Software Engineers doing Hardware

Lessons learned while building a robot from scratch

Emile van Gerwen

Being a software company working in the advanced machine building industry, many of our employees come across exotic hardware and the hardware engineers that build them. The project addressed in this paper is completely different. Instead of blaming the hardware engineers for their faulty work, just as they routinely blame our software, we have to build the whole package ourselves, including the hardware. The challenge is to build a robot for the TNO Robot Competition 2003[1]. Although the main goal is to have fun and all the work is to be done in spare time, it has all elements of a "real" project: a fixed set of requirements (rules), a fixed end date, and a fixed budget. At the moment of writing, all 19 teams have participated in a test mission. Our robot was one of the four robots, and the only robot that took part for the first time this year, that completed the mission successfully. It is tempting to write a "the secret of our success" kind of story, but with the real tournament ahead of us we need to be a bit careful. Nevertheless, we learned some important lessons that we think might be valuable for other software engineers doing hardware.

Rules of the Game

To get some feeling of the scope of the project, we will briefly describe what the robot competition is about. With a budget of 2500 Euro, a team has to build a single autonomous mobile robot that can accomplish 5 different missions. The robot, maximum size 60x60 cm, must complete the mission within 3 minutes to get 10 points awarded. The top 3 robots with fastest time in a single mission get bonus points. The robot that gathers most points over all missions, including the points gathered at a test round held two months before the competition, wins. The missions are:

1. Escaping out of a known maze;
2. Getting out of an unknown maze;
3. Finding and touching a soccer ball on a grass-like field;
4. Moving a soccer ball out of the playing field;
5. Driving to the end of an elevated race track without falling off.

All missions, except the last, take place in a 6 x 4 meter playing field.

Where to begin? (Definition)

Not having any reference to previous hardware projects, we decided to see how we could make use of our software project experience in this particular multi-disciplinary project. Being a CMM Level 2-almost-3 company, procedures and best practices for making software are well-known to us. But how could they be of any use for building a robot from scratch? As a start, the title of the documents we were going to make definitely had to change to reflect our new line of business. The table below shows the revised titles.

Old (software)	New (multi disciplinary)
Customer Requirements	Rules of the Game
Software Requirement Specification	Battle Plan
Architecture and Design	Construction Manual
Project Management Plan	Bill of Materials, Planning (see text)

Table 1: Document title translation

Let's discuss these documents in more detail.

The *customer requirements specification* or rules of the game were issued by the TNO jury. Just as in ordinary project, however careful written down, all specifications are subject to different interpretation. The jury in this case anticipated no different and would answer any questions related to the rules. All questions and answers would be distributed to all other teams in as "Frequently Asked Questions", unless this would reveal a team's secret strategy. Many teams, including us, used that opportunity to clarify the customer requirements.

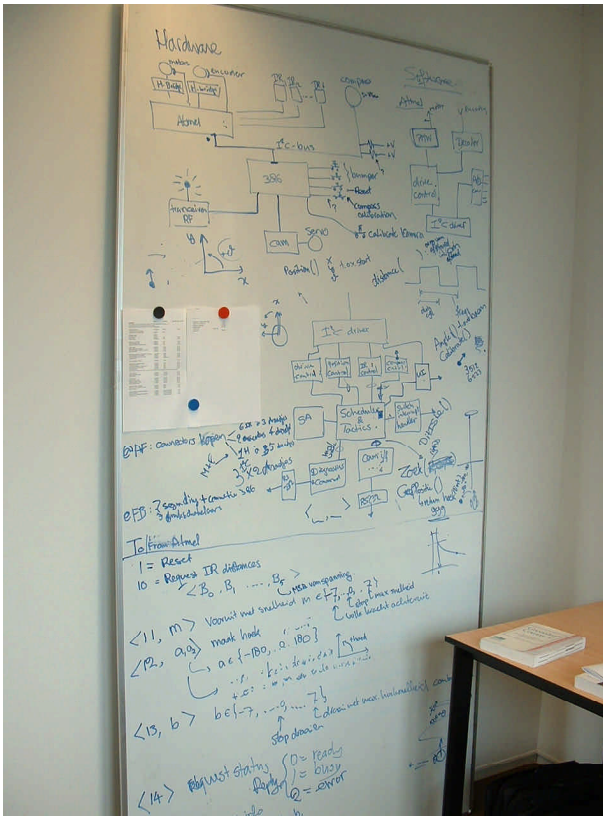

Figure 1: Definition phase

A *software requirement specification* translates customer requirements into the domain of software, a vision from where software design can start. In our case, we needed a vision on how our robot could accomplish all missions. The Battle Plan describes what the robot must be able to do in order to complete the missions. This already works towards to a solution as there many different ways to complete a mission. As an illustrative example, in last year's event, one team built a zeppelin kind of robot that would just fly over all obstacles in the maze.

The *construction manual* then describes how the robot can perform the functions laid down in the battle plan. This steers both hardware design (size, wheels, power required) as well as software (how "intelligent" must this be, what are timing issues). In our case we figured that building a robot from scratch requires a lot of work, so to be on the safe side we decided to make the robot as simple as possible. As our strength is building software, we would make the mechanical part of the robot as simple as possible and solve any problems that would cross our path in software. In hindsight, this turned out to be a good decision but even so it was based on a hidden assumption that was violated almost weekly, namely that simple hardware always works.

Lesson 1
Things you buy never work as advertised.

One particular problem illustrating this is our I^2C bus. The I^2C bus is the communication backbone of our robot as it connects our tactics processor (an on- board 80386 PC) to our motor controller (an 8051 based microprocessor). The 386 we bought had built-in I^2C support so the only thing we had to do was to connect the wires. Wrong. It took us a couple of weeks to realize that the I^2C clock frequency generated by the 386 firmware was too high for the microcontroller to absorb. The fact that the 386 firmware did not report a good status on its functioning (it always reported success) made things even more difficult to find[1]. The solution

[1]One can argue whether firmware must be regarded as software or hardware. I think any hardware engineer would say it is software, but being "high level software engineers", little black boxes that fail are a hardware problem.

came from the hardware supplier who suggested decreasing the processor speed when doing I^2C communication. As software engineers we are used to asking for more processing power, more memory, and more disk space. Intentionally slowing down the processor sounded like a bad idea, but turned out to be a good example of out-of-the-box (our box) thinking.

But let's go back to the definition phase. Part of any project initiation is making a *Project Management Plan.* The idea of being managed in the weekends was not very appealing so we decided to settle for a bill of materials with associated costs and a planning. The budget part we got nailed down fairly quickly once the Battle Plan and the Construction Manual were in their first revision. The planning on the other hand quickly turned out to be extremely off, both in effort and in duration. We can point out various reasons for all that, but in the end two lessons sum it all up nicely:

Lesson 2
If you think building a robot takes a lot of time, it takes three times more.
If you think building a robot is easy, do not start.

Lesson 3
If estimating duration is difficult, estimating duration for spare time activities is near impossible.

After the first month of development, we decided to stop tracking progress and stop re-planning. We would just go ahead and see where that would get us. As is not uncommon in these kinds of competitions, most work is done the night before the event, when the pressure is at its top. We were determined not to get into that kind of situation but at this point in time we are seriously taking such a scenario into account!

Putting things together (Construction)

Although all our five team members are software engineer by profession, some of them have an education and hobby in mechanical design and electronics (which was one of the reasons they took part in the project anyway). The challenge to build a robot was not a complete jump in the dark, but we clearly did not have the professional experience to be able to design and calculate all relevant parameters up front. We would just try and find out.

Figure 2: Our robot

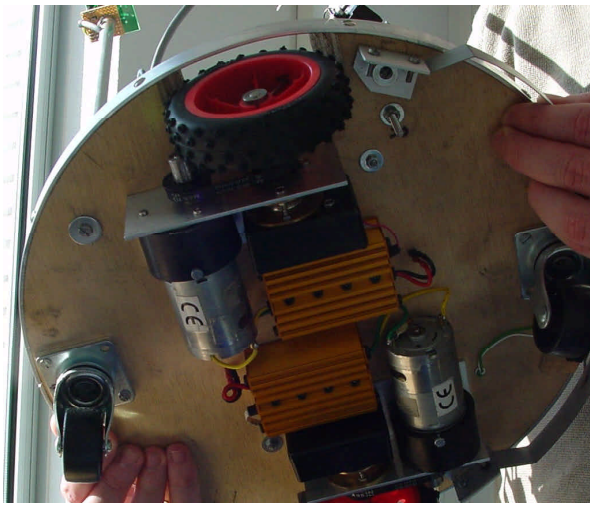

Figure 3: Bottom view

In general this strategy worked out remarkably well, but in one particular case it still causes troubles. To

decide on the motor to wheel gearing, it is important to know what the speed of the robot needs to be. Obviously, to score many points, it has to be as fast as lightning, but driving fast for example means coping with excessive decelerations during emergency stops. The idea was that by regulating the power to the motor, in software, we would be able to drive at different speeds. The optimal speed was to be determined empirically during testing. In practice, it turned out that our robot cornered too fast to manage its behaviour consistently. Supplying very low power to the motor means that robot has low torque and that in turn means that on some surfaces, our robot would not turn at all. So, a gearing decision at the beginning of development still causes our robot turning behaviour to be very much dependent on the surface texture of the playing field. Solving this issue would be to replace some pulleys and drive belts, but such a big overhaul would take a lot of valuable time and the risk of doing harm to an otherwise good working robot would be too great. We will have to deal with its shortcomings in some other way.

Lesson 4
Refactoring hardware is much more difficult than refactoring software. This implies that a hardware-related decision will have great impact on the rest of the project.

What you see is what you get (Testing)

From the very beginning it was clear to us that without experience in robot building we could easily think of great solutions that would turn out to be useless in practice. To compensate for our lack of experience at the start of the project, the idea was to quickly build up this experience by thorough prototyping and testing. We put a lot of effort in building a full scale test environment, in fact, the test environment was ready even before all robot components had arrived.

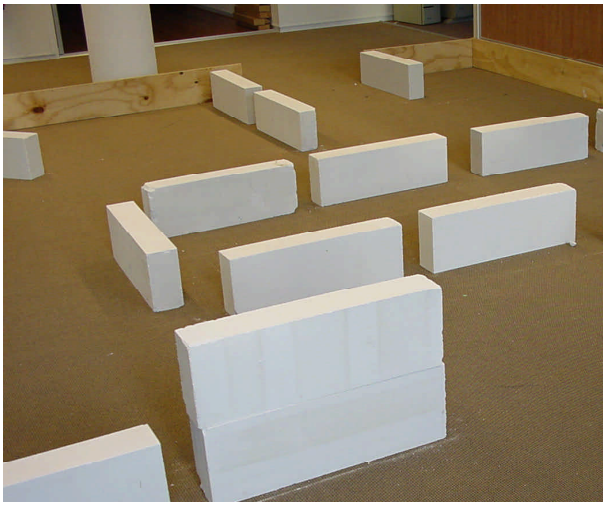

Figure 4: Test environment for mission 1

Figure 5: Test environment for mission 3

Was it worth the effort and money? Having seen robots perform during the test mission we tend to think so. One of the most heard exclamations was a desperate “what is it doing now?” That sounded familiar to us. The difference is we had those during our in-house testing, when there were no precious points at stake.

Lesson 5
In hardware, testing really pays off.

All software engineers know that “program” testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [2]. This pearl of wisdom is often used as an argument to put less effort in testing, and to develop proven correct programs to start with instead.

In this case, however, where the complexity of real world could not be modelled adequately (by us at least), testing indeed proved to be a very effective way to find bugs. Actually seeing the robot perform in its environment makes you realize your error within seconds.

Lesson 6
In the real world, analysing is good, prototyping is better.

So testing and debugging were considered essential in our project from the start. The very first software module created was indeed a diagnostics module that could log all kind of events and values to a terminal or a file if required. However, the very first time we asked a hardware-knowledgeable colleague to help us with some hardware problems, his first remark was "where are the measuring pins, where can I attach my oscilloscope?" Now who would think of that?

Lesson 7
Doing hardware implies using hardware debugging tools.

Evaluation

Looking back at all the things we have learned, there is one thing that sticks out. In our daily work we develop software for large, complicated machines, where we take all the hardware for granted. By building a robot from scratch, our respect for the hardware engineers has definitely increased.

Interestingly, we met teams with a more mechanical background with similar experience. They built the most beautiful robot but after 3 minutes driving around seemingly randomly made them realize that there is more to writing software than typing a few lines of code.

Have we done the right things, or are there more lessons in store for us? We will see on Competition Day, November 22!

References

[1] http://www.tno.nl/instit/fel/felnews/nl/robotcompetitie.html (in Dutch).

[2] The Humble Programmer, Edsger.W. Dijkstra, Communications of the ACM 15 (1972).

About the author

After graduating from the Technische Universiteit Eindhoven in 1990, **Emile van Gerwen** joined KPN Research where he developed optical character reading software, specialising in reasoning with uncertainty. In 1998 he joined the National Aerospace Laboratory where he worked on multi sensor data fusion and real-time decision support systems. He now works as a senior software engineer and consultant for Imtech ICT, where he develops software for advanced machines. Emile can be reached at emile.vangerwen@imtech.nl.