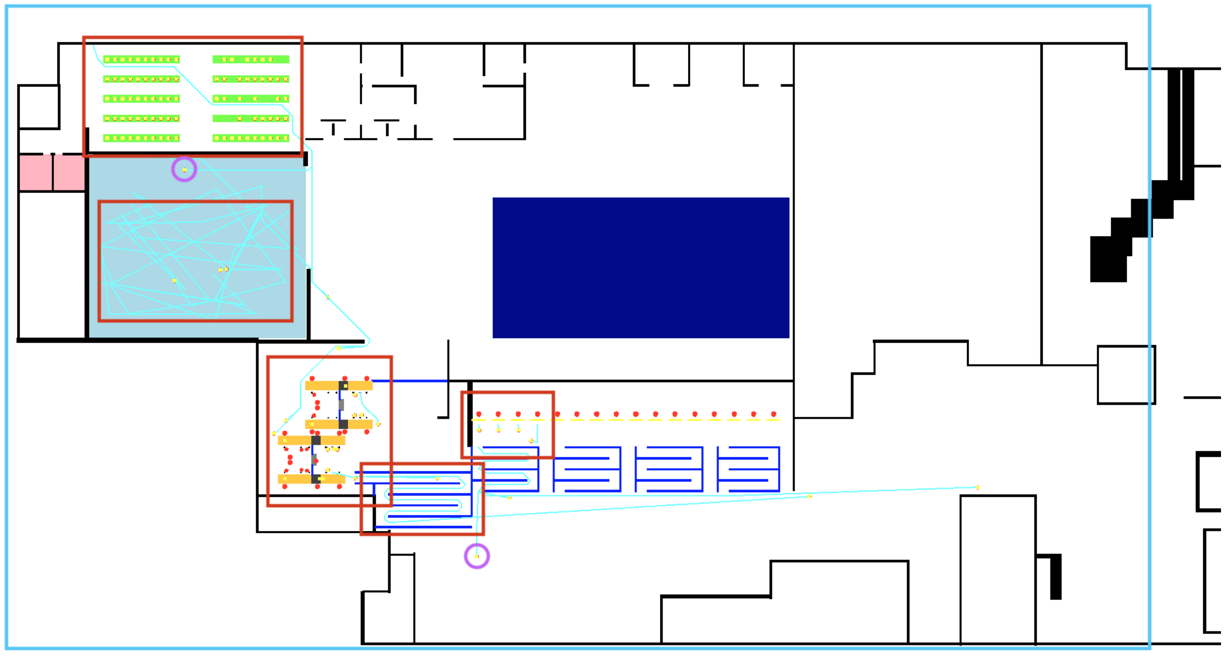


Figure 5: Visual example of an AATOM simulation instance.

E AATOM UML Diagram

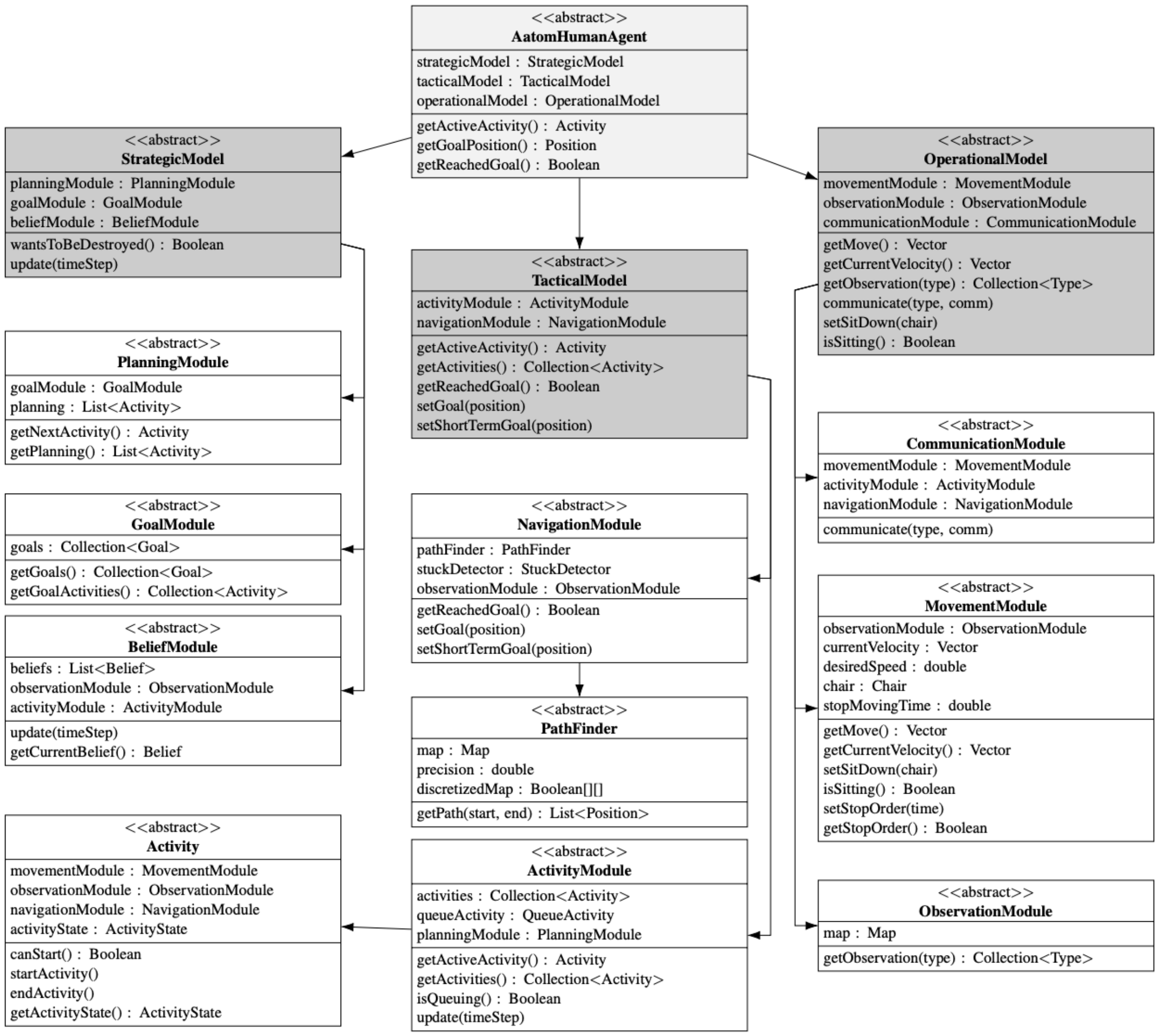

<<abstract>>
AatomHumanAgent
strategicModel : StrategicModel
tacticalModel : TacticalModel
operationalModel : OperationalModel
getActiveActivity() : Activity
getGoalPosition() : Position
getReachedGoal() : Boolean
<<abstract>>
StrategicModel
planningModule : PlanningModule
goalModule : GoalModule
beliefModule : BeliefModule
wantsToBeDestroyed() : Boolean
update(timeStep)
<<abstract>>
OperationalModel
movementModule : MovementModule
observationModule : ObservationModule
communicationModule : CommunicationModule
getMove() : Vector
getCurrentVelocity() : Vector
getObservation(type) : Collection<Type>
communicate(type, comm)
setSitDown(chair)
isSitting() : Boolean
<<abstract>>
TacticalModel
activityModule : ActivityModule
navigationModule : NavigationModule
getActiveActivity() : Activity
getActivities() : Collection<Activity>
getReachedGoal() : Boolean
setGoal(position)
setShortTermGoal(position)
<<abstract>>
PlanningModule
goalModule : GoalModule
planning : List<Activity>
getNextActivity() : Activity
getPlanning() : List<Activity>
<<abstract>>
CommunicationModule
movementModule : MovementModule
activityModule : ActivityModule
navigationModule : NavigationModule
communicate(type, comm)
<<abstract>>
GoalModule
goals : Collection<Goal>
getGoals() : Collection<Goal>
getGoalActivities() : Collection<Activity>
<<abstract>>
NavigationModule
pathFinder : PathFinder
stuckDetector : StuckDetector
observationModule : ObservationModule
getReachedGoal() : Boolean
setGoal(position)
setShortTermGoal(position)
<<abstract>>
MovementModule
observationModule : ObservationModule
currentVelocity : Vector
desiredSpeed : double
chair : Chair
stopMovingTime : double
getMove() : Vector
getCurrentVelocity() : Vector
setSitDown(chair)
isSitting() : Boolean
setStopOrder(time)
getStopOrder() : Boolean
<<abstract>>
BeliefModule
beliefs : List<Belief>
observationModule : ObservationModule
activityModule : ActivityModule
update(timeStep)
getCurrentBelief() : Belief
<<abstract>>
PathFinder
map : Map
precision : double
discretizedMap : Boolean[][]
getPath(start, end) : List<Position>
<<abstract>>
Activity
movementModule : MovementModule
observationModule : ObservationModule
navigationModule : NavigationModule
activityState : ActivityState
canStart() : Boolean
startActivity()
endActivity()
getActivityState() : ActivityState
<<abstract>>
ActivityModule
activities : Collection<Activity>
queueActivity : QueueActivity
planningModule : PlanningModule
getActiveActivity() : Activity
getActivities() : Collection<Activity>
isQueuing() : Boolean
update(timeStep)
<<abstract>>
ObservationModule
map : Map
getObservation(type) : Collection<Type>

Figure 6: UML Diagram of the architecture of AATOM. Reprinted from Janssen et al., 2019, p. 6.

F AATOM Agent

Modules, Layers, Agent. Figure 7 shows a diagram of an agent in AATOM. A module, in the codebase, is represented by a single class, represents a single aspect of decision-making, and has its own functionality. A layer represents an idea, which combines information gained by each module. Altogether, the layers form an agent.

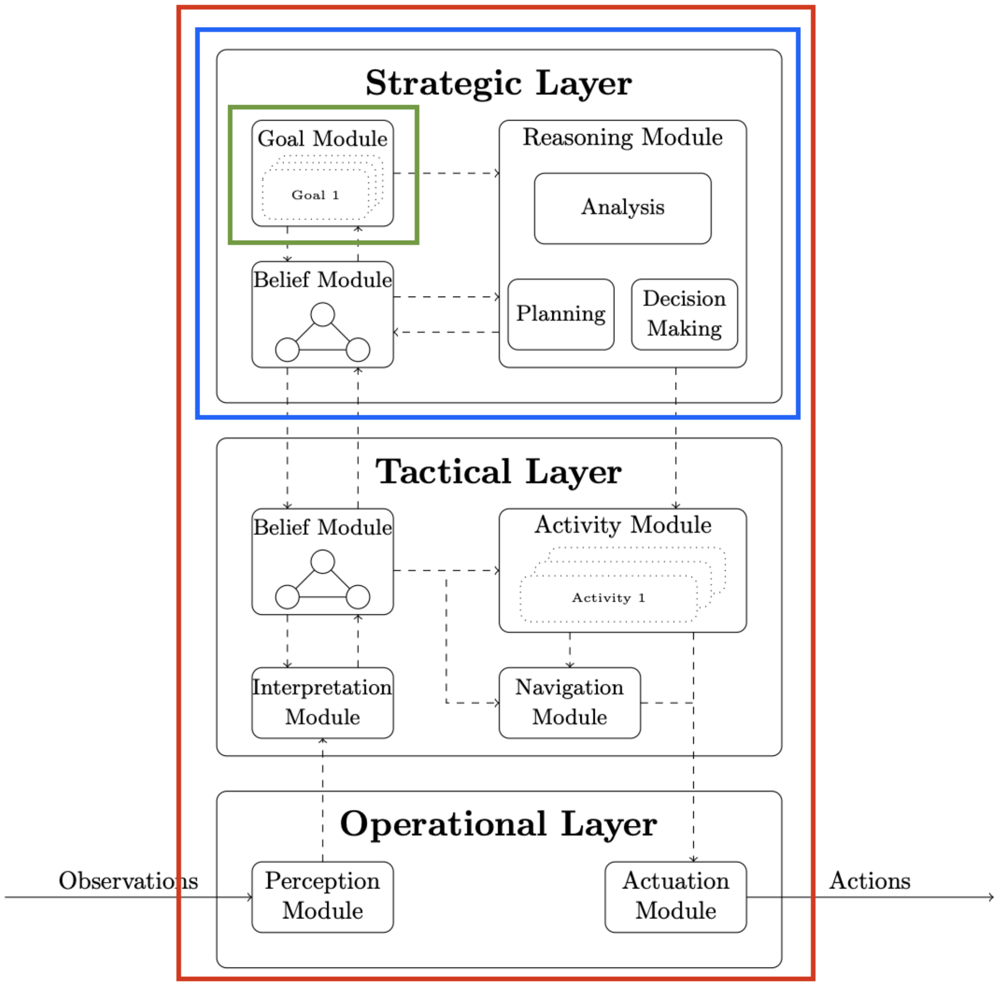
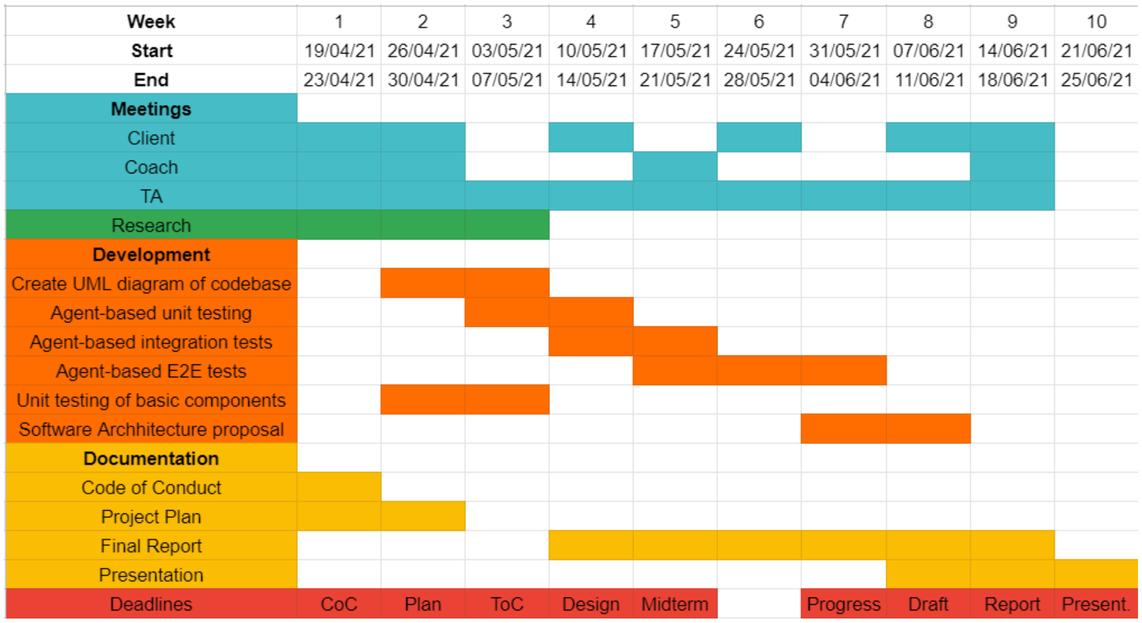


Figure 7: Architecture of AATOM's agents. Adapted from Janssen et al., 2019, p. 5.

G Project Schedule



Week	1	2	3	4	5	6	7	8	9	10
Start	19/04/21	26/04/21	03/05/21	10/05/21	17/05/21	24/05/21	31/05/21	07/06/21	14/06/21	21/06/21
End	23/04/21	30/04/21	07/05/21	14/05/21	21/05/21	28/05/21	04/06/21	11/06/21	18/06/21	25/06/21
Meetings										
Client	🟦	🟦		🟦		🟦		🟦	🟦	
Coach	🟦	🟦			🟦				🟦	
TA	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	
Research	🟩	🟩	🟩							
Development										
Create UML diagram of codebase		🟧	🟧							
Agent-based unit testing			🟧	🟧						
Agent-based integration tests				🟧	🟧					
Agent-based E2E tests					🟧	🟧	🟧			
Unit testing of basic components		🟧	🟧							
Software Archhitecture proposal							🟧	🟧		
Documentation										
Code of Conduct	🟨									
Project Plan	🟨	🟨								
Final Report				🟨	🟨	🟨	🟨	🟨	🟨	
Presentation								🟨	🟨	🟨
Deadlines	CoC	Plan	ToC	Design	Midterm		Progress	Draft	Report	Present.

Figure 8: Schedule of the project plan.

H Group division of labour matrix

To be added.