Emergent Architecture Design

Blocks World for Teams

BW4T Context Project

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

EMERGENT ARCHITECTURE DESIGN

BLOCKS WORLD FOR TEAMS

by

BW4T Context Project

Architecture Design Team:	Daniel Swaab	4237455
	Valentine Mairet	4141784
	Xander Zonneveld	1509608
	Ruben Starmans	4141792
	Calvin Wong Loi Sing	4076699
	Martin Jorn Rogalla	4173635
	Jan Giesenberg	4174720
	Wendy Bolier	4133633
	Joost Rotheweiler	4246551
	Joop Aué	4139534
	Katia Asmoredjo	4091760
	Sander Liebens	4207750
	Sille Kamoen	1534866
	Shirley de Wit	4249259
	Tom Peeters	4176510
	Tim van Rossum	4246306
	Arun Malhoe	4148703
	Seu Man To	4064976
	Nick Feddes	4229770

TU Delft Delft University of Technology

CONTENTS

ABSTRACT

Given the BW4T environment, we have been assigned the task to enhance this software and to extend its features according to what our customer needs. This new product will be used for further research in the domain of Joint Activity and human robot interaction. Not only will our product stand out in comparison to other products on the market, but it will also expand the horizons of students and researchers, allowing them to reason about processes humans naturally understand and follow.

1

INTRODUCTION

In this chapter, we will explain our goals for this project. We split them up into several tasks that work towards achieving our design goals, which include but are not limited to: reliability, modifiability and ease of use.

1.1. DESIGN GOALS

For this project we were given a large codebase. This codebase had grown complex and convoluted. The task for us was to refactor the code to reduce complexity, make it more manageable and give it a plugin architecture.
On further inspection, however, it was revealed that an exact plugin architecture wasn't precisely what the product owner wanted. By analyzing the product owner's final goals, we decided that a modular design which can be easily expanded or modified was actually desired.
Our goal is to have most, if not all core parts of the environment and agents built using interfaces. This allows for easy addition and modification of elements in the environment or agent behaviour. Creating these additional features would then be done by creating new classes utilizing one of these set up interfaces. Similarly these classes could then later be easily removed without causing parts of the system to start malfunctioning or breaking entirely. Section 2.4 covers a more detailed explanations of several design choices we made during the project.

For the refactoring of the code a number of tasks need to be completed:

- **Split up codebase into Client and Server** - Currently the client and server share the same codebase. This makes it hard to keep a proper overview as it isn't always clear at a glance which class belongs to which subsystem, and some classes are even shared entirely. We want to split these subsystems into individual projects to make it easier to maintain, extend and test them.
- **Convert to Maven project** - Maven is a build automation tool. It makes building and testing the project easier because it shields you from many of the details. It replaces the current complicated ANT buildscript, and accounts for dependencies. Maven provides guidelines for best practices, which is useful for large projects which can become very complex easily, like BW4T. Using Maven, it becomes much simpler to maintain a continous working project.
- **Improve and add to the GUI** - There is much room for improvement for the GUI. BW4T is currently run by executing serveral different batch scripts. We want to make it more user-friendly in addition to adding a whole new GUI for the creation of maps, and the configuration of bots.
- **Improve overall code quality** - Using sourcecode analysis tools like Sonar and inFocus, we discovered that there were a lot of code quality issues present in the existing BW4T codebase.
  - There is a lot of unused code present, and due to the complexity of the codebase it can be hard to tell which pieces of code are unused.
  - Furthermore, there are a lot of very complex methods and classes. We want to reduce complexity to make the code easier to understand, maintain and extend.

  - In order to make the code more readable, we want to apply a well-defined checkstyle throughout the project. This includes the use of curly braces, indentation and sufficient javadoc

- **Improve Javadoc** - Javadoc's are sometimes lacking in information or occasionally not present at all. We want to improve the quality of the javadoc, and add missing documentation for classes that are important. This makes navigating and understanding the code easier.
- **Add sufficient tests** - There are currently no tests present. In order to ensure functionality after changes, tests will need to be written. We want to achieve 50% line coverage for existing code, and 75% line coverage for newly written code.
- **Maintain backwards compatibility** - Old simulations should still work, and give the same results in the new BW4T. This includes that BW4T should still be usable for the Logic-based AI course in the first year of Computer Science.

2

Software Architecture Views

In this chapter we will explain our software architecture views. First we will talk about the subsystem decompostion. After that we will talk about the software and hardware mapping of this subsytems. Then we will talk about persistent data management. The last subject is concurrency.

2.1. Subsystem Decomposition

Our system consists of various parts. Currently, we have:

- The server project, containing all files to run a server instance of BW4T
- The client project, containing all files to run a client (controlling one agent) on a running server.
- The core project, which is the central "library" containing all shared assets and classes used by more then one project. The core project is not runnable, it only contains shared elements to be used by other projects.
- The map editor project, contains all the files needed to run the map editor as a standalone application.
- The scenario editor, contains all files for the scenario editor.

All these systems can run independently from eachother, except for the core project. All subsystems need the core project files to run.

2.2. Hardware/software mapping

BW4T will mostly be running on a single PC. Because of the Server-Client architecture, it is possible to run the server and the client(s) on different PC's. There are no special requirements. Both the server and clients can run on the same or different types of computer. The other subsystems like the map editor can run on any PC with Java 6 installed.

2.3. Persistent Data Management

In the Blocks World for Teams project there is no need to use databases as log files and map files are the only persistent data.

2.3.1. Log files

At the start, the log files were raw dumps of the actions of a bot (might be human, might be agent). These actions varied from walking to a spot in the map to interacting with an item.
There was no use for these log files besides debugging, but the log files were too poorly organised and contained insufficient information to be of significant value for this.
Now, the logging functionality has been extended to be a lot more flexible in logging various things. The log files may be used to reconstruct a series of actions or events. You would be able to feed a log file to the client,

which will run the bot according to the lines in the log file.
A better defined logfile can also be used for various other purposes. By running an analysis of the log files, you can gather information about the results, efficiency and possible room for improvement. This can also be used to determine why a bot made certain decisions, or why it did not do what you expected it to do.

2.3.2. MAP FILES

The map files are fairly large XML files but are saved as ".map" extension, which consist of a declaration of the map.
The XML file needs to contain at least the following:

- The size of the map (an x and y value)
- The bot entities (including an (x,y) starting position)
- The zones (including the name of the zone, the colour of the boxes inside, its neighbouring zones and an (x,y) value)
- The sequence of coloured blocks to be collected (or alternatively set to be random via a seperate boolean)
- Whether multiple bots are allowed in one room (boolean)
- Whether the number of blocks in the rooms are randomized or not (boolean)

Because these XML files can be rather difficult to write by hand, there is currently a tool which creates a map for you according to certain parameters. This tool will however always generate the maps in a grid formation, with all rooms being the same size, which results in the maps always looking rather similar.
The new map editor which we created still uses a grid layout but now you can add rooms, blockades, etc wherever you like by clicking on the grid. This way you can create custom maps which will be converted to a XML file instead of the similar looking grid formation maps. Some extra features also added to the tool:

- An option to create random maps. With this a random map can be generated with one startzone, one dropzone and a random amount of blockades, charging zones and rooms. This random map can be edited if the user wishes to do so.
- An option to create random blocks on the map. With this the user can choose the colors he wants and random blocks with those colors will be added to all the rooms instead of adding them manually.
- An option to create a random sequence. With this the user can select the colors and length wanted and a random sequence of colors is created.
- An option to preview the map. The user can now preview the map before saving. Furthermore the user can preview the map while editing and the preview gets updated with every change to the map.
- An option to open previously created maps. The user can now open previously created maps and edit them. This was not possible with the old map editor

2.4. CONCURRENCY

As stated in section 2.2, the BW4T-Server needs to accept connections from various BW4T-Clients. The following decisions have been made regarding client priorities:

- Every connected client is allowed to start the debugging process and/or control it.
- The agents' actions, percepts, etc. which are time dependent are offered by the server in a first-come first-serve basis.
- The agents should be indistinguishable and should not suffer from any effects due to being from a different client.

Concurrency will mostly be handled by a tick system. Every time a tick is executed, an agent performs a certain action which is allowed by the server within that specific time-frame. If a non-shareable atomic action is requested by an agent, the server will look up in the configuration what it is supposed to do. The default behaviour in this situation is that the server will select each agent in a random order and allow them to perform the requested action.

3

Design Choices

This chapter will cover the various design choices we had to make during the project. We explain why we implemented features the way we did, or why we chose not to persue some of the initial goals.

3.1. Repast Symphony

An initial requirement set by the client was the full implementation of Repast. Repast is a agent simulation toolkit. BW4T uses very small parts of its functionality, though heavily extended and modified. Repast offers various advanced simulation functionality, which could be interesting for a program like BW4T. We investigated the possibilities of implementing a lot more of this functionality. Unfortunately, BW4T was not built from the ground up with Repast in mind. This resulted in some basic framework use, with custom classes replacing most of Repast's real functionality. Because of the nature of both BW4T and Repast, it is very difficult to implement more Repast functionality in BW4T without breaking some of the current functionality. It should be possible, but not without a very very thorough restructure, or a complete rebuild from the ground up.

This was not feasible for the Contextproject due to the lack of time and experience. Therefore we decided to leave the current Repast integration untouched. We did however upgraded to the latest version of Repast, and fixed the massive dependency on the entire Repast project. Instead, the libraries were extracted and added as a project dependency, resulting in a drastic decrease of download size.

3.2. Maven and Code split

In the BW4T v2 code base, all the code was in a single folder/package structure. Code for the server, client, environment and all the other tools all resided in a single project. Because of this, getting a clear overview of the project structure was very hard. To resolve this, all separate sections were split into separate projects. This resulted in a separate project for the Server and the Client. Some of the code is shared between the projects. To avoid mutual dependencies, a 3rd project was created. The Core project contains code that is shared between all projects within BW4T.

All these projects can be seperately built, tested and released as a jar file. This also greatly improves the general structure and maintainability of BW4T. In order to simplify the whole build-test-release process, we also converted these seperate projects to Maven. A "parent" project was created to build all projects at once. This project does not contain code for BW4T, but manages the checkstyle, project hierarchy and dependencies.

Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2

Central Repository, and stores them in a local cache. This local cache of downloaded artifacts can also be updated with artifacts created by local projects.

For each of the additional features like the bot store and scenario GUI, a seperate project was also created and added to the parent project. This results in a lot of seperate project directories in the IDE, but the greatly improved overview and manageability compensates for this.

3.3. GitHub

The original BW4T project used SVN, and was hosted on a local SVN repository. For BW4Tv3, we switched to a different kind of version control; Git, hosted on GitHub. Git is more suitable for large groups working on the same projects by working with branches. Branches are copies of the codebase, in which you can make changes parallel to the main branch. After you implemented a feature or applied the intended changes, you can merge the code with the main branch. GitHub also offers various improvements over a self-hosted repository. Because of the public nature of the site, documentation is improved because it shows a good and clear history of the project. It documents the various issues, commits, pull requests and has a built in wiki for manual documenation purposes.

3.4. Logger

BW4T used a custom logger to track some agent actions. This was not very flexible, and had to be manually extended when more functionality was required. We chose to replace this with an existing logging solution, namely Log4J. This standard logger package is included in the Java libraries, so it does not cause an extra dependency. It allows for a lot more flexibility in log levels (i.e. Debugging, Warning, Error) and is very easy to call from anywhere in the project. This way, you can add logging to almost any part of the project when desired. The logger writes different levels to to the console, but only the highest level to a file. Ofcourse, this is configurable if desired.

3.5. Message Translator

Most of the refactoring done were relatively simple method extracts and if/else optimisations. One class however, was completely overhauled because of its excessive complexity. MessageTranslator.java is responsible for translating Strings to BW4TMessages, BW4TMessages to String, and BW4TMessage to Parameters, all used to interact with GOAL. This was done using 3 methods, all with a huge if/else tree, manually returning the right translation. In order to reduce cyclomatic complexity, the class was refactored to the Command-pattern. This comes down to using Hashmaps to look up the right action, instead of going through the entire if/else tree each time the method is called. Especially for the BW4TMessage-to-String and BW4TMessage-to-Parameter, this results in a big performance increase, next to a much clearer method.

For the String-to-BW4TMessage method, a Hashmap<String, Command> was created. All strings previously checked in the if/else tree, now have their own entry in the map. The Command interface defines a standard getMessage() method, returning the right BW4TMessage upon executing.

For the other 2 methods, a similar Hashmap<MessageType, Command> was created. Each MessageType has its own Command, defined in MessageCommand.java. The interface defines two functions, getString() and getParameter(), returning the right translation upon execution.

This new approach results in a big MessageCommand file, because each MessageType has its own command, but this makes finding bugs, adding MessageTypes, and changing behaviour much simpler.

3.6. Analysis Tools

To analyse code quality, and to make it easier to determine where to start, we used several analysis tools. We mostly used SonarQube, a web-based tool that analyses the code after each change to the master branch on GitHub. It analyses a large number of software metrics and tracks a lot of issues, subdivided in severance (Blocking, Critical, Major, etc.) We used Sonar to determine classes and methods with excessive complexity, checkstyle errors that can lead to bugs and other violated best practices. Next to that, it tracks test coverage, unneeded dependencies, duplicate code.

inFusion is a similar program that you have to run manually. It provides similar metrics as Sonar, but focuses less on issues. Instead, it points out problems with classes as a whole, next to problems with entire methods.