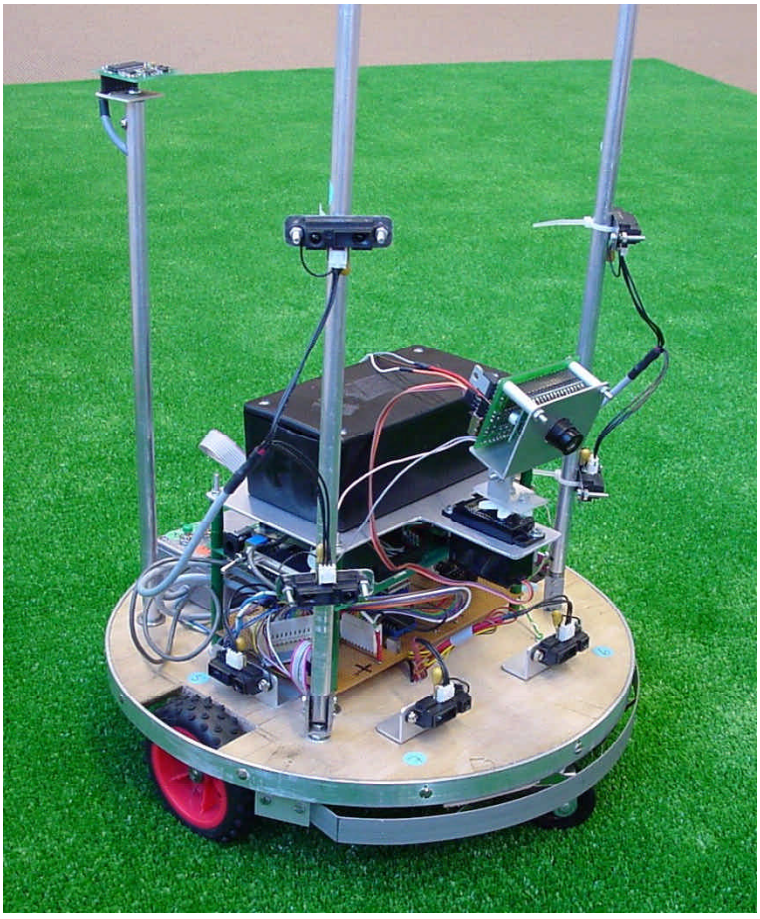


X00TIC
magazine

November 2003-Volume 10-Number 5

POST-MASTERS PROGRAMME SOFTWARE TECHNOLOGY

Multidisciplinary Development

Robotica

Discipline borders

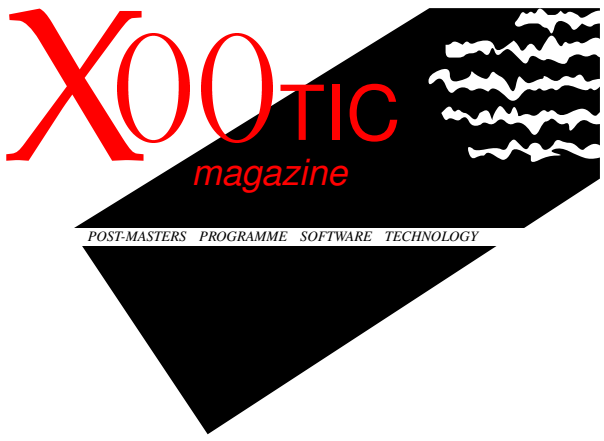
Software-hardware

Daily practice

The future

Contents

Advertorials



Colofon

XOOTIC MAGAZINE
Volume 10, Number 5
November 2003

Editors
C. Delnooz
N.H.L. Kuijpers
Y. Mazuryk

Address
XOOTIC and XOOTIC MAGAZINE
P.O. Box 6122
5600 MB Eindhoven
The Netherlands
xootic@win.tue.nl
http://www.win.tue.nl/xootic/

Secretariat OOTI
Mrs. M.A.C.M. de Wert
Post-masters Programme
Software Technology
Eindhoven University of Technology, HG 6.57
P.O. Box 513
5600 MB Eindhoven
The Netherlands
tel. +31 40 2474334
fax. +31 40 2475895
ooti@tue.nl
http://wwwooti.win.tue.nl/

Printer
Offsetdrukkerij De Witte, Eindhoven

Multi-disciplinary Development

Editorial Preface

In today's systems, embedded software becomes increasingly important. Because of its tight coupling with the hardware, developers of embedded software need to co-design with engineers of other disciplines, such as electrical- and mechanical engineering. All those engineering disciplines have their own way of looking at design, their own "best practices," their own design methods. Development team face the interesting challenge to overcome the boundaries posed by these differences and start to profit from the different views on a system.

In this magazine, the authors focus on the typical challenges faced in industry and the possible solutions people have developed.

Emile van Gerwen relates the lessons he and his fellow engineers learned while building a robot from scratch. The cooperation proved enlightning for both software- and hardware engineers and increased the mutual respect for eachothers discipline. The quality of their multidisciplinary effort will be measured on November 22, when the TNO Robot Competition is held.

In the second article, Kees van Overveld looks at the multidisciplinary challenges from a more academic point of view. He will talk about the challenges faced when designing across the disciplinary borders.

The next two articles are personal observations of two experienced architects. Ton Kostelijk and Marcel Boosten explain how multidisciplinarity influences their daily work.

Finally, Wim Hendriksen closes the magazine while looking ahead. In his article, he addresses some of the challenges faced while developing an increasing number of embedded systems and how research and education can contribute to the quality of those systems.

Enjoy reading this magazine!

Chris Delnooz, editor.

Advertorial: Thales

Page 4 (should be even)

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.

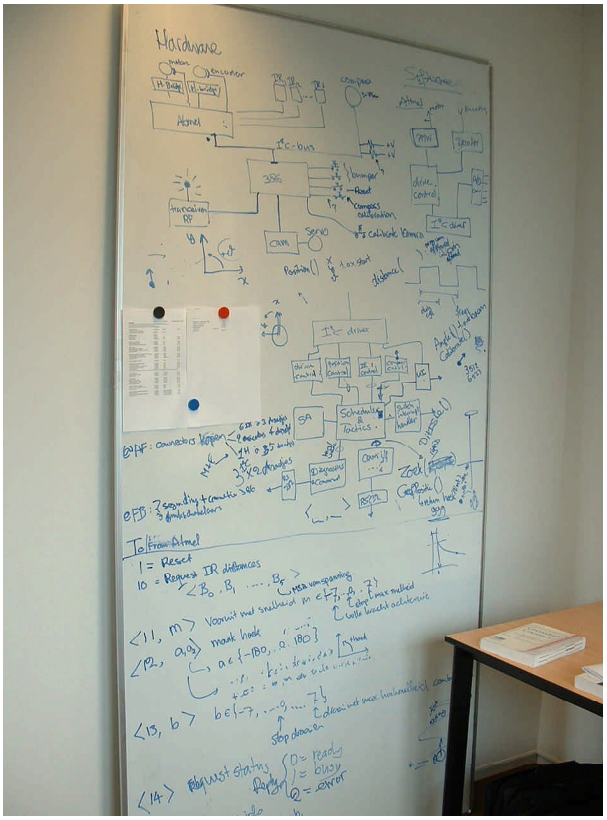


Figure 1: Definition phase

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

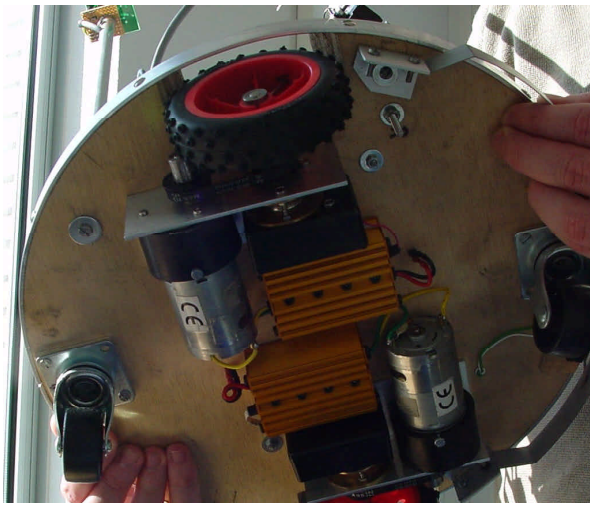


Figure 3: Bottom view

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.

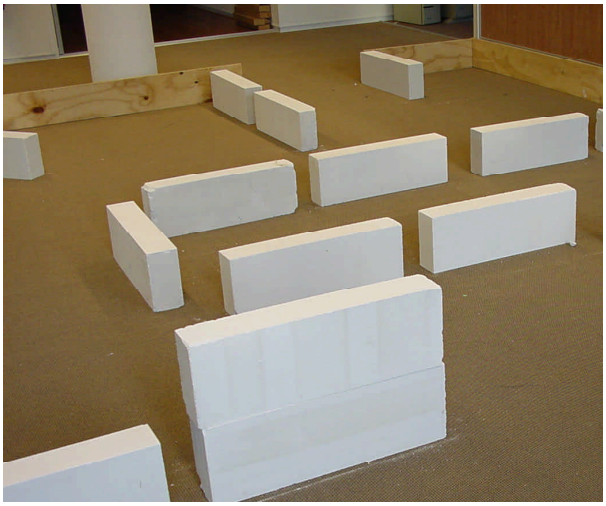


Figure 4: Test environment for mission 1

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.

Advertorial: Philips

Page 10 (should be even)

Designing across discipline borders: obstacle or option?

Kees van Overveld

Educating technological designers is difficult. An educational curriculum should provide both sufficient discipline-related skills, and cross-disciplinary skills. In order to argue about the balance between the two, we speculate on the relation between disciplines and application domains, and we give some considerations as to what disciplinary baggage gives the best preparation for prospect interdisciplinary designers. Finally, we hint at a particular role for software designers in the process of designing across discipline borders.

Introduction: Domains and Disciplines

In the past, life was easy. Professions could be easily distinguished. If John Smith was trained as a carpenter, it was clear what to expect from him. You should see John if you needed a table or a garden fence, but if you were suffering from a headache you should go to his cousin Peter who studied medicine. And if you wanted to divorce from your husband, you should consult his other cousin Charley who went to law school.

Lawyer, medical doctor and carpenter are professions with a relatively constant definition. They have quite a long-standing history, and their educational programs stayed largely constant for extended periods.

Nowadays, examples of such stable connections between education, career, and professional activities begin to be rare. Over the last, say, 10 years we have seen both a multitude of new professions, and a multitude of new educational programs. Further, the mappings between education and professional career, between initial and final job within a career, and between a job and the tasks and activities within that job, are no longer one-to-one.

In order to argue about the causes and consequences of this decreasing transparency in education programs, careers, jobs, tasks and activities, we need some vocabulary. In particular, we want to talk about *domains* and *disciplines*.

We call a coherent set of social needs or desires a *domain*. For example, transport is a domain. It comprises the desires of people to go from A to B, the need to maintain roads, the desire to have a dense and reliable network of petrol stations, et cetera. The adjective 'social' refers to the fact that any need or desire is attributed to (a group of) people. The type of coherence in a domain is often historically determined, and it may have a degree of arbitrariness. It may change over time, and as a result domain borders are fluid. Domains are not necessarily disjoint[1]. As an example, the various departments in a national government as a partition of the national concerns form a set of domains.

We call a coherent body of (professional) knowledge, skills and attitudes a *discipline*. For example, physics is a discipline. A discipline refers to a topic that is studied in a scientific community. Although to some extent historic whimsicalities influence the scope and contents of any given discipline, the coherence in a discipline also results from knowledge

[1]Overlapping domains are a frequent source of envy among professionals with different disciplinary backgrounds. Social needs and desires that do not (yet) belong to a well-recognized domain, on the other hand, often go unnoticed for a long while and once they are accepted they may cause new disciplines to occur.

hierarchies (see below). Disciplines, with some delay, give rise to academic curricula, and they can therefore be mapped approximately to the disciplinary baggage of recently graduated practitioners in any field. Notice, therefore, a self-stabilizing mechanism, such as depicted in Figure 1.

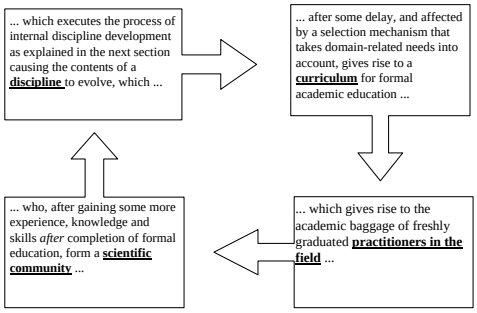


Figure 1: The evolution of a discipline.

We will study the nature and the relation between domains and disciplines in the section below, and we zoom in for the case of technological disciplines. Then we discuss the (mis-)match between domains and disciplines, and we conclude with a possible approach to remedy the problem.

The nature of disciplines and their relation to domains

Disciplines, as defined above, possess a structure that is vaguely hierarchic. A body of knowledge K is rarely self-contained. It assumes fragments of knowledge that are outside that body; these fragments may be part of another body of knowledge, say K'. For example: designing wireless telecommunication systems assumes knowledge of modulation. The topic of 'modulation' assumes knowledge of high frequency oscillators. The topic 'high frequency oscillators' assumes knowledge of linear networks. The topic of 'linear networks' assumes knowledge of linear algebra. The topic of 'linear algebra' assumes basic algebra. Finally, the topic 'basic algebra' assumes 'logic'. Here, the proposition 'A assumes B' is considered to be a partial ordering. It means that in order to actively use the knowledge (and the implied skills and the implied attitudes) in A, it is necessary to believe that the knowledge in B is both available and true; further, a practitioner of the knowledge in B is, for his present purpose, not interested in, nor dependent of the knowledge in A. This partial ordering model for bodies of knowledge is overly simple and it has some fundamental problems[2] but it gives us a convenient vehicle to argue about the complexities and possible remedies of multi-disciplinary design. The above example gives rise to the fragment in figure 2 of what we will call the DAG of knowledge or K-DAG (DAG = directed a-cyclic graph). A node in the K-DAG is a small chunk of related knowledge; an arc represents the 'assumes'-relation.

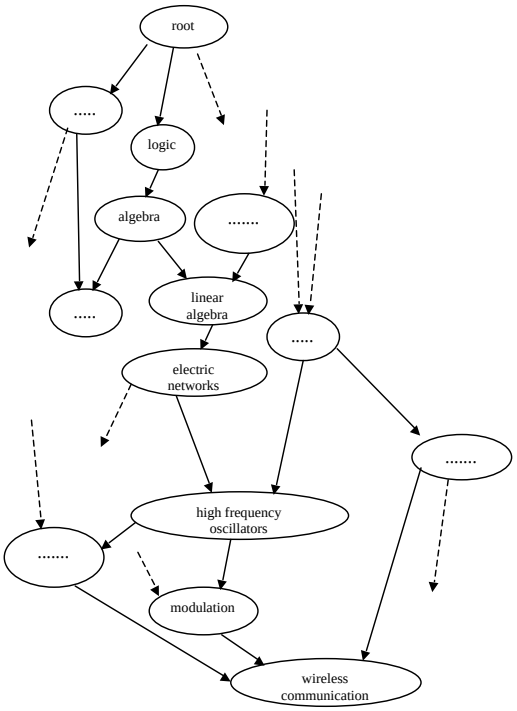


Figure 2: A fragment of the K-DAG.

[2]To mention but a few: it is not certain what constitutes the topmost chunk of knowledge (labeled 'root' in figure 2); it is not clear if there is such a thing as 'a consistent chunk of root knowledge'; it is not certain if loops in the assumptions can always be avoided, and it is not certain if the 'A assumes B' relation, apart from the knowledge components, also can be formulated for skill and attitude components of a discipline.

Nodes in the DAG close to the root represent generic notions that underlay much of scientific practice, such as 'observation', 'hypothesis', 'explanation', 'model', 'definition', 'approximation', 'causal relation', as well as formal reasoning and logic. Much of what is usually called 'common sense', as well as many abstract patterns for problem solving also reside in these nodes. Few disciplines, except from philosophy, some sub-fields of mathematics and some branches in cognitive psychology give explicit attention to these notions. In other disciplines, they are assumed implicitly, and usually they are not part of formal education. The abilities, represented by these knowledge fields are also difficult to asses by exams or tests, and therefore at present they play only a small part in our understanding of 'academic abilities'.

In the area of knowledge engineering, attempts are made to formalize (parts of) the K-DAG, including (some) root-like nodes, in terms of formal ontology - with the eventual aim to have disciplinary knowledge represented in knowledge bases that can be consulted by dedicated software applications (see for instance http://www.steplib.com). For our purpose, we will only use the terminology of graphs to argue about education and design practice.

A 'discipline' can now be defined, more precisely, as a connected sub-graph of the K-DAG. It seems likely, however, that a discipline is typically not the result of some august body of experts, drawing endless ellipses and arcs on a huge sheet of paper. It is interesting, therefore, to speculate on the evolution of a discipline - that is, to zoom in the mechanisms that are hidden in the various quadrants in figure 1.

We can imagine that there are two basic types of mechanisms involved in the evolution of a discipline. The first is the internal evolution (upper left quadrant in figure 1). When seeking answers to 'why' or 'how comes' questions, a chunk of knowledge A asks for the connection to, or the development of a chunk B with which it has an 'A assumes B'-relation. Similar (although maybe less frequent) a question such as 'what can we do with this' may lead to an instance of the opposite relation. In any case, the internal evolution maintains connectivity; it is a local process in the sense that the K-DAG grows with one arc at the time.

The second process can be called external evolution. This corresponds to the upper right quadrant in figure 1. It comes from the interplay between disciplines and domains. We remember that a domain also has an internal coherence, but this comes from social needs, governmental arbitrariness, or historical chance. The set of domains definitely does not have a DAG structure similar to disciplines - if it has any meaningful structure at all. Nevertheless, the persistence of domains causes certain needs to occur in matched combinations. For instance, in the domain of transport, the initiative of projecting a highway raises simultaneous questions in the disciplines of geography, civil engineering, economy, meteorology (with respect to the environmental consequences of the CO_2 production of the traffic on the projected road), and ecology. The merging of such (initially separate) disciplines into one new, 'multidisciplinary' discipline as a result of external evolution can be depicted, schematically as in figure 3.

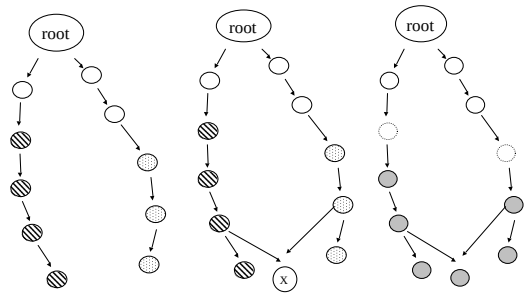


Figure 3: Three phases in the external evolution of disciplines.

In figure 3, we depict three stages of the evolution of a discipline as triggered by external factors.

In the leftmost diagram we have the initial situation. Two separate disciplines exist, both depicted by their (schematic) K-DAG. One discipline is indicated by line-textured ellipses, the other one by dotted ellipses. For simplicity, the disciplines have been drawn as simple chains of knowledge domains; in general, they will consist of many branching chains. Notice that typical disciplines don't contain the 'root' knowledge domain, nor knowledge domains close to the 'root'. The non-textured ellipses represent the 'implicit' knowledge that is assumed to underlay the disciplines, but that is not

part of the formal curriculum, and therefore it is not part of the *explicit* working knowledge of workers[3].

In the second stage (middle diagram in figure 3) a chunk of knowledge, labeled 'X' appears from an application domain. It assumes knowledge chunks from both disciplines. If the 'A assumes B' relations can be successfully established, and this circumstance appears sufficiently often, the third phase may occur.

In the third phase (rightmost diagram in figure 3), the two formerly separated disciplines have been merged into one new discipline (indicated by gray ellipses). For instance, if the two former disciplines were 'mechanical engineering' and 'logistics', then the new discipline could be 'transport sciences'. Notice that some ellipses, that were formerly textured, now have become blanc (indicated by dotted outlines). This is because, as we saw earlier, a discipline is carried by a curriculum, and a curriculum is the result of a resource constrained design problem. The constrained resources, in this case, are time (between 4 and 9 years for various tracks of scientific education) and learning capacity. In phase 3 we see that typically those knowledge domains are sacrificed that in the original disciplines were closer to the 'root'-nodes. The curriculum for the new discipline must contain the node formerly labeled 'X' and related nodes, and therefore doesn't have sufficient room left for the most upstream nodes in the original curricula.

This may cause a significant problem. Due to their nature, the domains close to the root are quite abstract, and many are widely applicable. Omitting these nodes makes it more difficult, in particular in freshly graduated practitioners, to see the underlying relations between (application) domains, and therefore learning these knowledge domains later (=after completion of the formal education) takes more effort. In phase 3, taught knowledge, skills and attitudes have the risk to be of a rather encyclopedic nature - which falls short in complex situations where deep and abstract understanding are required.

Disciplines in technological education

The above observations hold in arbitrary fields of science. For technological sciences, there are some additional conditions. First, the range of notions in the 'root' node and nearby nodes in the K-DAG is less broad than in general science. Many patterns can be identified that have proven adequate in a large range of situations.

Many of these have been formalized in terms of mathematical notions (e.g., 'function', 'discrete vs. continuous', 'variable', 'operation', 'state', 'singularity', 'graph', et cetera). Apart from their precise, technical meaning, they have an important value as metaphors.

For instance, even though 'monotonicity' is a formal property, typically applied to mathematical functions on sets of numbers, it is insightful to use this term in economical or even psychological contexts to indicate 'that something develops in one direction only'.

As another example, we have used the terminology of directed a-cyclic graphs in this paper to facilitate arguing about the development of disciplinary knowledge. It is important to notice that we don't (necessarily) use these terms for the purpose of technical manipulation, but we use them because of their metaphorical intuition. For instance, using the term K-DAG made it natural to talk about 'root-nodes' and 'partial ordering' without having to spend much words in explaining what these things mean. Furthermore, it invites us to think of algorithms for 'merging' and 'traversing' DAG's as metaphors for understanding the process of discipline development. Since terms such as 'merging', 'traversing', and the numerous other concepts from mathematics and computing science come with a cloud of useful associations[4], they provide convenient communication shortcuts that avoid a lot of potentially confusing verbose prose.

Moreover, once they are used in the early, exploratory phase in a communicative process (such as a design process), where terms are not yet formally defined, there is a large chance that the same terms can be used later when things get more pre-

[3] Which is not to say that they don't use this knowledge; they just don't know how to talk about it.

[4] These associations are useful in a large variety of contexts, otherwise the terms would not have gained the status of broadly used abstract notions.

cise - including (some of) their technical connotations.

Second, practitioners in technical disciplines have at least some familiarity with formal argumentation. They are trained to formally manipulate with terms and symbols in math courses. This means that the habit of using precisely defined terms comes more natural to technologically educated professionals - even when, in early stages of exploring a topic, the formal manipulations with such terms is not yet in order. There is a rich potential of problem-solving patterns hidden in the usage of precise terminology - if only this potential is recognized and stimulated by teachers and adopted and practiced by students.

Problems and solutions

After having studied the notions of disciplines and domains, and having explored the mechanisms of discipline development - both under internal and external factors - and having touched upon the particular circumstance of technological disciplines, we now arrive at the main theme of this paper.

Designing across discipline borders is a difficult process. It resembles the external evolution of a discipline as outlined in figure 3. The design problem at hand corresponds to the (application domain-induced) node 'X'. The disciplines that need to be connected are the bodies of disciplinary baggage of the involved designers. These disciplines seem unrelated because both involved designers lack a sufficient shared body of underlying, more abstract notions. As with merging disciplines, this is again a result of resource constraints (limited time, knowledge, effort and expertise of the designers at hand).

The resulting complication is most often called a 'communication problem', but it would be more appropriate to call it a problem of lacking shared, sufficiently abstract thought patterns[5].

In an attempt to remedy such 'communication problems', new 'interdisciplinary' curricula are presently developed. The underlying idea is probably that, as soon as an application domain is embedded within a discipline, and hence a curriculum is developed for this new discipline, the problem of designing across discipline borders vanishes. Indeed, in this new discipline, both earlier disciplines have been integrated, the discipline border has vanished, and the problem has gone away - or so it is hoped.

From our analysis, however, it seems that this argument is flawed. Rather, we think that the problems with interdisciplinary design increase when new 'interdisciplinary' curricula arise. Indeed, due to the resource constraints that are inherent in any curriculum such very broad and interdisciplinary curricula are increasingly devoid of the root-like nodes in the K-DAG. The ultimate version of such a trend would be that there is only one (technological?) discipline left, which would include all possible application domains. Then there are no discipline borders left, and hence no problems of cross-discipline design. In the limit case of such an ultimate interdisciplinary 'discipline', however, there would be hardly any room for abstract thought patterns, and as a consequence, there would be hardly any insight in underlying relations between (application) domains.

Instead, we recommend a more paradoxical remedy to prepare designers for interdisciplinary design challenges. Rather than spending large amounts of curriculum space to application domain-related knowledge, we propose to increase the amount of fundamental ingredients. This includes formal notions and mathematical and logical techniques. Notice: this should not be mistaken as a recommendation for 'more math'. Rather, it is a recommendation to focus on explicating thought patterns and problem solving strategies. A vehicle could be to study the intuitions behind mathematical notions, to practice with designing and studying models for the sake of understanding the methodology of model making, and to exercise definition-making skills in order to perfect precision and exactness in the expression of ideas, assumptions and propositions. Because of their abstraction and wide applicability, these skills seem to be the best candidates for dealing with arbitrary cross-disciplinary design and engineering problems.

A natural question would be what effects such an approach to cross-disciplinary design could bring. Since the distance to domain-specific applications is larger than in many 'interdisciplinary curricula', a curriculum according to the above recommendation may not be a fail-safe recipe for spectacu-

[5] Perhaps many communication problems are just the lack of sufficient shared, abstract, underlying notions.

lar innovation or for revolutionary new products. Indeed, ideas for new products often come from workers close to application domains. If such experts have less familiarity with fundamental issues, however, a thorough understanding of the underlying principles may be an underestimated ingredient - which can cause overstrained expectations and disappointing performances of hastily designed products. Rather, our recommendation for a more foundation-oriented curriculum to educate designers to work in cross-disciplinary contexts could give rise to well-structured, consistent and smooth design processes that are less hampered by communicative noise.

An option for computing science

Above we gave a recommendation that applies to curriculum design. There is, however, another route to mitigate the problems of designing across discipline borders. As follows:

Among all the sciences, computer science forms a peculiar case. In any other science, a scientific argument is judged for its convincingness with respect to (human) colleagues. In computer science, a scientific argument (e.g., an algorithm) is judged for its convincingness with respect to a formally defined machinery (namely, an (abstract) computer or some other formal framework).

This has a major consequence. In all sciences except for computer science and mathematics, there is a large amount of interpretation involved in assessing the validity of an argument. Even in empiric sciences, where so called objective observations are the cornerstones of progress in understanding, dealing with such observations often leaves room for interpretation. Interpretation, in turn, leaves room for misunderstanding, confusion or ambiguity.

A computer cannot tolerate ambiguity, and therefore a computer program cannot rely on interpretation. Hence computer scientists are trained to give precise and unambiguous definitions. At the same time, unlike some branches in mathematics, computer science is involved with modeling *reality*. A computer program has a purpose, namely to add in dealing with (aspects of) a real situation, whether this is a computer controlled machine, an administrative system or a communication network.

Therefore, by their education, computer scientists possess a rather unique combination of skills, that is essential in interdisciplinary design, namely (a) to be capable to think in terms in models (because computer programs that have something to do with real systems only do so by dealing with *models* of such systems), and (b) to be capable to formulate such models in a precise and unambiguous manner (because computers require precise and unambiguous instructions).

It is remarkable that most computer scientists only exercise this rather unique combination of skills when it comes to IT-related design. The reasons for this may be several (perhaps students choose for computer science because of the prospect of luxurious salaries, to be earned with writing software; a hobby in computer programming is the main motivation for others), but it would be tremendously helpful in all sorts of interdisciplinary design if computer scientists would offer their assistance to help clarifying interfaces between (models of) knowledge domains - irrespective whether it regards mechanical, chemical, biomedical or any other disciplines.

Maybe in the light of subsiding economic activity in the software branch, this could be an interesting option for computer scientists who are not afraid to broaden their scope.

To conclude with a paraphrase of Dijkstra's famous motto 'Beauty is our business', we might give as a characterization for this new group of professionals: 'Precision is our profession'.

About the author

Kees van Overveld (1957) obtained a MSc and PhD degree in physics at the Eindhoven University of Technology (EUT). In 1985, he joined the computing science department of the faculty of Mathematics and Computer Science of EUT as a university lecturer; since 1990 as associate professor. From 1989 to 1998 he was head of the Computer Graphics group. From November 1996 to June 1998 he was part-time employed as a Senior Researcher at Philips Research; further, he still held the position of associate professor at EUT. In June 1998 he founded 'Van Overveld Coaching', a consultancy company. In

this company he works on a regular basis for Philips Research, the University of Calgary (Canada) and the University of Magdeburg (Germany). In May 2000 he left the computing science department of EUT; he changed his academic affiliation to the Stan Ackermans Institute at EUT. In 2002, he joined the Faculty of Industrial Design (ID).

Among his previous research interests are the fundamentals of raster graphics algorithms and discretisation, computer aided geometric design, interactive motion specification and (dynamic) simulation for computer animation, image processing and some aspects of 3-D computer vision. Recently, he initiated a research activity in the field of the methodology of technological design. He is currently responsible both for the teaching program and the research in this field. In ID, he is mainly involved in teaching mathematical modeling and structured creativity-related techniques.

Some personal notes on multidisciplinary development

Ton Kostelijk

This text contains notes on my experience with some disciplines: software and mathematics, embedded software and hardware, proceeded by a more personal introduction.

Several months ago, I was invited to put some words on paper about my prejudices of different disciplines. Thinking about it, and extending it to the cultural issue as well, I realize that I have crossed a lot of "cultural" boundaries in my profession and because of my personal background.

I was born in 1961 as the 11^{th} child of an agricultural farmer, a religious family, in a small village in a polder (North-Holland). In particular here in Brabant, I still miss the wide outlook over the land 20 km far, the waters and wind that surrounded me there. With the sixties and growing welfare, a lot changed in my first 20 years. My eight brothers in particular were quite outgoing and paved my path to be allowed a lot of freedom in behavior. Already with my scooter, I visited my eldest married sister at the age of 4, a trip of 5 km away (I was very well taught to duck to the side when a car passes). As one of the very few of my town, I was educated at the Athenaeum, in the nearby (12 km) city. I realize now that it was the first time of many that I joined a new group, where many of my old group did not follow that step. In the city, things go very differently. My circle of movement kept on growing, and I decided to go to the Free University in Amsterdam, to become a scientist.

In different ways, most of us cross cultural boundaries, even when you are Dutch, and you live in The Netherlands. Differences between groups are an interesting subject to discuss, provided that it is done with the willingness to understand. When done in a reproaching or even accusing way, there are only losses. However, in the end the differences between individuals are larger than the differences between groups. So I propose you read the remainder with some relativism.

My education (experimental solid state physics) combined a strong mathematical foundation with pragmatism and a sharp eye for deviations in expected behavior. In short, physics is all about constructing, rejecting, validating and extending models of the experimental reality. Mathematics is about models of the imagination. When an experimental model is equivalent to an imaginary model, knowledge of mathematicians can be used.

Software and mathematics

Some people think that programming is a form of applied mathematics. In my view software originates more from engineering than from mathematics. I respect Dijkstra for his contributions but I disagree with his rigid view on software development. Even Gödel's incompleteness theorem has proven that proofs and completeness are not fully united. Still there are people that keep on searching for the ultimate software development method or tool that will enhance productivity enormously, and prevent any mistake that happened in the past. Instead of posing the question "what is the best method for all systems?" one should pose the question "what is the best approach for my system?" In this way, mean and lean solutions most likely occur. First of all, one can benefit of the intrinsic structure of the problem at hand. Secondly, the notion "approach" instead of "method" indicates which route to take without

claiming to know the solution. The typical problem is too difficult to tackle in one go, leading to another element: usage of an incremental approach, or in other words, a spiral development model. And last but not least, the job is performed by a (set of) group(s) of people, where experience, communication and other cultural items play an important role.

Nevertheless, one can benefit a lot from mathematical insights, just like physics benefits from it, without implying that physics is a sub-domain of mathematics. Since problem / system modeling is apparently so important, it may no longer be a surprise that so many software architects originally studied physics.

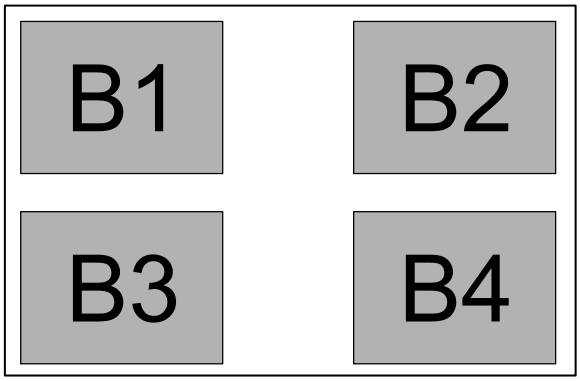


Figure 1: Typical hardware view of a system

Embedded Software and Hardware Development

There are several differences between hardware and software.

First of all, the definition of a system is quite different for hardware or software. For a hardware designer, a system is a collection of hardware blocks that are interconnected. For a software designer, a system consists of several layers; an application (including services) runs on drivers where drivers are "almost hardware." Whereas for hardware designers interrupt routines are considered out of their scope. This implies that a gap in responsibility exists, the hardware software interface.

Secondly, embedded software is mostly engaged in handling use-case transitions, whereas hardware designers are focused to make a block process well in a steady-state use case.

Thirdly, hardware typically is designed bottom-up ("re-use") whereas software is typically designed top-down. This latter statement may no longer be completely true: because of high development cost, software re-use is growing rapidly, resulting in more and more glue code in systems. One may wonder whether the amount of glue code is in balance with the shielded amount of core code.

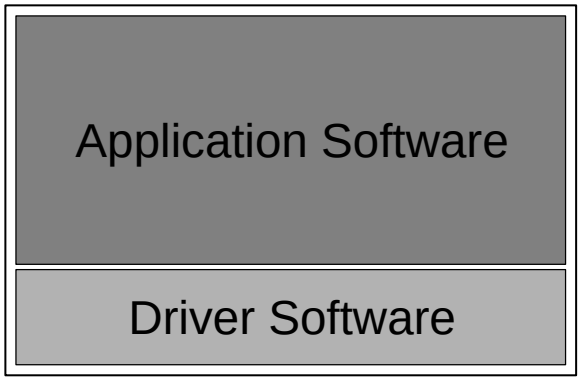


Figure 2: Typical software view of a system

Fourthly, hardware has real concurrently running functions. Software only has timesharing, where concurrency is faked by the operating system. The software performance is deteriorated many factors by limited caches, uncached access, context-switches (interrupts in particular and task-switches) and busload. As a result, hardware designers typically overestimate the software performance of a system. The average software designer is not at all focused on performance whatsoever. This means that performance set-backs are common when execution architecture view is not elaborated in a system.

Fifthly, one of the major separating issues is that software development lags behind one or two generations of the hardware development. In other words, the hardware and software focus differs.

Finally, software suffers from hardware bugs, not vice versa. On average, 30-50% of the effort to make drivers is related to bug fixing, unclear specs, system errors, etc. One of the opportunities is to keep the overall balance sound: whether to save effort in hardware design by spending effort at the software side, or vice versa.

Final remarks

The cliché that technical software people talk to computers only and work in isolation is so beside reality. This paper could have grown many pages more because making products involves so many different disciplines. In the global world we live in, it involves many different cultures and nationalities as well. In our work, this gives extra color to our profession. It is also the reason why the educational focus of an architect shifts from technical towards social during his or her professional life. Since the differences between individuals are larger than the differences between groups, it helps to forget the group your colleague might be part of, and appreciate the unique person you are in contact with. In case of misunderstanding, the group habits may be a cause though, but firstly focusing on the group clichés ignores most value of the person. Multidisciplinary work can thus become an adventure of synergy and appreciation.

About the author

Ton Kostelijk Born in 1961, Married. From 1979 to 1985 Ton Kostelijk studied physics, Experimental solid state physics with IT, at the VU Amsterdam. From 1985 to 1995 he worked at Philips Natlab on CAD for VLSI Design, for which he received his Ph.D. in 1994. Thereafter, he was Chief software archtict Digital Receivers program (G+4 settopboxes) at Philips' ADC/ASA until 1999. Currently, Ton Kostelijk works as system architect at the Philips Digital Systems Lab in Eindhoven.

Currently engaged in

- System Performance Feasibility
- Member of Core Architecture Team Disk Systems product family.
- Coaching of architects
- Chairman of "QITARCH" of PDSL
- Teaching courses for CTT (3), ESI, OOTI.

The Multi-Disciplinary Aspect of System Development

Observations from Daily Life

Marcel Boosten

Within our daily professional lives, many of us feel that multi-disciplinary cooperation and development is a crucial aspect of the system development process in which we participate, and consider it of vital importance to the success of the developed products. In this article, I will report some observations on multi-disciplinary development from my own daily practice.

Multiple disciplines

In my current role as technology manager and project architect of the Volumetric Imaging product at the Cardio Vascular Product Management Group of Philips Medical Systems (PMS), I cooperate on a regular basis with people from the following disciplines:

- software (engineer, architect, team leader, test manager, configuration manager)
- system design (technology managers, architects, designers, norm compliance)
- image quality
- project management
- product management (neuro vascular, cardio)
- clinical science (neuro vascular, cardio, workflow), clinical application
- system engineering
- mechanical engineering
- service innovation
- pre-development (developers, group leaders)
- research

In the architectural roles I fulfill, multi-disciplinary cooperation and negotiation is daily work. More than others, architects are involved in multi-disciplinary cooperation. Even so, multi-disciplinary cooperation is regarded as something so common by many people within the organization, that it is usually not looked upon as a separate entity or activity. It is simply embedded in the organization, and naturally present in daily working life.

Multi-disciplinary cooperation

In my daily working situation, the many different disciplines cooperate via common-sense communication and negotiation. In order for different disciplines to communicate effectively, they use a shared 'language': *a language consisting of concepts, terminology, and principles that they share and understand, and that covers the topic at hand*. Typically, the shared language is naturally developed while talking about a specific problem; it is built up from concepts originating from the disciplines participating in the discussion. The developed concepts end up in specifications and designs, and thereby become the language of the team. Everyone in the team, but especially the architect, has the responsibility to keep the shared language simple and understandable for all involved disciplines. Often, drawing pictures helps in supporting the multi-disciplinary documentation or discussion. Personally, for multi-disciplinary communication, I strongly believe in the power of artistic

free-style pictures. I'm convinced that in most situations free-style pictures are more powerful than standardized (e.g., object-oriented) diagrams, especially because the free-style pictures allow you to artistically express the concepts at hand, and because standardized semantics are only remembered by experts from software development.

I often experience that people from other disciplines are very much willing to present their views in simple and understandable terminology. I remember visiting Prof. Moret, one of the world's top neuro radiologists. While Prof. Moret was at work, a colleague neuro radiologist commented on Prof. Moret's work. He used terminology he knew we would understand - similarities between plumbing and Prof. Moret's life-saving work were regularly made. I'm convinced that people who really know what they are talking about, are also able to express their ideas in simple down-to-earth terms. Keeping it simple is a key to success.

In general, in system-level requirements documentation and corresponding discussions, I try to keep everything understandable for all involved disciplines. I thereby avoid having to actively maintain different views on the same topic for different stakeholders - this reduces maintenance work, reduces inconsistencies, and stimulates discussions between the involved disciplines.

Uni-disciplinary cooperation

Nearly all the decision making teams in which I cooperate consist of people from different disciplines. However, even in case people belong to the same discipline, they typically have a different area of expertise, or they play a different role in the development process. For example, in our software team, we have experts in the following areas of expertise: platform, processing, and viewing. All three areas have their own concepts, terms, and corresponding technologies. So, even though all three areas of expertise are represented by architects from the same discipline, i.e., software, the area of expertise differs significantly. Consequently, for me, as project architect, the cooperation with these three software architects has many similarities with true multi-disciplinary cooperation.

Avoid walls between disciplines

It is of importance that the different disciplines do not create isles of isolation. They have to be open to each other, and depend on each other. As an architect, you sometimes notice signals of isolation; at that point stimulating the multi-disciplinary communication is important. Furthermore, the project organization can have a major impact on inter-disciplinary communication. Often, projects are organized as a collection of subprojects, one for each discipline. In many situations it would be preferable to organize projects according to the components defined in the system architecture, thereby grouping disciplines working on the same component in one team. This has been illustrated in Figure 1.

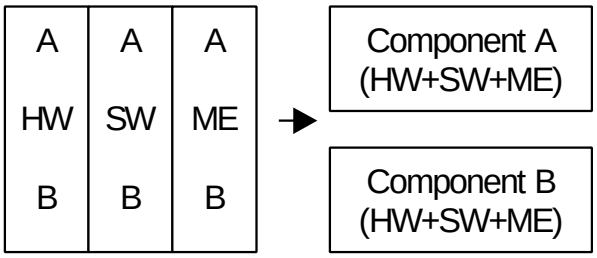


Figure 1: Reorganizing projects from a discipline-oriented structure to a component-oriented structure.

Experts on key technologies

For many products, including most of the medical systems that PMS develops, the success of a product often depends on achieving excellence in a few key system aspects. These system aspects usually are of a multi-disciplinary nature. Cardio Vascular uses an organizational structure in which individuals, called Technology Managers, are assigned to carefully selected key aspects of the system. I fulfill this role for all technologies related to Volumetric Imaging. This approach works well: it leads to focus in the organization and stimulates multi-disciplinary development in key success factors of the product family. On a smaller scale, we use the same approach. We have, for example, a team that specializes itself in 3D image quality and builds up multi-disciplinary knowledge on a critical system aspect.

Multi-disciplinary design methodologies

Even though, in my daily work, I cooperate with people from many different disciplines, I must say that we do not use any structured multi- disciplinary design methods. This is not surprising - in our team, one *development* discipline dominates: software. Teams in which multiple development disciplines (hardware, mechanics, software) coexist, and in which success depends on intensive and carefully balanced interaction between the parts developed by the different disciplines, could very well benefit from such techniques. However, I'm not aware of any development team that uses design methodologies tailored for multi-disciplinary development in practice. Some opportunities in this area might be:

- **HW/SW co-design and implementation**
  During my period at CERN, I ran into Handel-C, a programming language and toolkit enabling hardware/software co-design and implementation. It allows easy migration of code between software and FPGA firmware.
- **Multi-disciplinary modeling and simulation**
  For mechatronical designs, I can imagine that modeling hardware, software, and mechanics in a single simulation model would allow for even more advanced systems to be designed, and to be tested beforehand. Considering the pressure towards more and more integrated systems, I expect that such approaches will be picked up in the near future.

Conclusions

Summarizing, I see that:

- Multi-disciplinary cooperation occurs via simple down-to-earth negotiation and cooperation.
- The language understood by all disciplines, consisting of concepts, terminology, and principles, grows together with the product.
- Uni-disciplinary cooperation can have strong similarities with multi-disciplinary cooperation when the areas of expertise differ significantly.
- Avoid isles of isolation. Consider reorganizing projects to combine cooperating disciplines to component oriented teams.
- Assign people to multi-disciplinary key aspects of your product line.
- Multi-disciplinary design methodologies are little used. They will become more important.

About the author

Dr. ir. Marcel Boosten MTD is Volumetric Imaging Technology Manager at the Cardio Vascular Product Management Group of Philips Medical Systems. In this role, he is responsible for the technological aspects of one of the key success factors of the Cardio Vascular business. Since 1994, Marcel holds a M.Sc. in computing science from the Eindhoven University of Technology. In 1996, he finished the post-master program in software technology (OOTI). He holds a Ph.D. in software design for work performed at CERN, Geneva, Switzerland. Since 2000, he works for Philips Medical Systems.

The future of Embedded Systems

You Ain't Seen Nothing Yet

Wim Hendriksen

If you would have a choice, which time in the history of mankind do you want to live in? I would have selected today immediately. Listening to the Violin Concerto of Beethoven I write this story. One hundred years ago we had to hire a complete symphony orchestra to hear a performance of this violin concerto. That is a little bit outside the budget. What a pity that you could hear such a marvellous masterpiece only once in your lifetime. Or not at all. . .

Introduction

Last century human beings started to store and reproduce music. It was a struggle with a nail that scratched through the rills of a black plastic disk to reproduce music. We could hear the music through the rumble of the electrogramophone and we knew all too good what wow and flutter meant. When you turned up the bass and the volume, you could experience the concept of positive feedback with a loud booming sound.

Today, we can hear at home the world's finest performers - with a better quality than in the local concert hall - with our own stereo or home cinema system at any moment. Although music is just carefully moved air, you feel it is pure emotion. Your Quality of Life improves.

How is this all possible? By using Embedded Systems in our CD players, DVD players and Sound Processors. It is not difficult to find equal examples in cars, nuclear power plants, medical equipment, smart bombs, domotica or gaming devices: they cannot exist without embedded systems. Embedded Systems are invisible. You don't see them, you cannot drop them on your toes and they don't smell. Only when they don't work as expected, you notice their existence. They hide very modest in their embedding system. But what is an embedded system? In literature many definitions are found. In this paper we will use the definition of NetBSD, found in [1]:

> *An Embedded System is a combination of computer hardware and software and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, Embedded Systems are part of a larger system or product.*

Quality

One of the greatest misconceptions in the area of embedded systems is the insistence of developers to strive for zero defect products. Unfortunately, with today's means and methods this goal can be reached only in an infinite amount of time. Not one company has the financial resources to reach this goal. Of course for nuclear power plants you have different reliability requirements than for a MP3 player. But, even with all possible effort the zero defects goal still cannot be reached. So unfortunately this goal must be consigned to Utopia. It is better to accept a "just good enough" approach.

What this means is different for every product. For a pacemaker just good enough is that the pacemaker keeps the user alive during the lifetime of the product. So you need very high standards on reliability,

availability, robustness and lifetime. However the number of features on a pacemaker is limited and time to market is no issue.

We accept that a GSM phone breaks down after one or two years, and, when it locks up once a week, we know that we have to remove the battery for a few moments to make it work again. But the number of features increases year after year. Time to market makes the difference between a commercial success and commercial disaster.

What is the quality of a product? Is the quality of a Rolls Royce better than the quality of a Volkswagen Golf Diesel? No, car owners buy a product that fits their needs for a reasonable price. So value for money is an important issue, but also status, reliability, bells and whistles, image and design are important. Robert Pirsig, author of Zen and the Art of Motorcycle Maintenance [2] changed my mind by writing:

And what is good, Phaedrus,
And what is not good-
Need we ask anyone to tell you these things?

You know what quality is when you see it.

About three bugs per thousand lines of code can be found in commercial products with embedded software. And this number of lines increases exponential, so we have to live up with more and more bugs in the systems we buy. And still we buy these products. Why? Because customers do have different ideas about quality than the quality departments of embedded systems builders. Customers want the right product on at the right moment for the right price. So delivering a good product one month late is just as bad as delivering on a time a product with fewer features.

Here is an example:

Last month I bought a DVD player. After unpacking I read the owner's manual [3]. Somewhere in the middle of the book I saw the following instructions about a button on the remote control: "Press the P-scan button to activate the progressive scan feature for 480p output to a HDTV monitor. The unit powers down briefly and restarts in a new mode". I agree, this didn't look like a well thought out software architecture: I should have been warned.

On a separate inlay I read "Do NOT activate the Progressive Scan feature when the video output is set to the SCART setting in the setup menus; unstable operation of the RDV-1060 will result". So they ask me NOT to push a button on the remote control of a new piece of equipment. How long is an engineer able to resist such an order? I managed to avoid pushing the button for five long minutes. The DVD player died instantaneously when I finally did.

It took fifteen minutes before it worked again after following the "Troubleshooting instructions in case of No Power or unit freezes up after activating progressive scan in PAL systems." Maybe this piece of equipment should not yet have been shipped to customers.

But it looks good, sounds great, and displays a marvellous picture and all this for a reasonable price. So if I had to select again, would I have selected the same Rotel DVD player? Oh, yeaaah !

Challenges in Embedded Systems Design

Embedded systems are always designed in multidisciplinary teams. So people with backgrounds in electronics, mechanics, informatics or optics must work together to get the desired system. Nowadays projects are done in large to extreme large teams, sometimes divided over several sites, sometimes in different time zones, sometimes with different companies. Problems in optics are solved in software; problems in software are solved in electronics. Or vice versa, depending on the cheapest way, calculated over the whole lifecycle of the product.

For instance removing one of the power supplies from a product may save 500 Euro; with 3000 machines per year over 3 years this saves 4,5M Euro on cost of goods. You can program a lot of code for this amount of money, even though the software architecture may be not as transparent as before. (Because two motors in completely different subsystems are not allowed anymore to run simultaneously, the embedded software must perform some energy management).

First a team has to define the multidisciplinary requirements. Fortunately today processes and tooling are available to define requirements, but it only works in one discipline. So here comes the fun part: Mechanics, Physics and Electronics engineers are

used to work a little bit bottom up: "Let's start using known parts and see where we end". Informatics engineers in the meantime only want to think top down, so they refuse to think about solutions, they only want to think about requirements. In their hearts they want to start at the "oersoep," the beginning of life. After a while building this product starts with the architecture and the system design. As you can expect the bottom uppers are already far ahead, but are hindered by the top downers, who are asking nasty questions about the "what" and the "why" of the product. Things the first had forgotten to ask. With this way of working, you have a quick start, but not always in the good direction, followed by steering in the right direction. This may be the best of both worlds, only you don't learn it at school.

Somewhere halfway the project all disciplines meet again and in close cooperation the product is tested, debugged, verified and validated. When you look at such a project, it looks more like a jazz band than a symphony orchestra: everybody starts playing at the same time, everybody stops at the same time, but in between everybody plays his own solo without much attention to his peers. And still the audience can hear which song they play.

The story about top down and bottom up can be seen in a lot of embedded systems companies. Every discipline thinks that the others are complete and utter idiots, which often results in a lot of red faces, bad heartbeats, stonewalling and moaning: if everybody would use my methods, life would be much simpler. But the others have the same thoughts. Why don't we just accept that every discipline has its own design method, suited best for that discipline? So we are professionals in our own discipline. And in the meantime we must learn to communicate in the language of the other disciplines.

This is one of the reasons why formal methods never will survive in the embedded systems world: only people with informatics background are able to understand what is written down there, so nobody outside the own informatics group is able to review the output of these methods. As a result you very efficiently build the wrong system (twice), which was not the initial intention.

When you are able to understand what the restrictions of other disciplines are, then you are able to balance solutions in different disciplines. This means lots of communication in the project. Lots of communication is only possible when people are working close together. That is why projects designing mechanics in Eindhoven and software in Bangalore are doomed. But the managers don't see it until it is integration time. And then it's too late.

To probe further take a look at the Gaudí site of Gerrit Muller [4]. It gives a very good insight in the state of the art of embedded systems architecture, written by one of the most experienced embedded systems architects in the Netherlands.

Research

On a number of universities research is done in the area of Embedded Systems.

PROGRESS (PROGram for Research on Embedded Systems & Systems) wants to improve the knowledge in the area of Embedded Systems at Dutch universities and companies in order to improve the competitiveness of Dutch industry. PROGRESS has written an Embedded Systems Roadmap [5]. This book describes a vision of embedded systems and is used to steer research in the right direction. More information about PROGRESS and the running research subjects can be found on the PROGRESS website [6]. A new initiative of PROGRESS is to start with Public Outreach, which will disseminate the results of PROGRESS research to the appropriate people in companies, schools and universities. When you need more information about the results of PROGRESS research, please mail the author of this story.

ESI, the Embedded Systems Institute in Eindhoven, is organized around large Dutch embedded systems companies. Huge research projects have been started and will be started. Check the ESI website [7].

Dutch Institutes of Higher Education (HBO) have introduced so called "lectoraten". Lectorates about on e.g. embedded software, embedded systems and mechatronics are all up and running now. They mainly focus on pragmatic applied research and want to serve mainly the small and medium sized companies in the Netherlands.

In the European programs ITEA and MEDEA another set of projects is running, but it is difficult to

get an overview over of those these projects.

Education

In the future the Embedded Systems community needs people with a variety of skills on different levels.

For 30 years Institutes of Higher Education have delivered engineers on Bachelors level with knowledge of more than one discipline. A lot of the multidisciplinary architects today have this background. They are the pragmatic generalists who have found their way in the embedded systems jungle.

Unfortunately mathematics is taken out of the curriculum nowadays, while math is the Esperanto of the technicians. There is no doubt that in the future this will have its impact on the employability of these students in the embedded systems world.

Nowadays Institutes of Higher Education also deliver engineers with a Masters degree. These engineers are meant for the above mentioned excellent bachelors students. These Masters are more generalist or more specialist. An example is the Hogeschool of Arnhem en Nijmegen which delivers a masters study in Control Engineering. These Masters studies are also a good way to keep knowledge up-to-date of experienced engineers. They are also available in part time versions.

Dutch universities deliver Bachelors and Master with a more scientific approach: less generalist, more specialist. Universities are working on more multidisciplinary masters. The Universities of Eindhoven and Enschede are going to deliver Masters of Science for Embedded Systems students.

On the Stan Ackermans Institute of the University of Eindhoven you can get your MTD degree in Technical Informatics. I hope these people become the next generation of System Architects. An extra level of abstraction is needed in the complex systems of tomorrow and MTD's are able to cope with this. It is a pity that last year the Stan Ackermans Institute was broken up and organizationally placed under the ivory towers of the old faculties. This means that only mono-disciplinary MTD studies are possible from now on. A missed opportunity!

What we are missing is "post traumatic education". First you build a product in the real world and you have a feeling that you can do better. Therefore a fast track is needed for "born" embedded systems architect. Now it takes too long before they have enough experience to design large large-scale projects. Maybe a new future for OOTI?

Epilogue

We are building Embedded Systems for no more than 30 years now. And look where we are today. Now try to imagine what YOU can do in the next 30 years. The only restriction is your own creativity!

References

[1] Wim Smit, Wim Hendriksen (2003). *Embedded Systems, Smart and Intelligent Tools in an Increasingly interconnected globalized world* (ISBN 90-806440-2-1)

[2] Robert M. Pirsig (1974) *Zen and the Art of Motorcycle Maintenance* (ISBN 0-688-05230-4)

[3] Rotel Owner's manual RDV-1060 DVD audio/Video player 2003.

[4] Gerrit Muller: Gaudi site
http://www.extra.research.philips.com/natlab/sysarch/

[5] Embedded Systems Roadmap
http://www.stw.nl/progress/ESroadmap/index.html

[6] PROGRESS
http://www.stw.nl/progress

[7] ESI
http://www.embeddedsystems.nl

[8] Route67
http://www.route67.nl

About the author

Wim Hendriksen is part time Lector in the area of Embedded Systems at the Hogeschool van Arnhem en Nijmegen besides being an independent consultant in his own company Route67 [8]. He is involved in the applied research of multi- disciplinary requirements management and the application of embedded systems in the homes for senior citizens. Before 2000 he was manager software development at ASML and design engineer at ICT. He received his masters' degree in electronics at the University of Twente in 1979. He is a member of the advisory committee of OOTI and the program committee of PROGRESS. Email: wim.hendriksen@route67.nl or wim.hendriksen@han.nl

m

Recent OOTI Publications

The post-masters programme OOTI is concluded with a design project. The final reports of these projects are in general publicly available
unless stated otherwise. The following reports have been published lately.

Tim Albu
a Visual Programming Language
ISBN: 90-444-0260-9

Alina Albu
Design Criteria for FPGA Applications
ISBN: 90-444- 0323-0

Michiel Tas
Two Case Studies in Software Analysis and Design
ISBN: 90-444- 0314-1

Yegor Bondarau and Lucian Voinea
IBO+ Dual Streaming
ISBN: 90-444-0315-X

Chris Delnooz and Laurens Vrijnsen,
The Caribou Project: a Scenario-based Approach towards a Prototyping Framework
ISBN: 90-444-0316-8

Dmitri Jarnikov
Towards Balancing Network and Terminal Resources to Improve Video Quality
ISBN: 90-444-0317-6

Jana Kapustova and Alena Kryvinchanka
Metadata for Everyone
ISBN: 90-444-0318-4

Xiaoyo Liu
Integration of Analogue TV Services into MHP
ISBN: 90-444-0327-3

Yarema Mazuryk and Cristian Pau,
Frame Accurate Editing of DV Material on a HDD/DVD+RW Recorder
ISBN: 90-444-0318-4

Andrey Mikheev,
Rendering of 3D Geometrical Objects Embedded into Volume Images
ISBN: 90-444-0320-6

Michiel van Osch,
Client Side Caching of Dynamic Web Pages
ISBN: 90-444-0321-4

Elena Shumskaya,
User Friendly Pulse Programming Environment for Philips MR Scanners
ISBN: 90-444-0322-2