PAD010 Master Thesis
Software Engineering
Thesis No: MSE-2002:29
2002 10

PERFORMANCE-ORIENTED VS. MAINTAINABILITY-ORIENTED IMPLEMENTATION: A CASE STUDY OF THE REACTIVE PLANNER OF TEAM SWEDEN.

By
David Johansson & Daniel Lehtovirta

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 2x20 weeks of full time studies.

Contact Information:
Authors:
Daniel Lehtovirta
David Johansson

University advisor:
Stefan Johansson
Agent Systems Research and Technology

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet: www.bth.se/ipd
Phone: +46 457 38 50 00
Fax: + 46 457 271 25

Contents

Preface

Our supervisor Stefan Johansson was the first to introduce us to the domain of RoboCup, which is an international competition for Artificial Intelligence and Robotics. This because he had himself been involved since the beginning of the competition through the currently only Swedish contribution to the contest, called Team Sweden. Team Sweden competes in a league called the Sony Four Legged League. This particular league uses four legged autonomous robots supplied by Sony, currently the AIBO [13] series.

Our work was in the beginning supposed to be evaluating the impact of different enhancements that could be made on a game-planning module in the system. This work was somewhat complicated by the complex and poorly documented implementation of the module. Because of this we decided to restructure the entire module to make alterations and improvements easier.

However as the work proceeded we realized that it would be more interesting to evaluate the difference between a maintainability-oriented implementation and a performance -oriented implementation. We felt that this would be a more appealing topic in the field of software engineering.

Abstract

Our work is a case study for Team Sweden, which is a national effort to produce a team of soccer playing robots.

We took the present structure of the Reactive Planner, which is the game-planning module of the system, and made two new parallel versions of the Reactive Planner. One where we tried to optimize for CPU and memory usage called the performance-oriented implementation. We also made one implementation where we tried to optimize for maintainability called the maintainability-oriented implementation.

To evaluate the implementations we ran a series of CPU and memory usage tests to assess the performance. We also estimated the maintainability of both implementations. The test results where later used to decide which implementation we should recommend to Team Sweden.

The results showed that the difference in maintainability did outweigh the difference in performance. The conclusion is that the maintainability-oriented implementation is in this case the preferred solution.

Keyword: Performance, Maintainability Effort, Team Sweden, and RoboCup.

1. Introduction

This section contains a short introduction to why we chose the topic and a short description of RoboCup.

1.1. Background

RoboCup [16] is an international football league for autonomous robots originally started in 1993 by a group of Japanese researchers. Their goal was to create an arena to test and promote research in the field of Artificial Intelligence (AI) and Robotics in Japan. As the project took form, AI and Robotics researchers from other nations gained interest in the competition and convinced the arrangers to make the contest an international one. The competition has since then grown and is today divided into a number of different branches of AI and Robotics research.

Team Sweden [17] is a Swedish national effort to produce a team of soccer playing physical robots to enter the RoboCup international competition. Currently Team Sweden is a member of the Sony four-legged league (SFLL). All teams in SFLL have the same hardware, the Sony AIBO robots [13], the focus of this competition is therefore not to build robots but to program them.

We will try to evaluate two different implementations of the Reactive Planner module (RP) with the same functionality. This to deduce how a maintainable-oriented implementation (MOI) will perform compared to a performance-oriented implementation (POI) regarding CPU usage, memory usage and maintainability. The difference between a POI and a MOI is only relative to each other, there is no absolute scale of how maintainable a system is.

In this real-time application the performance of the different modules are important (see Section 2 for a description of the modules). This because all the decisions are made in real-time during the game, and a game planner with high performance will respond quicker to changes in the game. If the response is quicker than that of the opponents this will result in an advantage during a match. Although there is not a critical shortage of memory it is still important to take the memory usage into account so that it does not grow out of proportions. Since the SFLL domain is updated each year with new rules and regulations, the software is under constant revision, which also makes maintainability a very important factor.

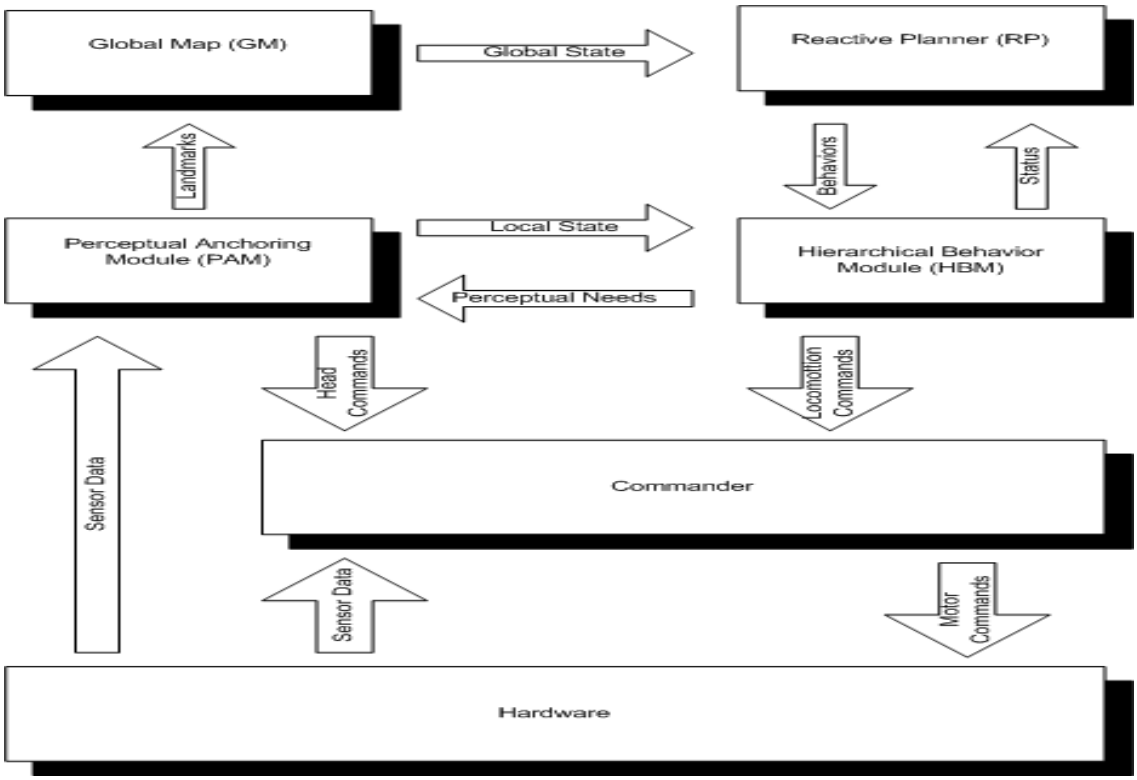

Figure 1. The Team Sweden architecture, which is based on the Thinking Cap architecture.

1.2. Hypothesis

Our hypothesis is that the use of a MOI will decrease performance and increase memory usage, but only to an extent that is negligible in comparison to the gain in maintainability. In the case of the POI, it will increase the performance but decrease the maintainability to a larger extent than the gain of performance.

1.3. Scientific Method

To test if our hypothesis is correct, we carried through a number of tests on performance and maintainability qualities of the two implementations. The tests where followed by an evaluation of the test results to determine which implementation we should recommend to Team Sweden.

1.4. Outline of the Thesis

We will present an architectural overview of the system followed by closer look of the functionality of the Reactive Planner (RP) and the two new versions of the RP. After the overview follows a description of how we conducted the experiments and their results. Lastly we round off with an analysis of the test result, a discussion and a conclusion.

2. Team Sweden Architecture Overview

This section contains an overview of the current architecture used by Team Sweden in RoboCup [10].

- **Commander**

  The Commander is the interface between the hardware and the rest of the system. It sends motor commands to and receives sensor data from the hardware. It also receives locomotion commands from the Hierarchical Behavior Module (HBM) and head commands from the Perceptual Anchoring Module (PAM).

- **Perceptual Anchoring Module**

  The PAM creates a map with positions of object relative to the robots own position from information supplied by the hardware. It also controls the head movements of the robot and various image processing.

- **Hierarchical Behavior Module**

  The HBM gets behavior commands from the RP, and uses position information from the PAM to send locomotion commands to the Commander.
- **Global Map**

  This module creates a map over the playing field from information supplied by the PAM and from information received from the other robots in the team.
- **Reactive Planner**

  The RP processes the map information that is supplied by the Global Map (GM) and the PAM. It calculates the most appropriate behavior for any given map state and sends this information to the HBM. The RP is based on the Electric Field Approach (EFA) [5, 6].

3. The Structure of the RP

This section contains a more detailed overview of the functionality of the RP and an overview of the two different implementations.

The main function of the RP is to emulate a number of behaviors; these are chosen based on the current situation on the map. The RP evaluates what impact these behaviors will have using of the EFA;

"*The Electric Field Approach is a logic replication of a real world phenomenon. The environment is abstracted to a representation of a virtual electric field with areas of positive and negative potential, used as a heuristic function; Dangerous or bad places are represented by negative virtual charges while positive ones represent good places*." [6]

Most fields are calculated in real-time to describe objects such as players, but it is also possible to add static charges to favor or disfavor a certain part of the field such as the goals. The RP calculates the sum of the potential of these fields by probing strategic positions, for instance the position of the ball in the game. As the ball moves over the game field it will be affected by the different electric fields depending on the relative distance and size of the charge on the fields. By calculating the charges it is possible to form an opinion of what behavior that is the best for any given map. The strategic positions can be used to evaluate behaviors that move the ball, the robot etc. This evaluation is done continuously during the game to get a dynamic choice of behaviors.

For our experiment we first constructed a RP module with a certain amount of functionality, this original module had basic performance optimizations such as the use of new and malloc was restricted to the initialization of the module. This version was then developed into two separate versions, which in turn were restructured into a MOI and a POI, this to avoid performance issues related to a difference in functionality. See Section 3.1 and 3.2.

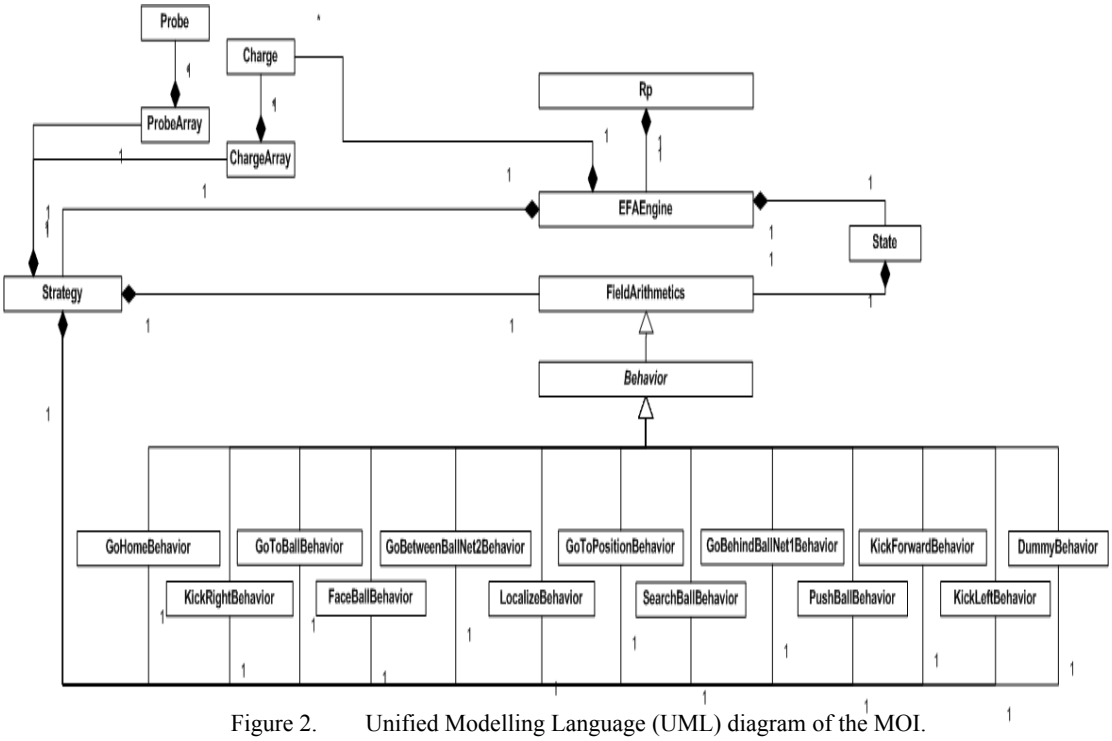

Figure 2. Unified Modelling Language (UML) diagram of the MOI.

3.1. The Maintainability-oriented Implementation

To improve the maintainability of the implementation, we did a logical division of it, shown in Figure 2. This to make replaceable objects within the implementation that could be subjects of replacement or improvement due to errors or environmental changes:

- The Rp object serves as an interface for the module.
- The EFAEngine functions as a controller object in the implementation. It uses the other modules to emulate behaviors and to calculate the resulting potential.
- The State object is used to decide which state the game is in, which in turn is used by EFAEngine to decide what strategy to choose.
- The Strategy object is used to decide what strategy to choose i.e. what behaviors are feasible to evaluate. It also decides how the probes and charges should be used.
- The FieldArithmetics object is used as a utility object to calculate distances and angles on the map.
- The different behavior objects inherit general behavior functions from Behavior and are used to emulate different behaviors on the map. Each behavior object implements a certain behavior that the robot can perform. The behaviors control how the robot should act in a certain situation.
- Probes are used to measure the potential of a behavior. The probes can be placed in positions that correspond with real objects such as the ball. The behavior that raises the potential at the probe the most is considered to be the best behavior.
- The Charge object represents a charge on a point on the map.
- Charges- and Probe-Array objects hold as they imply probes and charges**,** which are used to emulate electric fields and positions to probe.

$$M_{tot} = \sum_{n=1}^{k_s} \left[P(S_n) * \sum_{m=1}^{k_c} V(S_n, C_m) \right]$$

$P(S_n)$ = Probability weight of scenario n
$V(S_n, C_m)$ = Affected volume of component m in scenario n.
k_n = Number of scenarios.
k_c = Total number of components

Figure 3. Describing the Bengtsson-Bosch formula for assessing maintenance effort [3].

Structural Criticism

When making the MOI we took no consideration to performance issues, this means that there probably could be made performance improvements without changing the structure to any large extent. There could also be a more performance friendly structure with the same degree of maintainability [4]. We could for instance remove some of the objects to increase the performance of the implementation.

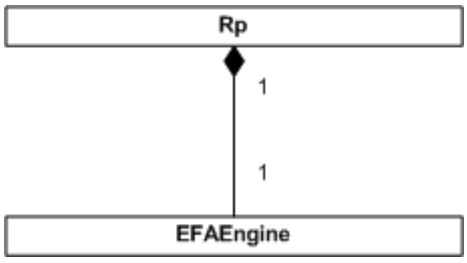

Figure 4. A UML diagram of the POI.

3.2. The Performance-oriented Implementation

The POI is quite straightforward as it only uses two objects, the RP interface, just as in the MOI, and the EFAEngine that now contains all the functionality that the other objects had in the former case. We chose this structure to minimize the calls between objects and the use of dynamically allocated memory.

Structural Criticism

Further improvements in the area of performance seem hard to make in this structure, possibly to remove the interface altogether, but we reasoned that this would result in only a negligible performance gain. However, the C++ language has a number of ways to enhance the performance within the object, for instance with the use of inline functions. It is likely that we have not used these to their full extent, which could affect the performance negative we judge this, as in the case of the MOI, not to alter the final comparison to any large extent. To raise maintainability in the POI we could create a more structured implementation, we could also split the code into several files to make the code easier to understand.

4. Experiment setup

This section contains information about how we conducted our experiments and what we hoped to gain of them.

We wanted to test the relative difference between the implementations because absolute values are different on different platforms. To test the different qualities, we have tried to find techniques that tested the most of the system, in regards to both performance and maintainability.

4.1. Maintainability

To test the maintainability we decided to follow the general guidelines and technique of architecture level software maintenance prediction, presented by Bengtsson and Bosch [7, 8]. The technique involves making changeability scenarios and probability estimations to show how often the change scenarios will occur in the future. These values are then to be used in the mathematical formula in Figure 3.

To get better estimations of the required effort for the change scenarios we also decided to get outside help with estimating the maintainability. We selected a number of architecture and design experts to evaluate how much difference in effort there where between the implementations for each change scenario. For a description of the scenarios see Appendix I.

4.2. CPU Usage

The performance tests could not at this stage be run on the target platform, which is the AIBO series 210 robots, due to the fact that the robot was in a stage where the development environment was not ready for use. We decided to run the tests with the target compiler GCC [12] under the Linux [11] operating system. This could result in measurement errors that cannot be avoided unless the tests are performed directly on the hardware platform.

The tests cases were designed so that they would test the RP's operation and measure the speed difference in percent between the implementations. To follow-up the first test, we also tested an increase in the number of charges, probes and behaviors to see if this would have any impact on the difference between the implementations. We also altered the input i.e. the maps to evaluate if this would have any impact on the difference between the implementations.

We decided to test the performance according to the following test cases;

- **TC1 Normal Case:**

  In this test case we wanted to test the two implementations of the RP with 6 charges, 9 behaviors and 1 probe and as we made it initially.
  This test case is going to be used as a template test, to evaluate the general performance for the implementations.

- **TC2 Increasing the number of charges:**

  In this test case we wanted to measure the difference in performance between the two implementations if we changed the number of charges that was used.
  We chose to increase the charges from the normal case of 6 to the maximum of the current system configuration, a total of 50 charges.

- **TC3 Increasing the number of probes:**

  Here we wanted to see if there is a difference in performance between the implementations if we altered the number of probes in the systems.
  We increased the number of probes with 4 making a total of 5 probes in the system. The reason we decided to test 5 probes was that this was the current maximum number of allowed probes in the current system configuration. We saw no reason to increase the maximum number of probes.

- **TC4 Increasing the number of behaviors:**
  In this test case we wanted to test the difference in performance between the two implementations if we changed the number of behaviors that were evaluated.
  We multiplied the number of behaviors by a factor 4 making a total of 36 behavior evaluations. We decided to increase the behaviors by a factor 4 because we felt that this would give us a good enough base for our test results. We also estimated, based on our knowledge of the system, that this was the highest number of behaviors that would be added within the next 2 years if no larger change to the behavioral system would be implemented.

- **TC5 Increasing the number of charges, probes and behaviors:**

  With this test we want to see what the difference in performance was if we maximized the implementations. We used 5 probes, 50 charges and 36 behaviors. This test in other words incorporates TC2, 3 & 4.

4.3. Memory Usage

To test the memory we used the same test cases as in the CPU tests. See Section 4.2 for a description of the test cases. We used a tool to measure the memory usage called “top” for Gnu/Linux, which is an integrated program in the Red Hat distribution [11]. The reason we chose “top” was that it is a widely spread software and very easy to use.

5. Analysis

This section contains the results of our tests and our interpretation of them.

5.1. Maintainability

Here is a presentation of the different effort evaluation results.

Effort Evaluation by experts

The experts evaluated the difference in effort between the two implementations for each change scenario. The experts consisted of two domain experts and three general software designers. See Appendix II for the estimations.

One thing to take in account is that the domain experts are familiar with both implementations and their functionality, because of this they might not be entirely objective in their estimations, as it is hard to estimate the time already spent on understanding the implementations.
Another thing to take into consideration is that 47 percent of all maintainability effort is estimated to be spent on understanding the code of system [9]. The experts have not taken these 47 percent into consideration in these estimations.

Result
The overall estimated effort difference was 32 percent in favor of the MOI. That means that it takes 32 percent more time to make a change in the POI than in the MOI. We got this result by calculating the arithmetic average for each scenario and then calculating the total average for the MOI. See Appendix II for the data and calculations.

Effort Evaluation with the Bengtsson-Bosch Formula

The Bengtsson-Bosch formula is another way to calculate the effort that is required for maintenance of an implementation. A formula takes an arithmetical approach to estimate the effort required for maintenance based on software metrics. To make use of the formula we needed to estimate the probabilities, or weights, for each of the scenarios to occur. Two domain experts made these estimations. The experts had also reviewed historical data that was available for the domain. We also needed the affected code volume for each scenario, which we retrieved through estimations of each component. See Appendix III Table 1, 2 and 3 for the data and calculations.
Result
To get the effort of the different implementations, we multiplied the probability estimation of each scenario with its affected code volume and added up these to get the total effort for each implementation, as described in the formula in Figure 3. We then calculated the difference between the MOI and POI to 14 percent, i.e. the POI takes 14 percent more effort to maintain during a two-year period than the MOI. See Appendix III Table 1, 2, 3 and 4 for the data and calculations.

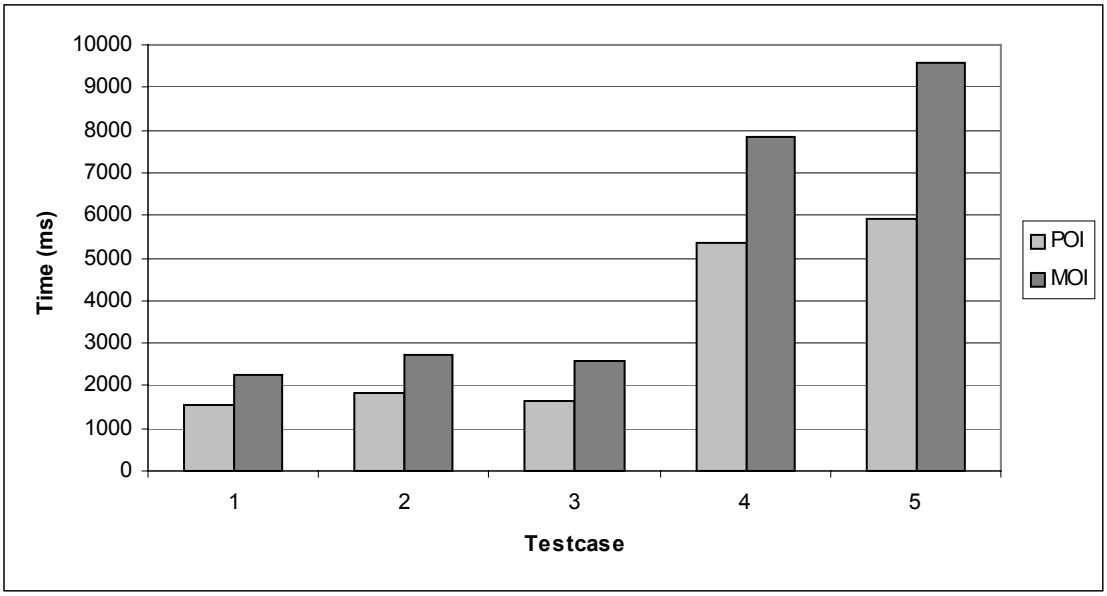

Figure 5. Describing the performance difference between the implementations.

Effort Comparison

After reviewing the results of the former estimations, we came to the conclusion that we could not compare the results. This because the Bengtsson-Bosch formula took into account the probability estimation of the scenarios, whereas the estimations of the experts did not. The expert estimated the difference between the implementations and not the size of the changes. We decided to solve this by removing the scenario probability estimation from the Bengtsson-Bosch formula in Figure 3, thus only expressing the effort as the affected code volume of each component.

Result

Using the modified formula we got an estimated effort difference of 15 percent, which is comparable with the 32 percent we got from the experts. To make use of both estimation methods, we calculated the arithmetic average of these to 24 percent.

5.2. CPU Usage

The test results can be seen in Appendix IV Table 1 and Figure 5 illustrates the difference between the two implementations in milliseconds.

- **TC1 Normal Case:**

  We got an average speed difference of 46 percent in favor to the POI.

- **TC2 Increasing the number of charges:**

  The result showed an arithmetic average difference of 49 percent in favor to the POI.

- **TC3 Increasing the number of probes:**

  The test case generated an arithmetic average difference of 58 percent in favor to the POI.

- **TC4 Increasing the number of behaviors**:

  Here we got an arithmetic average difference of 46 percent in favor to the POI.

- **TC5 Increasing the number of charges, probes and behaviors**

  In this test case we got an arithmetic average difference of 61 percent in favor to the POI.

The difference in overall performance between the two implementations is mostly due to the communication between objects in the MOI. Within our scope the findings show that the performance difference between the implementations is linear. This does however not mean that additional increases will follow the same trajectory.

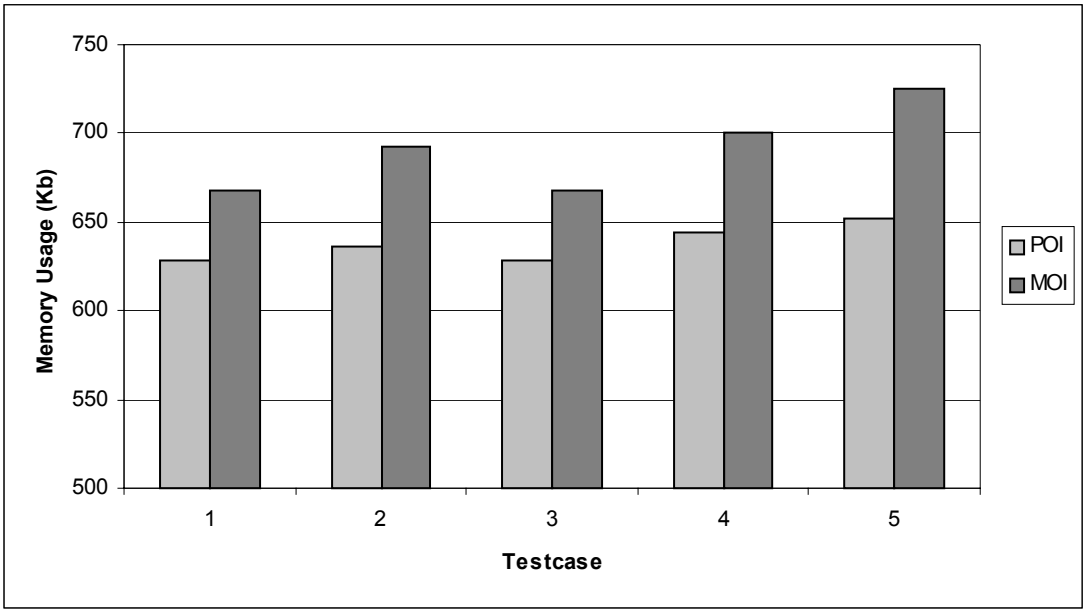

Figure 6. Describing the memory usage difference between the implementations

5.3. Memory Usage

Figure 6 illustrates the difference in memory usage between the two implementations in Kilobyte (Kb). The Memory usage test generated the following results see Appendix IV Table 3:

- **TC1 Normal Case:**

  POI used 628Kb and the MOI used 668Kb, making a difference of 40Kb or 6 percent in favor to the POI.

- **TC2 Increasing the number of charges:**

  POI used 636Kb and the MOI used 693Kb, making a difference of 57Kb or 9 percent in favor to the POI.

- **TC3 Increasing the number of probes:**

  POI used 628Kb and the MOI used 668Kb, making a difference of 40Kb or 6 percent in favor to the POI.

- **TC4 Increasing the number of behaviors:**

  POI used 644Kb and the MOI used 700Kb, making a difference of 56Kb or 9 percent in favor to the POI.

- **TC5 Increasing the number of charges, probes and behaviors**

  POI used 652Kb and the MOI used 725Kb, making a difference of 73Kb or 11 percent in favor to the POI.

The increase of memory usage in the MOI versus the POI is due to the number of objects. This is also supported by the fact that the increase in memory usage is an accumulation between the test cases, meaning that the total memory usage increase in test case 1,2,3,4 is the same as in TC5.

6. Discussion

This section contains our interpretation of the test result and a general discussion about what improvements that could be made to the RP.

The values we have estimated leave room for interpretation. We believe that to get more certain estimation more effort estimations methods could be used. This is an area where more research is needed.

The estimation method could also be viewed with some skepticism since the experts that estimated the effort required between the implementations could be more familiar with the implementations. However we believe that experts should not be too familiar with the implementations either, because then it could then be hard to perceive the effort difference based on the complexity of the implementations. The Bengtsson-Bosch formula on the other hand estimates the size of the affected volume in lines of code (LOC) to be proportional to the effort required to maintain the components, this might not be the best way to estimate the effort required to maintain a system. We feel that maintenance is somewhat more than just the number of lines of code, for instance, documentation of the system will also decrease the maintenance effort.

"… there is a clear and intuitive connection between poorly structured and poorly documented products and their maintainability." [1].

One aspect to take into consideration is the fact that this real-time system, unlike many others, is altered and improved frequently, at the very least each year. This makes maintainability an important factor. In addition, there are also a number of new people introduced to the system each year, namely students, which makes it even more important to have a more maintainable implementation of the system. A possible symptom of not having a maintainable system is that the programmers lose control of the system and it grows out of proportions, which is bad for both performance and maintainability.

6.1. Future improvements

We suspect that improvements could be made to the MOI to improve its performance without affecting its maintainability to any large extent. One concrete and easy improvement could be changing the charge and probe classes to structs thereby avoiding a large number of objects and object calls in the system.

The scenario that affected the most modules was S1, which was to change the map structure. To reduce the affected modules and thereby increase the maintainability of the RP, a container class that converts the incoming map formats to an internal format could be created.

One good way to increase maintainability without loosing performance is to have a good documentation of the system. For instance design documentation, sequence diagrams etc. There are some programs that could help Team Sweden with documentation for instance "Doxygen" [14] which is a program that auto generates documentation from source files. Other ways to improve the maintenances process is to use a Concurrent Versions System (CVS) [15], which manages the different versions of the source code.

7. Conclusion

According to information received from Team Sweden, the RP version from the 2001 tournament used 1.5 percent of the AIBO robots total CPU capacity and all of the modules used about 60 percent of the CPU capacity. If the POI uses the same amount of CPU capacity as the previous version, an increase of 40-60 percent of the RP module will only result in a CPU usage of 2.4 percent (1.5 * 1.60 = 2.4 percent) of the total CPU capacity. This is to be compared towards a 24 percent more maintainability effort spent on the POI then the MOI.

Based on historical data of the modules CPU usage, we draw the conclusion that the increase in CPU usage of the MOI version of the RP is negligible in comparison to the gain in maintainability. Therefore in the case of team Sweden, we would recommend the MOI, given the fact that there are new developers introduced to the system each year.

It is hard to draw and any general conclusion since this is a case study. The choice of choosing a maintainability oriented solution or a performance oriented, depends on factors such as the importance of performance and how much effort that is spent on maintaining the system.

8. References

[1] Software Engineering Shari, Lawrence, Pfleeger (1998) ISBN 0-13-081272-2

[2] Design and use of software architectures, J. Bosch (2000) ISBN 0-201-67494-7

[3] Design and Evaluation of Software Architecture P. O. Bengtsson (1999) ISSN 1103-1581

[4] Maintainability Myth Causes Performance Problems in Parallel Applications, D. Häggander, P.O. Bengtsson, J. Bosch, L. Lundberg, (1999) Proceedings 3rd Annual IASTED SEA pp.288-294

[5] Using the Electric Field Approach in the RoboCup domain S. Johansson and A. Saffiotti (2001) Proc. of the RoboCup Symposium (Seattle, WA, 2001) in press

[6] An Electric Field Approach – A Strategy for Sony Four-Legged Robot, J. Johansson (2001) M.Sc. Thesis MSE-2001-03, Blekinge Institute of Technology

[7] Architecture Level Prediction of Software Maintenance, P.O. Bengtsson, J. Bosch (1999) in Proceedings of Third European Conference on Software Maintenance and Reengineering, Amsterdam, Netherlands, March, pp. 139-147

[8] Architecture-Level Modifiability Analysis, P.O. Bengtsson (2002) Doctoral Thesis ISSN 1103-1581, ISRN HK-R-RES-99/10—SE

[9] Tutorial on Software Maintenance (1983), G. Parikh, N. Zvegintzov, IEEE Computer Society Press, pp 61-62

[10] Team Sweden (2002) A. Saffiotti, A Björklund, S. Johansson, Z. Wasik, Birk, S. Coradeschi, and S. Tadokoro (eds) *RoboCup 2001*, Springer-Verlag, 2002, in press

[11] Red Hat – Linux, Embedded Linux and Open Source Solutions http://www.redhat.com (2002-10-04)

[12] GCC Home Page – GNU Project – Free Software Foundation (FSF) http://gcc.gnu.org (2002-10-04)

[13] Entertainment Robot AIBO, http://www.AIBO.com (2002-10-04)

[14] Doxygen, http://www.stack.nl/~dimitri/doxygen (2002-10-06)

[15] Concurrent Versions System, http://www.cvshome.org (2002-10-06)

[16] RoboCup Official Site, http://www.robocup.org (2002-10-06)

[17] Team Sweden at RoboCup, http://aass.oru.se/Agora/RoboCup (2002-10-06)

[18] RoboCup Legged Robot League, https://www.openr.org/page1_2003/ (2002-10-06)

Appendix I: Scenario Description

1. A description of the scenarios.

Name	Scenario Category	Scenario Description
S1	Algorithm Changes	New sensor makes the map more accurate and the old map structure is exchanged. We have to change the calculations of the map. Results in: System wide changes
S2	Hardware Changes	A new locomotion device is fitted to the robot, which results in changes in the speed, distance calculations i.e. the kick-distance etc. Results in: Changes in the behaviours.
S3	Environmental Changes	The RoboCup rules changes to allow a different number of players per team. Results in: State and Behaviours
S4	Environmental Changes	The RoboCup rules changes so that the game field size is changed. Results in: Changes in behaviours.
S5	Environmental Changes/ Hardware Changes	The system is to be changed to allow communication for synchronization of strategies between the robots. Results in: Changes in Strategy
S6	Structural Changes	There is a need to add a new behaviour because the present behaviours are not enough to solve a new problem during a match. Results in: An additional new behaviour class.
S7	Structural Changes	There is a need to add a new strategy because the present strategies are not enough to solve a new problem during a match. Results in: Changes in Strategy.
S8	Structural Changes	We want to add charges to get better estimations of the expected result of the behaviors. Results in: Changes in Strategy
S9	Structural Changes	We want to add probes to get better estimations of the expected result of the behaviors. Results in: Changes in Strategy
S10	Algorithm Changes	We want to change the potential calculations to make them more suited for new problems in the game. Results in: Changes in EFAEngine
S11	Algorithm Changes	The algorithms for altering the objects position in the map need to be changed due to optimisations issues. Results in: Changes in the abstract Behaviour class.
S12	Structural Changes	Remove the Lps map so that we only make use of one map structure. Results in: EFAEngine, Behaviour classes and Rp
S13	Algorithm changes	Extend the RP to include estimations of how much time it takes to perform a certain behavior. Results in: Changes in Behavior classes and EFAEngine.

Appendix II: Expert Effort Estimations

1. The evaluation of the Experts

E1 – E2: Domain Experts
E3 – E5: Design Experts
S1 – S13: Scenarios

	E1	E2	E3	E4	E5	Average
S1	5%	10%	0%	10%	0%	5%
S2	20%	15%	15%	50%	25%	25%
S3	30%	15%	25%	10%	100%	36%
S4	10%	10%	35%	100%	0%	31%
S5	10%	10%	30%	50%	50%	30%
S6	10%	20%	100%	100%	100%	66%
S7	5%	5%	25%	20%	100%	31%
S8	0%	0%	20%	10%	0%	6%
S9	0%	0%	20%	10%	0%	6%
S10	10%	15%	10%	10%	50%	19%
S11	10%	10%	10%	40%	25%	19%
S12	30%	60%	50%	100%	50%	58%
S13	25%	20%	100%	100%	25%	54%
					Total Average	32%

Appendix III: Bengtsson-Bosch Estimations

1. Lines of code in the maintainability-orientated implementation

Scenario	LOC
S1	800
S2	70
S3	50
S4	50
S5	20
S6	50
S7	20
S8	10
S9	10
S10	20
S11	50
S12	400
S13	50

2. Lines of code in the performance-orientated implementation

Scenario	LOC
S1	800
S2	70
S3	50
S4	50
S5	20
S6	70
S7	20
S8	10
S9	10
S10	20
S11	50
S12	600
S13	70

3. Scenario Weight

E1 and E2: Domain Experts
S1 – S13: Scenarios
P (S): Scenario probability, which is calculated by dividing the average for each scenario with the total average for all scenarios.

Scenario	E1	E2	Average	P (S)
S1	20	30	25	0,04
S2	10	90	50	0,07
S3	50	20	35	0,05
S4	50	20	35	0,05
S5	80	90	85	0,12
S6	90	97	94	0,13
S7	85	92	89	0,13
S8	70	80	75	0,11
S9	60	62	61	0,09
S10	30	41	36	0,05
S11	20	64	42	0,06
S12	20	19	20	0,03
S13	50	58	54	0,08
Sum			701	1

4. Bengtsson-Bosch Calculations

Volume: The volume of the affected code in the maintainability- or POI in Scenario n.
Calculated Effort: The Effort required maintaining the structure during a maintenance cycle.

Scenario	P(S)	MOI Volume	POI Volume	Calculated MOI LOC	Calculated POI LOC
S1	0,04	800	800	28,5	28,5
S2	0,07	70	70	5,0	5,0
S3	0,05	50	50	2,5	2,5
S4	0,05	50	50	2,5	2,5
S5	0,12	20	20	2,4	2,4
S6	0,13	50	70	6,7	9,4
S7	0,13	20	20	2,5	2,5
S8	0,11	10	10	1,1	1,1
S9	0,09	10	10	0,9	0,9
S10	0,05	20	20	1,0	1,0
S11	0,06	50	50	3,0	3,0
S12	0,03	400	600	11,4	17,1
S13	0,08	50	70	3,9	5,4
Sum	1			5,5	6,3
			Difference		14%

5. Modified Bengtsson-Bosch Calculation

	MOI Volume	POI Volume
S1	800	800
S2	70	70
S3	50	50
S4	50	50
S5	20	20
S6	50	70
S7	20	20
S8	10	10
S9	10	10
S10	20	20
S11	50	50
S12	400	600
S13	50	70
Average	123,1	141,5
Difference		15%

Appendix IV: Performance Test Data

1. CPU-Usage test in milliseconds.
TC1 – TC5: Test cases.

MOI	TC1	TC2	TC3	TC4	TC5
	2270	2720	2600	7830	9560
	2250	2720	2590	7820	9570
	2270	2720	2600	7820	9570
	2250	2720	2600	7820	9560
	2260	2720	2590	7820	9560
	2260	2730	2600	7820	9570
	2260	2730	2600	7830	9580
	2260	2720	2600	7820	9570
	2260	2730	2600	7820	9560
	2260	2720	2590	7820	9560
	2260	2720	2590	7820	9560
	2260	2720	2590	7820	9570
Average	2260	2723	2596	7822	9566

POI	TC1	TC2	TC3	TC4	TC5
	1550	1820	1650	5350	5930
	1550	1830	1640	5350	5930
	1550	1820	1650	5340	5930
	1550	1830	1640	5360	5940
	1540	1820	1650	5350	5930
	1550	1830	1640	5350	5930
	1550	1820	1640	5360	5930
	1550	1820	1650	5340	5930
	1550	1820	1640	5340	5930
	1550	1830	1640	5350	5940
	1550	1820	1640	5350	5930
	1550	1830	1640	5350	5930
Average	1549	1824	1643	5349	5932

Percentage Difference	46%	49%	58%	46%	61%

2. Memory Tests

	TC1	TC2	TC3	TC4	TC5
MOI	668	693	668	700	725
POI	628	636	628	644	652
Percentage Difference	6%	9%	6%	9%	11%