Generating Quality Software Specifications For Decision Support: A Novel Approach

by

Michael P. Voightmann

B.S., Mathematics and Computer Science (2001)

University of Illinois: Urbana-Champaign

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aeronautics and Astronautics

at the

Massachusetts Institute of Technology

June 2004

Signature of Author .....
Department of Aeronautics and Astronautics
May 14, 2004

Certified by .....
Charles P. Coleman
Boeing Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Generating Quality Software Specifications For Decision Support: A Novel Approach

by

Michael P. Voightmann

Submitted to the Department of Aeronautics and Astronautics
on May 14th, 2004 in partial fulfillment of the
Requirements for the Degree of Master of Science in
Aeronautics and Astronautics

Abstract

An approach utilizing cognitive support tools is presented with the purpose of improving upon the generation of quality software requirement specifications. Based on successful tools in product development and cognitive engineering, the framework suggests how to improve upon user comprehension by the creation of decision support tools. These tools can facilitate the construction of a well-structured system map, incorporating key cognitive aspects, as well as identifying hidden requirements. Minimizing the requirements errors early in the process can have a tremendous impact on the success of the project. Current practices have been unable to improve upon the large failure rate of software systems. They generally lack the structure that cognitive support tools can provide. In order to assess the proposed approach, an overview of the current state of software requirements practice is provided. Key product development and cognitive support approaches were then analyzed. The proposed approach is described, followed by suggestions for future tools that could have a great impact on requirements engineering. Finally, the proposed approach is evaluated by applying it to an automotive application: adaptive cruise control.

Thesis Supervisor: Professor Charles P. Coleman
Title: Boeing Assistant Professor of Aeronautics and Astronautics

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE: INTRODUCTION

Software systems continue to miss deadlines and cost many times the amount budgeted. On top of that, 80% of the total life cycle expenditure is spent on maintenance work, where 50% of the total is spent on corrective maintenance [19]. Many of software problems can be directly correlated to an incomplete set of requirements (inadequate requirements process). It is well known that projects with quality requirements save greatly in the maintenance stage of the life cycle.

The difficulty in software development is in the design. How can a software team gather and verify a set of requirements that correctly captures the utility of the system? There is a need to deliver the product that the customer wants. Currently, many software projects are developed using little or poor requirements processes. Those who do use requirements engineering processes have had success at smaller scales but for large complex applications, they have not succeeded in meeting budgets, schedules, or acceptable success rates.

Customer centric approaches seem to work well for hardware products: value innovation, quality function deployment (QFD), and the metrics thermostat are key examples. These techniques use customer information to inform the design of high utility products. But more is required for the complex software systems that are currently being built.

How can these hardware approaches be mimicked for software? In addition to improving utility, will an effective process for software also improve architecture, efficiency of software development, flexibility in changes, and maintenance? Where could one start to develop such an approach in software?

Is a new approach enough? There are plenty that claim successes in some instances. New process models, programming paradigms, high-level languages. It is important to

have a comprehensive toolset and to be knowledgeable of when to use the appropriate tool. Some projects succeed at utilizing object-oriented programming while others fail horribly. What is the secret to success? Complex software must be developed from a systems engineering standpoint. It does little good to have the best tools available if the developers have motivations contrary to the company's goals. Can the constraints and incentives necessary to develop a vibrant learning culture be built in to such tools? Cognitive engineering can assist here.

In order to see improvements in the design of software systems, it is necessary to be aware of the environmental constraints, including the cognitive loads placed on the users. The proposed approach is to extend the above customer centric approaches with cognitive support tools that will improve the requirements elicitation process, the design of the system, and decrease the need for future maintenance.

As an initial test of this concept, the proposed approach will analyze a functionality that is beginning to be incorporated into automobiles, adaptive cruise control. Automobile software is a great example of the complex sociotechnical environment that this approach can improve upon. New tools to simplify automotive software development could also have a tremendous impact.

CHAPTER TWO: MOTIVATION

The purpose of this chapter is to describe the major difficulties of developing software. It will be shown that many of these difficulties largely reside in the generation and management of the requirements specification, and not in the actual development of the code. In order to accomplish this goal, the common terminology involving the requirement specification will be reviewed. The location in the life cycle of many of the software development community's problems will be reviewed, as well as the many reasons for these difficulties. The general approach to requirements development currently used will then be reviewed. Reasons will be given as to why there is a need for improvements in current practice. Finally, this need will be expanded upon and it will be asserted that there is need to utilize an approach that builds and reinforces the right culture.

2.1 Definitions

In order to begin on level footing, it is important to begin with common definitions to facilitate communication. This section will provide a brief description of what a software specification entails, and more particularly, give definitions of the many types of requirements.

Put simply, "a software requirements specification precisely states the functions and capabilities that a software system must provide and the constraints that it must respect." [67] It should describe what the system should do in various situations, but it should not describe how these actions are done. The specification should contain the system model, details on how the system should evolve, the system goals, the constraints, the priorities, the interfaces to the environment, and other particulars (glossary, data dictionary). It is

important to remember that all specifications are abstractions. They leave out unimportant details, and what is important depends solely on the problem being solved. [40]

A software specification for an automobile would include all the requirements necessary for the software to perform nominally, including safety requirements, performance goals, and functional requirements (such as the functionality for the intelligent power train control).

This document should be precise so that it can be utilized by a large and diverse group of people including the customer, the marketing and sales staff, the project managers, the software developers, the testing team, the training staff, the legal staff, and the subcontractors. It is quite a challenge to write a document to satisfy all of these interests and levels of understanding.

There are basically three categories of specifications: informal, formatted, and formal. An informal specification is written in natural language and is generally easy to read/understand. However, these documents lack organizational structure, and the ambiguity can lead to inconsistencies and misunderstandings. A formatted specification (such as UML) has a standardized syntax, and can be checked on basic consistency and completeness tests, but since it has imprecise semantics, other sources of errors may be present. Finally, a formal specification has precise semantics and syntax. It is mathematically rigorous so one is able to verify equivalence between the specification and the implementation. Imprecision and ambiguity are eliminated, but at the cost of a document that can be very difficult to read without training. For more information, please refer to [40][67].

So what is a requirement? A definition given by Lethbridge states that a requirement is a statement about what the proposed system will do that all stakeholders agree must be made true in order for the customer's problem to be adequately solved. [36] A requirement details how the software should act for a particular aspect of the system. An

example of a requirement: "The editor shall require the user to confirm global text changes, deletions, and insertions that could result in data loss." [67]

There are many types of requirements that must be taken into account. These requirements are attributed at times to certain groups of people and may describe varying levels of abstraction. The following terms are commonly encountered in the requirements engineering domain. Many difficulties in communicating project goals can be alleviated by a common understanding of the many types of requirements.

Some of the more common categories of requirements most often mentioned include: system, business, user (at times referred to as customer), functional (behavioral), and nonfunctional.

The term system requirements describes the top-level requirements for a product that contains multiple subsystems-that is, a system. A system must take into account not only the software involved, but also the hardware it is based on and interacting with, as well as the people interacting with it. A system requirement for an automobile may be that the airbag deploys at the appropriate time. Business requirements describe the high-level objectives of the company who requests the system. They describe the reasons why the organization is implementing the system; they are the objectives the organization hopes to achieve. A business requirement for an automobile could be a budget (or a schedule), to ensure a healthy profit margin. User requirements describe user goals or tasks that they must be able to perform with the product. A user requirement could be an "easy to use" cruise control. Functional requirements specify the behavior of the software that the developers must build into the product to enable users to accomplish their tasks. "The system shall e-mail a reservation confirmation to the user." Functional requirements describe what the developer needs to implement. Finally, the nonfunctional requirements include performance goals and descriptions of quality attributes. These characteristics include safety, usability, portability, integrity, efficiency, maintainability, and robustness. The airbag must deploy in a given time frame for example.

Wiegers provides a checklist of characteristics for producing excellent requirements. They must be: complete, correct, feasible, necessary, prioritized, unambiguous, and verifiable. For a more thorough treatment of software specifications, refer to [67].

2.2 Software Difficulties

Now that the common terminology has been provided, this section will look at some of the difficulties in software development. There will be a brief overview of the current state of software development. This includes a number of statistics on software related costs. Then the actual difficulties in creating software will be reviewed and it will be determined at what point in the development process most needs to be improved upon.

A study conducted in 1995 by the Standish group estimated that 31.1% of projects would be cancelled before they ever get completed. Furthermore, 52.7% of projects would cost 189% of their original estimates. This is not including the lost opportunity costs (that are not measurable), which could easily be trillions of dollars. They estimated that in 1995, American companies and government agencies would spend $81 billion for cancelled software projects. These same organizations would pay an additional $59 billion for software projects that will be completed, but that exceed their original time estimates. [60]

There is not much evidence to show that this has improved dramatically. Caper Jones tells a similar story (1997): that 1/3 to 2/3 of software projects will exceed their schedule and budget, and 1/2 will be canceled for being out of control. [28] And more recently (2002):

"Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic

product, according to a newly released study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST)" [50]

And this is only the difficulty in completing the project. The actual development costs are a small part of the total life cycle cost. As mentioned in the introduction, the maintenance phase can cost up to 80% of the total, and corrective maintenance 50%. [19]

What are the key drivers behind the difficulties listed above? What is causing these software projects to miss deadlines, to never be completed? In 1986, Frederick P. Brooks, Jr laid forth the key difficulties of developing software in his well-known paper, "No Silver Bullet: Essence and Accidents of Software Engineering."

"I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation." [6]

He then listed a number of essential difficulties regarding the development of software systems. These include: the inherent complexity of software, the need for software to conform, the ease to which software can be changed, and the difficulty of visualizing software. Brooks argues that despite the claims of some, there is no silver bullet, that there is no magical solution to the "software problem." The following analogy bears this out:

The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today. [6]

Let's look at these difficulties in more detail:

Software entities are more complex for their size than perhaps any other human construct because no two parts are alike. In this respect, software systems differ profoundly from the mechanical systems generally built: computers, buildings, or automobiles, where elements can be reused much more easily. The high level of coupling also causes problems. Software systems can easily go beyond the user's level of comprehension through the level of interaction there exists between the components. This can lead to seemingly chaotic behavior, where a small change can lead to a small or large effect with little to information to guide the engineer on which will occur. Other reasons include the incredibly large discrete state spaces, the inability to effortlessly scale the software system up, and the need to extensively test these systems. Finally, software is dynamic. It can be extremely difficult to track the flow of control and data through the system.

"Software is error-ridden in part because of its growing complexity. The size of software products is no longer measured in thousands of lines of code, but in millions. Software developers already spend approximately 80 percent of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors. " [50]

Much of the complexity in software can be attributed to the necessity to conform to the arbitrary rules put forth by organizations, at times without rhyme or reason. While mechanical systems must conform to the laws of physics, software must only conform to the human developers. This freedom can cause problems, since it gives software the appearance that it is much easier to have the software conform than other components. Software is performing tasks for which it wasn't originally designed. That adds to the software failure rate.

Since software appears to be easy to change, it is generally the first to be changed. Why redesign and reprint a circuit board if problems can be resolved by simply changing the software? It is the lack of constraints that gives software this appearance. But software is not too easy to change. It can be extremely difficult to integrate software subsystems, one prime reason being the level of coupling mentioned above.

The apparent ease of change gives rise to a common problem, feature creep. Since software seems to be easy to change, the temptation to redesign and rewrite it at will is high, particularly as a deadline approaches. This attitude can render the system highly unstable (since software is difficult to test in the first place). Software is so flexible that one can start working with it before fully understanding what is required to be done. Combined with the appearance of stability, it is a difficult attribute to overcome.

Software is invisible and unvisualizable, there is no ready geometric representation in the way that land has maps, silicon chips have diagrams, and computers have connectivity schematics. [6] Software is composed of abstract interlocking concepts: data sets, and algorithms for example. There have been advances in the comprehension and visualization of software, but there is still much to do in order to improve upon the engineer's ability to document these systems and to assist those maintaining the system to come quickly up to speed.

This difficulty can be seen by the statistics from the previous section, there is still a large challenge to locating software errors. Adding to this, a study by the IEEE stated that peer reviews of software will catch 60 percent of all coding defects. But few, if any, companies subject their programming to peer review.

These difficulties can be alleviated by improvements in the requirements process. When deficiencies are present in the requirements generation, errors will propagate to later stages of the lifecycle. This will lead to unnecessary "rework." Rework can consume 30 to 50 percent of your total development cost (Boehm and Papaccio 1988), and requirements errors account for 70 to 85 percent of the rework cost (Leffingwell 1997). [67]

As illustrated in the figure below, it costs far more to correct a defect that's found late in the project than to fix it shortly after its creation. Preventing requirements errors and catching them early therefore has a huge leveraging effect on reducing rework.

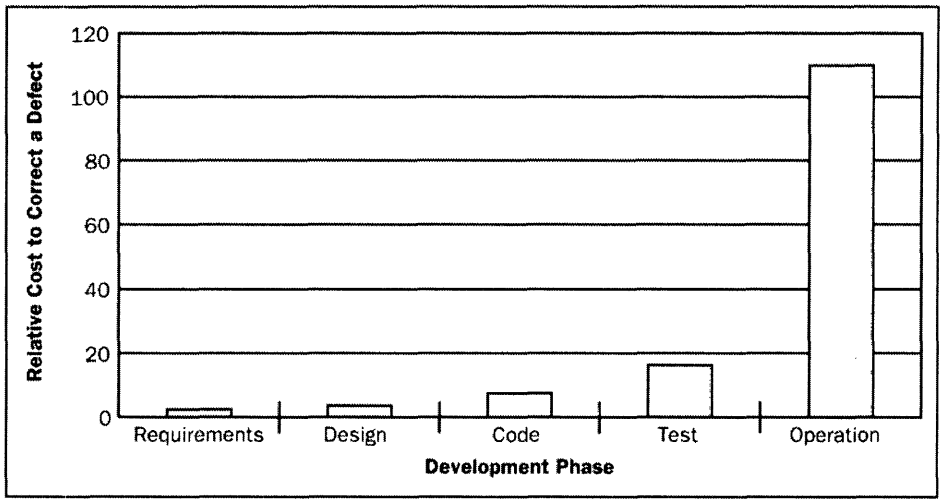


Figure 2.1: Relative cost to correct a requirement defect depending on when it is discovered. [67]

It costs 5 to 10 times more fix a defect during or after implementation versus correcting it in the planning stage. "If you didn't have 5 hours to fix it the first time, where are you going to find 50 hours to fix it later."

The evidence is overwhelming and long known that the cost to remove defects in requirements increases geometrically with time. Once your software hits the field, removing a requirements defect costs at least a hundred times as much, assuming you can fix it at all. [67]

Much of these cost overruns can be attributed to a poor requirements process, or poor software engineering in general. Too much emphasis may be placed on one aspect of the process (functional requirements for example). But the individuals designing and building these systems are not novices. Why are they having so much difficulty? This question will be addressed in section 2.5 on the surrounding culture.

2.3 Current Practice

This section will review the current practice in requirements development. For a more thorough review, please refer to [67].

Requirements development is generally subdivided into four parts: elicitation, analysis, specification, and validation. These sub-disciplines incorporate all the tasks involved with the gathering, evaluating, and documenting of the requirements for a software product.

In the elicitation stage of the requirement development process, the goal is to identify as many relevant requirements as possible. This is done in a number of ways. An important first step is to write down the system's business requirements. The objectives of the system will provide an important reference when writing down requirements to come. Next, those who will be using the system must be identified as well as a list their characteristics. It can be quite expensive to add a missed customer group after release. The characteristics are also vital since their needs may vary greatly from group to group. Once the users have been identified, one must gather requirements from them in a few ways. This can be done by surveys, workshops, focus groups, direct communication, and observation. It is valuable to step through possible scenarios and system events (and the response) with the user. The system's success is dependant upon the ability to capture their needs accurately and create what the customer wants. One must not forget the equally important nonfunctional requirements. The acceptable levels of quality, performance, and safety (among others) should be documented. Finally, it is vital to learn from the past. Evaluate the requirements documents from current and past systems to understand what has and hasn't worked well. It is also useful to reuse previous requirements if similar functionality will be in the new project (the same government regulations for example.

In the analysis stage, the requirements begin to refine, and they must be checked for omissions and errors. It is important to ensure that all those involved understand the

requirements. One must also check the feasibility of the system, as well as begin to make engineering decisions to resolve conflicts. The goal is to create a set of requirements that is stable enough to move ahead with scheduling and the design of the system. Wiegers advocates viewing the requirements from multiple views. This can help to reveal new requirements as well as assist the users come to a common understanding. Key approaches to analyzing the requirements include creating a context diagram, creating user interfaces and technical prototypes, prioritizing the requirements, modeling the requirements graphically, creating a data dictionary, split the requirements into appropriate subsystems, and perform Quality Function Deployment (QFD is spoken more of in section 3.1.1)

The requirements document is constructed in the specification stage. In practice, the business requirements are stored in the high-level vision document. User requirements have been written in a number of ways, a common approach is in the form of use cases or scenarios. The software requirements specification generally contains the detailed functional and non-functional requirements. Wiegers provides a standard template for documenting the requirements [67] In chapter 3, another approach will be seen advocated by Leveson [40], the intent specification. Workflow diagrams and data flow diagrams are also commonly used. Important to the documentation process is to have a solid structure underlying the document that it is easy to use and understand. It is a good idea to keep track of which users correspond to particular requirements, so that they can be easily identified at a later time.

Validation is the final check for correctness. All participants must inspect the documents carefully to ensure they meet their needs. Informal and formal inspections of the document must be performed to check for correctness. Test cases should be developed from the user requirements to determine whether the system performs properly in different situations. They should also be used to verify the correctness of the analysis models and prototypes. Finally, the acceptance criteria should be defined. The development team can ask users to describe how they will determine whether the product

meets their needs and is fit for use. Acceptance tests can then be based on the usage scenarios.

Now that the general requirements process has been reviewed, a few caveats must be mentioned. The requirements development process is a non-linear iterative process. These four subdivisions can be performed sequentially, but they are generally interleaved. There is no formulaic approach to requirements development.

2.4 The Need for Robust Requirement Processes

The problems that incomplete requirements can cause are now well known. The standard practices for developing requirements have also been reviewed. But the question remains, how much must one spend on requirements? Of course the amount depends on the project. For the construction of a web application that may have the lifespan of six months, one wouldn't want to spend an enormous amount of time on the requirements specification. But as was pointed out in section 2.2, even a modest improvement to the requirements specification could have a huge impact on the lifespan of the software. While the importance of a quality software specification has been increasingly spoken of in recent years, there are still questions as to how much is enough?

Some approaches advocate minimizing the upfront design time, since the requirements can change at a quick pace. Chris Peters, a very successful manager at Microsoft, states that the specification should not be so detailed that it constrains innovation.

"The spec absolutely changes during development. A spec is a living document. It's the current best description of the features in progress. It gives plenty of latitude to the developers to invent. Hopefully, it's thought through the hard issues, like how this feature interacts with other features... You gain so much information on your customers. You gain information on your competitors. You gain information on how easy

something is to implement. Or you actually implement one feature and then you suddenly realize that, with a small extension, it could be used in this whole other area of the product that you didn't realize when you first started. You now see it, it's obvious. If you somehow said, "Okay, here is the spec, and then we're going to lock coders – which is a very insulting term for a developer – into this room and they're going to write a spec," then you'd have terrible morale, the thing would be junk, and it wouldn't come out on time. [10]

The specification should not constrain creativity. But it is vital to be aware of the constraints users must work within; the environmental and cognitive constraints for example. As Chapter 3 will show, there are methods that can help to extract this information before the development stage. The Cognitive Work Analysis helps to identify key constraints that the company can't afford to have discovered late in the process. Quality Function Deployment is a good example of a method that helps to gather important customer requirements early.

It is not necessary to specify every detail of the project ahead of time. In fact, a healthy work environment will allow the workers to use their abilities to finish the design. But it is vital to have a "complete" specification that informs the workers about the constraints. The users must be knowledgeable of (and be able to document) the constraints and assumptions made at the many project levels. Any object can be simple or complex depending on the representation. These objects must be represented from many levels of detail to aid comprehension. Decision support tools must be provided to take this information into account.

At times, an evolutionary approach may be more beneficial. But as systems continue to become more complex, engineers will be unable to make knowledgeable adjustments. There is a need to continue to improve upon the current development tools because the process is still too error-prone today. The designer needs support tools to be able to take into account not only the system architecture but also aspects such as the performance

and maintainability of the system, data structures, and memory management. There is also a need for a supportive learning culture, which is addressed in the next section.

2.5 The Surrounding Culture

The surrounding culture is vital to the requirements process. What does the surrounding culture entail? It is the work environment: the environmental constraints as well as the business, political, and social constraints (how supportive the environment is to learning, the abilities of the workers). A healthy work environment is vital since it can make or break the project. If the development team lacks certain attributes or motivation, the project is at high risk to fail, regardless of the technology used. On one extreme, personal rivalries between team members can cause a project to fail. Another example, the documentation of core components could be added at the last minute just to "satisfy" upper level management. At the other extreme, there are quite a few people who choose to write high quality code in their free time and give it to others for free.

The importance of the culture can be seen with examples from the tools in use. There are numerous examples of technology in the software field that some claim to be of high value, while others claim they do not solve the problems, that they actually make the situation worse. One example, object-oriented programming, has become entrenched into common practice. Meyers argues that this is the path to the future [46], while others argue that object-oriented programming doesn't sync with the engineer's mental models. [19] What's the correct answer? Like every other tool, object oriented programming has its place in the toolbox. The tricky part is knowing when to use which tool. And this is where the surrounding culture should be supportive. There have been claims [46] that the reason object-oriented projects have failed is that the developers don't know how to correctly utilize the concept. Proper training before the use of any tool is vital to success. But at the same time, if experts in the field of software engineering have trouble using a concept efficiently, is it their fault or the fault of the concept? Or neither? Perhaps the proper motivation was lacking in some cases. Or the tool may not match the mental

model of the developer using it. Imagine there is a new instrument being used in an aircraft. If the pilot has trouble using this instrument, regardless of whether the pilot or the instrument is too blame, something needs to be fixed. Perhaps the pilot needs more extensive training, or more motivation drivers. Or the instrument needs to be more (HCI) in tune with the user's mental models. There needs to be more cognitive support so that one can use it effectively. The point here is to express the importance of understanding the surrounding culture and how to shape it for the project's and company's utility.

"Trust is a broad concept, and making something trustworthy requires a social infrastructure as well as solid engineering. All systems fail from time to time; the legal and commercial practices within which they're embedded can compensate for the fact that no technology will ever be perfect." [47]

Software engineering differs from the other engineering fields with regards to documenting historical usage information. "The classic engineering paradigm of control process feedback, ("do not make the same mistake twice"), is almost completely absent from software engineering with repetitive failure modes common, (i.e. the same fault failing repeatedly because it cannot be located from the evidence available). This is a direct consequence of the almost complete absence of measurement and analysis coupled with the poor understanding of prediction and diagnosis in software engineering systems."[20]

There is a need to shape the work environment so that engineers can learn from others' successes and mistakes. There must be incentives to share individual's work with others in order to create a vibrant learning culture.

Humans will continue to make errors. Software engineers will make design errors, fail to install a patch, or worse, install a patch that damages the system further. Simple mistakes can cascade into disastrous ones. Users will still be working with imperfect mental models, collaborating with others' different imperfect mental models. There are a tremendous amount of assumptions that are taken for granted. (people have very

different goals) And changes in the project will happen continuously. Engineers have the ability to build tools to support their imperfect mental models, to facilitate communication as well as increase the fidelity of the communication between coworkers. They also have the ability to ensure the proper culture is build to reinforce good practices, provide feedback to eliminate errors, and to provide a safety net when errors do occur. This culture can be developed with the assistance of cognitive support tools. The proposed approach to this problem will be explained in section 4.1. The next chapter gives brief overviews of a few tools that will help with this task.

CHAPTER THREE: Successful Approaches

The problems that must be faced in constructing large software systems have been shown. The need for improved tools for generating software requirements has also been provided. The purpose of this section is to review a few approaches that have performed well in areas outside the software engineering industry. The first section will review three tools that have performed well with hardware products: Qualtify Function Deployment (QFD), Value Innovation (VI) and the Metric Thermometer (MT).
The second section will review three approaches that incorporate cognitive engineering into their design, that may be of assistance in creating complex systems: two cognitive models that have been of assistance in creating complex systems, the Cognitive Work Analysis (CWA), the Resources Model (RM), and the Intent Specification.

For each of these approaches, a short overview will be given including the motivations driving them, as well as what the approach consists of. Particular aspects of these approaches that may be of assistance will then be provided.

3.1 Product Approaches

This section will look at product approaches that are strong in capturing customer input, and driving the culture to satisfy this goal. They excel at translating the customer input into technical requirements, prioritizing them, and representing this process in ways that help the user to better comprehend the system and the underlying assumptions. These aspects are vital to depicting the system accurately.

3.1.1 Quality Function Deployment

Quality Function Deployment (QFD) is a set of powerful product development tools that were developed in Japan to transfer the concepts of quality control from the manufacturing process into the new product development process. Yoji Akao first implemented QFD at the Mitsubishi Heavy Industries Kobe Shipyard in 1972. A few years later, Toyota began having success using QFD, which sparked interest in the approach in the west. Toyota had been able to reduce their product development costs by 61%, a decrease in the development cycle by one third. [61] Other often cited benefits include: reduced time to market, reduction in design changes, decreased design and manufacturing costs, improved quality, and increased customer satisfaction.

Akao defines QFD as "a method for developing a design quality aimed at satisfying the consumer and then translating the consumer's demands into design targets and major quality assurance points to be used throughout the production state" [2] It allows customers to prioritize their requirements, give input as to how the company compares to competitors with regard to these requirements, and then directs the company as to how to optimize those aspects to bring the greatest competitive advantage.

In order to do this, QFD addresses three classes of requirements: expected requirements, normal (or revealed) requirements, and exciting requirements. Expected requirements are those that the customer may not even mention, but would be upset if they were missing. Normal requirements are those gathered by customer request. Exciting requirements are those beyond the customer's expectations and are generally difficult to discover. These features provide high value to the customers but there is little penalty if they are not present. Businesses must meet all three types of requirements – not just what the customer says.

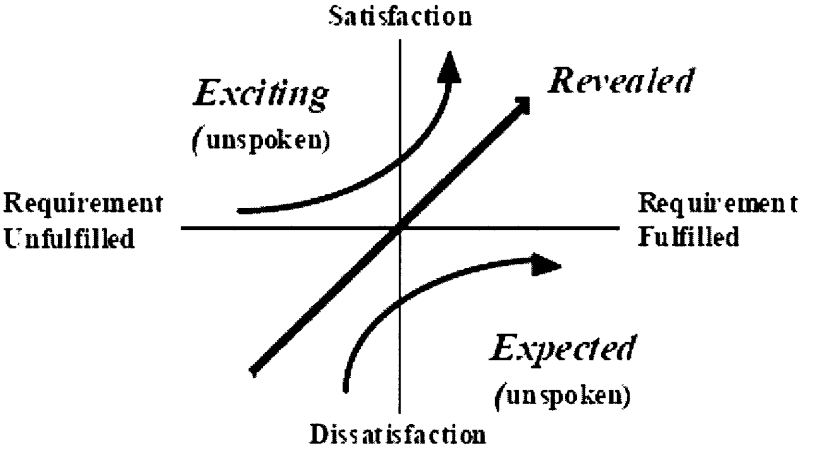


Figure 3.1: The Kano Model (adapted) [45]

Looking at requirements from this perspective helps to keep the focus on the customer's wants and needs. It can be useful in uncovering explicit and implicit requirements.

In order to transform customer requirements into the product they want, QFD uses a series of matrices and tools to document the information as well as process it, in order to create a plan for the product. The QFD methodology is based on a systems engineering approach similar to the requirements process shown earlier. QFD utilizes a few management and planning tools which are used in many of its procedures. Of particular interest are the affinity diagrams, hierarchy trees, and the House of Quality.

An Affinity diagram is a useful tool to organize a set of qualitative information; the input from the customer for example. A hierarchy structure is formed by sorting the customer requirements into groups, as seen below. [43]

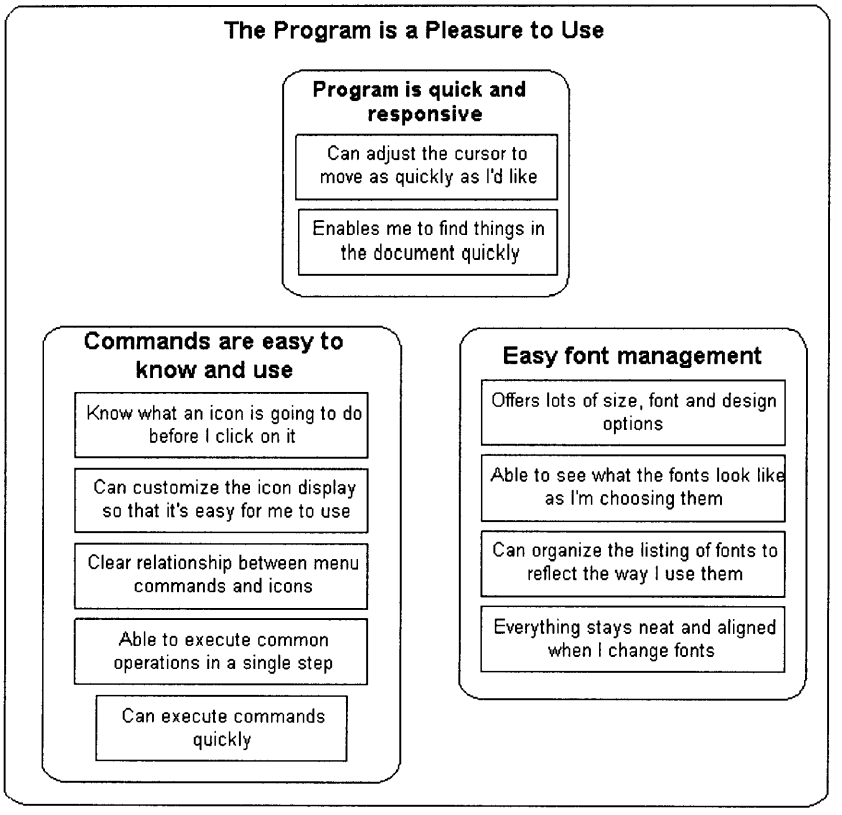


Figure 3.2: Affinity Graph [43]

A Hierarchy tree also displays the relationships between groups of statements, but it is constructed analytically from the top down. It is generally used after the Affinity Diagram to check for flaws in the customer requirements.

The "House of Quality" matrix is utilized by a multidisciplinary team to translate a set of customer requirements, drawing upon market research and benchmarking data, into prioritized technical requirements for the product. The matrix can be adapted in many ways to fit with the goals of a particular project. The general format of the "House of Quality" is made up of six major components, which are completed in the course of a QFD project:

1. Customer requirements (HOWs) - a structured list of requirements derived from customer statements.

2. Technical requirements (WHATs) - a structured set of relevant and measurable product characteristics.
3. Planning matrix - illustrates customer perceptions observed in market surveys. Includes relative importance of customer requirements, company and competitor performance in meeting these requirements.
4. Interrelationship matrix - illustrates the QFD team's perceptions of interrelationships between technical and customer requirements. An appropriate scale is applied, illustrated using symbols or figures. Filling this portion of the matrix involves discussions and consensus building within the team and can be time consuming. Concentrating on key relationships and minimizing the numbers of requirements are useful techniques to reduce the demands on resources.
5. Technical correlation (Roof) matrix - used to identify where technical requirements support or impede each other in the product design. Can highlight innovation opportunities.
6. Technical priorities, benchmarks and targets - used to record the priorities assigned to technical requirements by the matrix, measures of technical performance achieved by competitive products and the degree of difficulty involved in developing each requirement. The final output of the matrix is a set of target values for each technical requirement to be met by the new design, which are linked back to the demands of the customer. [43]

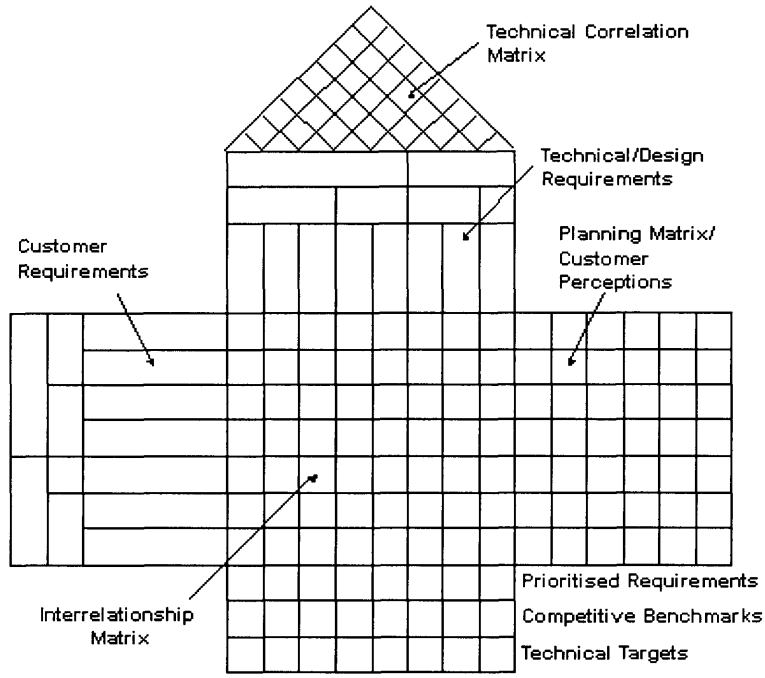
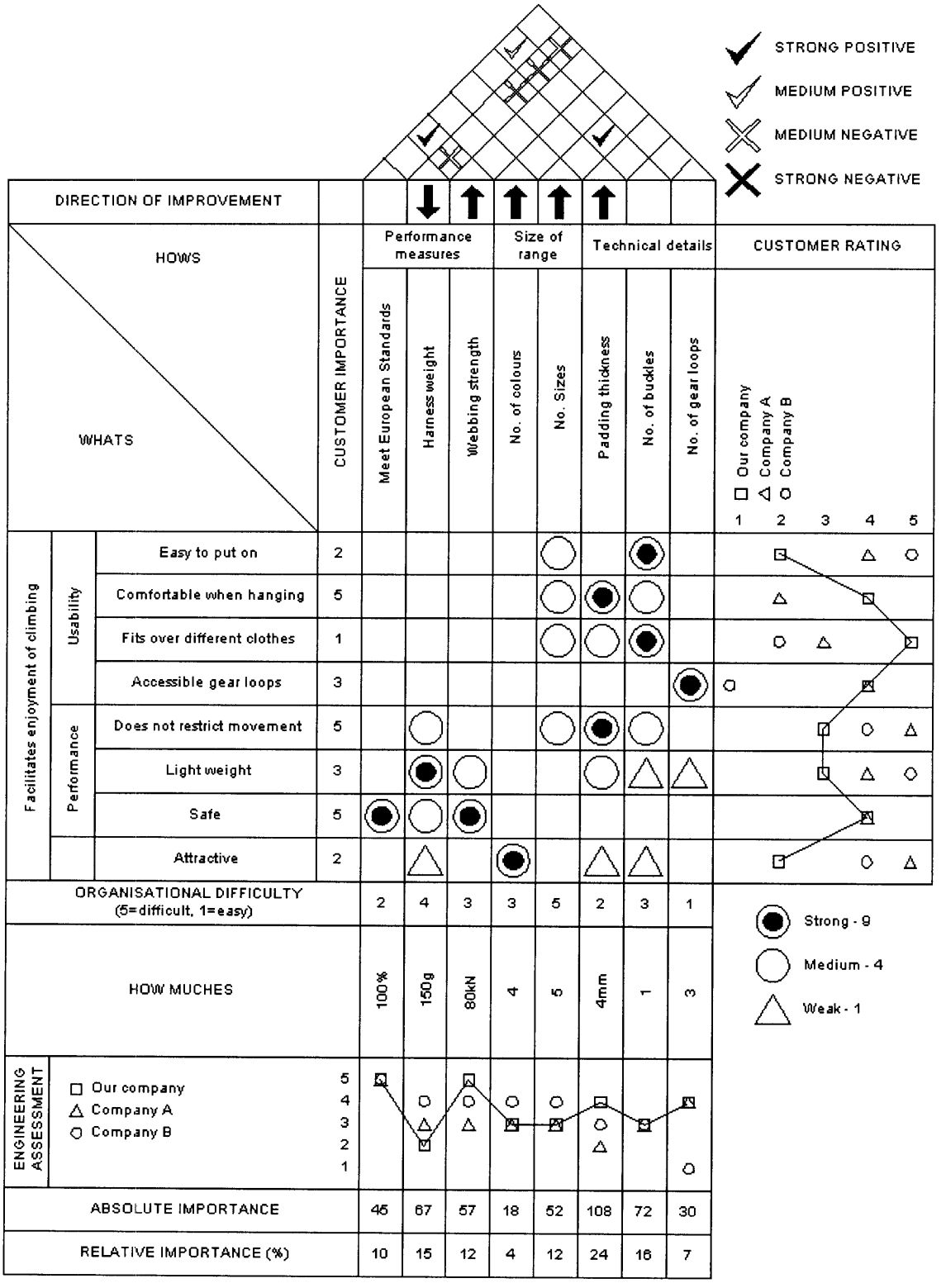


Figure 3.3: Blank House of Quality [43]

Above is a blank House of Quality, and on the following page is a filled in House of Quality, which assisted in determining the requirements for a mountain climbing harness.



DIRECTION OF IMPROVEMENT		↓	↑	↑	↑	↑				
Facilitates enjoyment of climbing	Usability	Easy to put on	2					○		●	
		Comfortable when hanging	5					○	●	○	
		Fits over different clothes	1					○	○	●	
		Accessible gear loops	3								●
	Performance	Does not restrict movement	5		○			○	●	○	
		Light weight	3		●	○			○	△	△
		Safe	5	●	○	●					
		Attractive	2		△		●		△	△	
ORGANISATIONAL DIFFICULTY (5=difficult, 1=easy)				2	4	3	3	5	2	3	1
HOW MUCHES				100%	150g	80kN	4	5	4mm	1	3
ENGINEERING ASSESSMENT	□ Our company, △ Company A, ○ Company B		5 4 3 2 1								
ABSOLUTE IMPORTANCE				46	67	57	18	52	108	72	30
RELATIVE IMPORTANCE (%)				10	15	12	4	12	24	16	7

Figure 3.4: Filled in House of Quality – Climbing Gear [43]

The main features of QFD to take away are its focus on meeting customer needs by directly translating their requests to technical requirements. It facilitates multidisciplinary teamwork through the use of a comprehensive matrix for documenting information, perceptions and decisions. QFD has also shown its flexibility as it has been applied in a number of fields other than product development. One of the main areas is in strategy formulation and implementation. One aspect that needs to be improved upon is the amount of time contributed to completing the QFD process. It can be quite labor-intensive, even though the participants feel it is well worth the cost. Another possible area for improvement is a cultural one. QFD has not had the same level of success in the US as it has had in Japan. Hales lists some of differences between the cultural archetypes and gives suggestions on how to adapt QFD to a US company's environment. [18]

The name QFD expresses its true purpose, which is satisfying customers (Quality) by translating their needs into a design and assuring that all organizational units (Function) work together to systematically break down their activities into finer and finer detail that can be quantified and controlled (Deployment) [45]

3.1.2 Value Innovation

After studying many high growth companies and comparing them to their competitors, W. Chan Kim and Renee Mauborgne, both of INSEAD, noticed a common strategy emerge which they labeled value innovation. [32] They found that the difference between the companies was situated in each group's assumptions about strategy. Managers of the less successful companies followed conventional strategic logic. Rather than promote a strategy driven by competition, the successful companies concentrated on providing the customer value.

The difference between value innovation and conventional business logic can be seen by comparing them along the five dimensions of strategy: industry assumptions, strategic focus, customers, assets and capabilities, and product and service offerings. Conventional logic perceives that industry conditions are given. That one must build competitive advantages to beat the competition. A company should retain and expand its customer base through segmentation and customization by focusing on differences in what customers value. A company should leverage new programs against existing assets and capabilities. Finally, the industries traditional boundaries determine the product and services a company offers. The goal is to maximize those offers. Value innovators feel that industry conditions can be shaped. For strategic focus, the competition is not the benchmark; companies should seek value to dominate the market. A value innovator targets the mass of buyers and willingly lets some customers go. It focuses on the commonalities in what customer value. A company must not be constrained by what it already has. What would be done if the company were starting anew? A value innovator thinks in terms of the total solution customers seek, even if that takes the company beyond its industry's traditional offerings.

"Mauborgne: The essence of business strategy can be traced to military strategy. In terrain and war there's only so much land that exists. Fundamentally that explains why business strategy - including competitive strategy - has been predominantly based on how you divide up an existing pie. It's about relative power. It's a zero sum game because you cannot multiply the size of land available. But, in the realm of business, the new market spaces that can be created are infinite. Historically, real gains came when people created an entirely new area - a whole new market space. You can create a win:win game. You can create new land. Scientifically we know the same amount of chemical compounds that exist has not changed over time. But look at what you had in the beginning - just dinosaurs. And today by creatively combining them in numerous new ways we have - Starbucks. What we can buy today in a 7-Eleven store beats what a king like Louis XIV had. The possibilities are endless." [33]

The Value Curve is a tool that may assist in identifying possible areas to target for improvement. This curve does a great job of quickly displaying the strengths and

shortcomings of a particular product. Once one has rated similar products in the industry along a number of competitive factors (qualities of the product, service, & distribution), the Value Curve can show where possible innovations may lie. This is in great contrast to the traditional incremental innovation against the industry standard. Using the Value Curve, managers can filter out the factors that the industry takes for granted and should be eliminated. They can see which factors should be reduced well below the current standard, or raised well above the current standard. And they can identify the factors that should be created of which the industry has never offered.

The value curve describes a simple way to 'map' the domain space, based on the preferences of the customer, so one can see which particular aspects may be ripe for innovation.

On the following page is an example of such a curve, regarding low budget hotels.

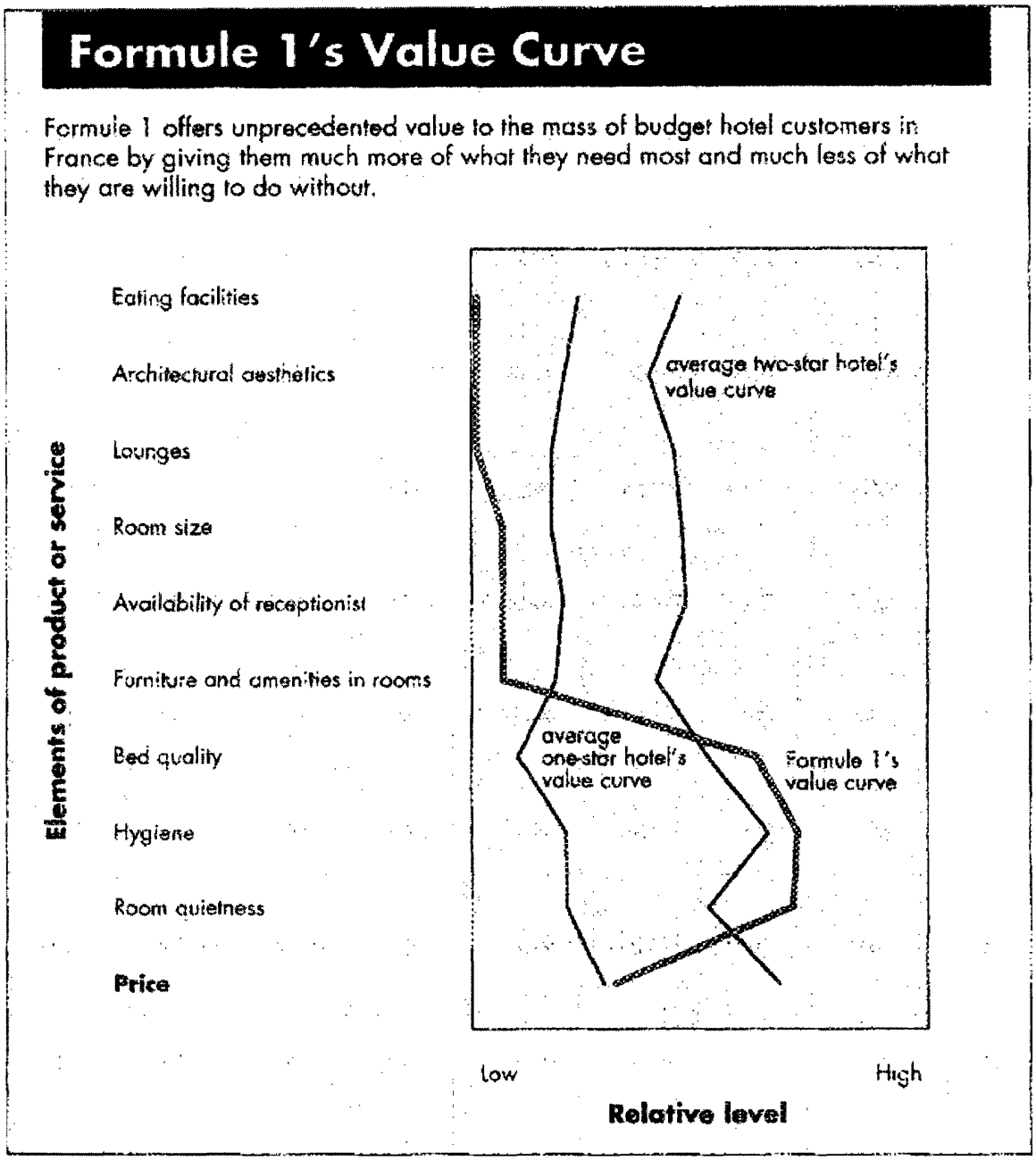


Figure 3.5: Value Curve for low budget hotels. [32]

A key example of the success of value innovation is Southwest Airlines. The airline industry is well known for being extremely competitive. If one carrier cuts prices, the others quickly follow. When another allows cell phone use on the taxiway, the others do likewise. This competitiveness makes it difficult for any company to make a consistent profit. But somehow, Southwest Airlines has been consistently profitable for more than 25 years, an amazing record for an airline. And their secret is well known, they created a

new market: short-haul air transport, with an average distance of 425 miles. They did away with many of the features that were assumed to be key to success: meals, in flight entertainment, multiple seating classes, and the hub-and-spoke system. They are profitable despite the fact that their fares are often 60 percent below their competitors'.

Instead of focusing on their competitors, southwest looked to the automobile, to focus on the factors that lead people to choose driving over flying (They could have just as easily chosen the bus industry, but did not since only a minority of Americans utilize this form of transportation). They then moved to eliminate or reduce the non-essential items normally included with air travel. Do people fly because they want multiple seating-classes, meals, or in-flight films? The answer is, quite simply, no. There is basically only one factor that makes people choose flying over driving: speed. People fly to save time. [31]

There are a number of key points to extract from value innovation. First, prioritized customer input can give great insight into how to best provide value, rather than the common competitive incremental approach to innovation. It is of most value to focus on how to best satisfy the customer and avoid popular functionality and assumption in order to remain "competitive." The value curve provides a simple model of how the customer sees the domain space, and one can quickly see where current functionality is lacking, as well as, where assumptions can be identified as incorrect (as was seen in the Southwest example). Finally, value innovation displays the importance of looking beyond the company's industry to other industries for products and services that perform similar functionality. By concentrating on the strengths of these substitutes and eliminating or reducing everything else, new innovations can be created.

3.1.3 Metric Thermostat

For every product development process, it is vital to have the ability to select the right metrics, as well as establish a culture that rewards properly for those metrics. But how does one know which metrics to concentrate on, and how much emphasis to place on each? The metrics thermostat approach advocates fine-tuning a company's relative emphasis on the metrics they have chosen. The thermostat analogy refers to the adaptive control feedback mechanism that is applied to incrementally improve upon the project's priorities (increased profits for example).

There are hundreds of detailed actions from which the development team must choose. At times, the team members may act in their own best interests to the detriment of the company. The management does not need to keep track of or dictate these actions to the development team. They can control the process by the establishment of implicit weights on their metrics. The thermostat functions by adjusting these weights to optimize the project's objectives.

The general approach is to first identify a set of projects that follow a similar culture. The metrics used to run the company are then gathered. These metrics should determine the implicit rewards. Statistical methods (multiple regression) are then performed on the company's data to obtain estimates of the weights for each metric. User surveys are also incorporated. Iterations of this process help to maximize profits as well as adapt to changes in the environment.

This approach was applied to a large office-equipment firm with $20 billion in revenue.. Three top-level strategic metrics were considered: customer satisfaction, time-to-market, and platform reuse. The customer satisfaction metric is composed of information from a few sources. Teams are strongly encouraged to use voice-of-the-customer methods, consider service during the design, identify the key areas to attack in the target market,

use rigorous processes such as the house of quality, and take other actions to achieve customer satisfaction. With the metrics thermostat, the research team found that the firm had about the right emphasis on time-to-market, but had overshot on platform reuse and had lost its focus on customer satisfaction. The firm agreed with this analysis and acknowledged that the metrics thermostat had been able to quantify their intuition.

The metrics thermostat offers a number of benefits. The idea underlying the metrics thermostat could help tremendously in developing a culture to develop quality software products. This approach provides an example of the ability to tune the culture to produce particular necessary requirements. Acknowledging that members of the team have goals and ambitions that may not be compatible with the firm's goals, implicit rewards can still shape the culture in the desired direction. We seek metrics that will induce the user to do the right thing at the right time.

Also, even with the product team perform a large number of actions, it is possible to motivate properly with only adjusting the relative emphasis on small number of weights. Finally, as the firm in the case study stated, this approach does allow to some extent to quantify the previously unquantifiable.

3.2 Cognitive Engineering Approaches

The last section analyzed three examples that have been successful for hardware products, but have not been able to have a tremendous impact on software products. A different approach is needed for complex systems. Many approaches function under the assumption that the aspects other than technical functionality are irrelevant, but this is not the case in complex systems. Cognitive Engineering is concerned with the analysis, design, and evaluation of such systems. [64] Besides the technical requirements, the environmental context, organizational structures, and the workers play a huge role in the success of the product. This section presents three approaches involving cognitive engineering that may be useful to improve upon the software specification.

3.2.1 Cognitive Work Analysis

Complex systems must take into account not only the technical aspects, but the social and organizational as well. The Cognitive Work Analysis framework has had success with systems that fit this description [64].

Rather than dictate the behaviors necessary for the system to be successful, the Cognitive Work Analysis framework identifies the constraints and allows the worker to finish the design. Its concern is in identifying the constraints that shape behavior. This allows the framework to add the capability for designing for the unanticipated. It can be likened to comparing a set of directions to a map. They both will assist you in reaching your destination, but a map is much more useful if you are lost or need to take a detour. By identifying and documenting the environmental constraints in an intuitive manner, the Cognitive Work Analysis assists in constructing this map.

There are five phases in the CWA framework: Work Domain Analysis, Control Task Analysis, Strategies Analysis, Social Organization and Cooperation Analysis, and Worker Competencies Analysis. The Work Domain Analysis documents the functional structure of the system in which behaviors take place. This phase creates a "map" that details the actions that are possible. The Control Task Analysis identifies the tasks that must be accomplished, independent of how they are completed or by whom. These are the actions performed based on the information received from the work domain. The Strategies Analysis identifies the different approaches possible to complete each of the tasks, independent of the workers involved. The Social Organization and Cooperation Analysis helps to create the proper organizational structure for the project. This includes decisions on when, where, and how much automation to use. Finally, the Worker Competencies Analysis determines what is required of the users to function efficiently in the system. This is based on Rasmussen's skills, rules, knowledge (SRK) taxonomy. [55] These are the constraints on the workers, from the general human limitations to the

particular competencies needed for these tasks. The order is important as each phase inherits the constraints from the previous.

Modeling tools are necessary in order to perform these analyses. The abstraction-decomposition space, and the decision ladder are of great importance. The AD space is a two-dimensional matrix that helps to create a map of the work domain. Along the top is the decomposition hierarchy (part-whole), which allows a user to "zoom in (or out)" from the top-level system down to low level components. Along the side is the abstraction hierarchy (means-ends), which ranges from the abstract purposes to the concrete. For example, the top left cell represents the purposes of the system, a rough functional view of the work domain. This model aids in comprehending the complex system by depicting it in an intuitive way. Also, the abstraction hierarchy supports goal-directed problem solving by incorporating the three questions important to every user: Why? What? How? The level in the hierarchy that is observed at a particular moment in time represents the *what* level, the level above represents *why*, and the level below represents *how*. This aids in identifying possible errors (components or functionality that are needed). While it may take a large amount of effort to construct the "map," it gives the user the ability to adapt, since it applies broadly (as opposed to a sequence of actions, directions).

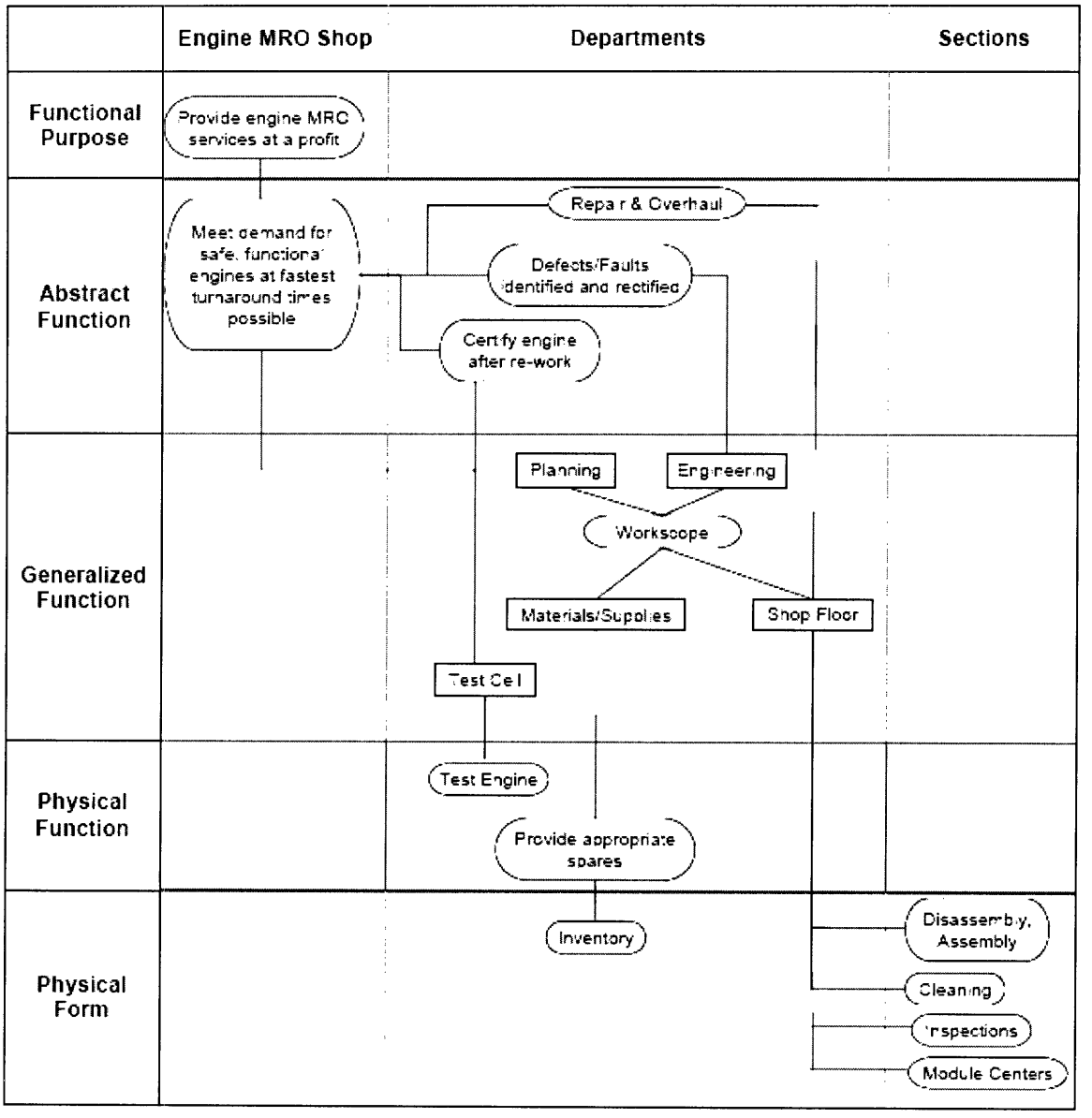


Figure 3.6: Engine MRO shop manager's abstraction-decomposition space [16]

The decision ladder helps to decompose a task into its constituent information processing activities. Since decision-making is often not linear, the model allows for jumps between states of knowledge. This template helps to identify the requirements necessary for the task to be completed. It is useful after the fact to determine if requirements completely cover what is required. It also is of assistance in determining which activities would be beneficial to automate.

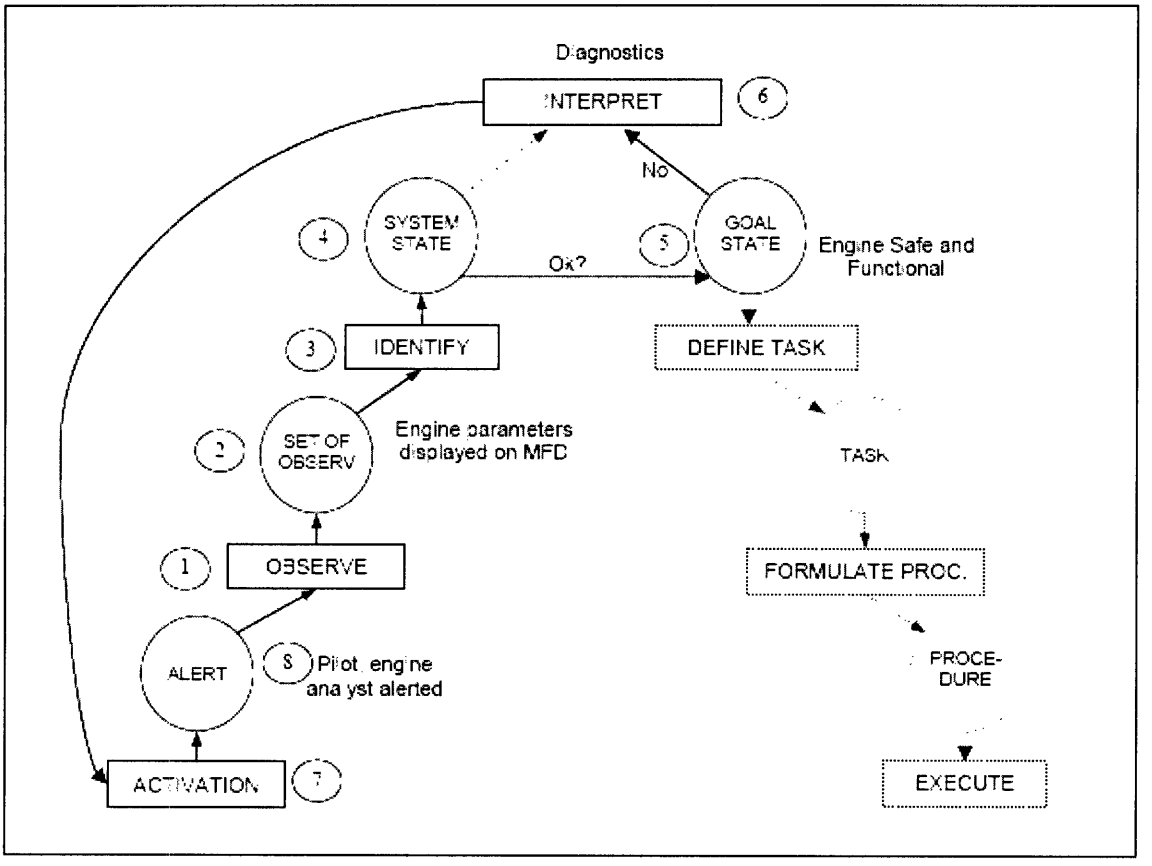


Figure 3.7: Decision Ladder [16]

In this normal operating mode, the system operates in the following sequence:

1) Observe:

   On-board engine monitoring system observes engine parameters in-flight.

2) Set of Observations:

   Engine data is collated by engine monitoring system and displayed on multi-function displays for the flight crew.

3) Identify:

   The monitoring system attempts to identify the engine's system state.

4) System State:

   The engine's system state is identified and classified as normal or abnormal.

5) Goal State:

   If engine is normal, then the goal state is achieved. But if the engine is operating abnormally, the monitoring system will run a diagnostic routine.

6) Interpret:

   Preliminary diagnostics are used to determine the criticality of the engine malfunction, namely if this is safety-critical or a minor malfunction that can be fixed at a later time.

7) Activation:

   The alerting system is activated

8) Alert:

   The flight crew are alerted receives a fault message on the CMC, the engine displays change color and an audio warning tone may be heard.[16]

The CWA framework assists design in many ways. First, the abstraction-decomposition matrix helps the user gain insight into the sociotechnical environment. One can view the system in a structured comprehensible display. Engineers are also made aware of the constraints and assumptions they may have not had without this structure. Likewise for the decision ladder, by thinking about the tasks from an information-processing standpoint, users can gain new insights. Finally, CWA is useful in building a culture with safety measures put into place. It helps to build a healthy environment where workers make decisions, within the constraints of the system.

Is it necessary to have use valuable resources to create 'extra' map? The person designing the system should have a great understanding of the boundaries of the system, whether there are holes or not. But the map is necessary for handling unplanned events. Consider a standard city map. Of course, there is not a need for the map when traveling to well known places. But the value of the map comes in when traveling to new places, i.e. a requirements change. Without the map, one may overlook an assumption that could

have plainly seen if a map were available. Perhaps more important to mention, the map is vital to someone new to the area, i.e. one maintaining the system. This engineer could read documentation (like reading a lengthy list of 'directions') but a map would be invaluable to this person.

3.2.2 Resources Model

The resources model is an approach to identify the key cognitive resources used in work. The model has two central components: the abstract information structures, and how these structures are utilized.

There are six information structures that are defined at an abstract level (they do not claim that these are the only possible). The abstract information structures are plans, goals, affordances, history, action-effect relations, and states. Each of these information structures may be represented externally, internally in the head of the user or more often, distributed across the two. It would be quite useful to know what combinations of structures are needed for the tasks in the system. It is also important to know the optimal way to represent these structures for each task (it may vary from task to task). The information structures will now be described in more detail.

A plan is simply a sequence of actions, events or states that can be carried out. Examples of externally represented plans include a computer program and a pilot's checklist. Likewise, a pilot also must store plans internally (emergency procedures). A place marker of some time is needed in order to utilize the plan. This can be accomplished with the use of the history resource. A goal is a required state of the system. A pilot following a heading is using an internal representation of a goal. The pilot could also use the flight director (external representation), which indicates how the pilot should steer the aircraft to stay on the heading. Affordances are the set of possible next actions that can be taken by a user at a given state of the system. An example of an external affordance is

a menu. It allows you to select from a list of actions to move the system from the current state. A history is a sequence of actions, events, or states that have already passed. Although similar to the plan structure, a history is a linear sequence. A plan can consist of branches and even loops. The history resource is important for web browsers (the back button), but of course, this resource is also often internalized (remembering a familiar route). An action-effect relation is a link between an action or event and a state where the state is what will result when the action or event is executed. The manual for the Windows XP operating system informs the user that the action of holding down ALT and TAB would transport them from the current application to the next open application. The state is the complete set of relevant features of the systems objects. A pilot requires state information such as altitude, attitude, velocity, and vertical speed.

The resources model also introduces the concept of interaction strategy and describes the way in which strategies can exploit different information structures for action. For example, people can interact with the same display in many different ways. There are four main strategies discussed (and again, there is no claim these form an exhaustive list). These are plan following, plan construction, goal matching, and history-based selection. In plan following, the user proceeds through a list of actions until the end of the list. It is necessary for the plan to have already been created. Also of importance is the interaction history (to know which action in the plan one is located) and the state and goal resources to determine whether the plan set forth was accomplished. Plan construction is a more complex activity. In this case, the user is required to compare the current state to the goal state and select possible actions to take to reduce the difference. This will most likely have to be an iterative process. Plan construction must balance the goal, states, affordances, and action-effect resources. Adding to the complexity is the fact that the affordances can change while proceeding from state to state. Goal matching refers to the situation where a user does not follow a plan but attempts to achieve the goal "on the fly." They use affordances to try to match the goal state. The key resources for this activity are the goal, affordances, and action-effects. Finally, history-based selection refers to the decisions made based on past results. For example, in a restaurant, a strategy

for selection includes eliminating choices that you have tried previously and disliked. Important resources to this strategy are the goal, affordances, and history.

The information structures and the interaction strategies contained in the resources model can help to design and build more user-friendly systems. The model helps to decompose the tasks required for the system by the particular cognitive needs. This focus on cognitive resources may help to determine appropriate requirements for the system. It provides an approach to compare how different strategies are performed.

As with the decision ladder in the Cognitive Work Analysis, this model can help to identify which portions of the task it would be beneficial to automate, and what resources are best kept "internal" for this task.

It can help to create better support tools by comparing the combinations of resources utilized. Finally, it lays a framework to measure how to represent information. Their work shows that particular combinations of internal resources and external resources may be optimal for certain tasks (rather than the paradigm to externalize as much as possible to minimize the cognitive load.

3.2.3 Intent Specification

Intent specifications are an interesting approach proposed by Professor Leveson [40] to creating a human-centered software specification. It is designed to support human problem solving and the tasks that humans must perform in software development and evolution. The approach is built on research in systems theory, cognitive psychology, and human-machine interaction. The underlying ideas have been developed and used successfully in cognitive engineering by Vicente and Rasmussen for the design of operator interfaces, a process they call ecological interface design [64].

This section will first describe the principles behind this approach. The details of the intent specification will then be given. Finally, the key benefits to be drawn from this approach will be described.

The goal of the intent specification is to support human problem solving and in particular, to develop and maintain software systems. This can help by providing the engineers tools that will assist them in designing and developing complex systems. At the same time, it is necessary for the tools to facilitate communication between the many people working on the project. The intent specification assists the engineer with this difficulty by integrating the formal and informal aspects of software development. As has been mentioned earlier, there is a need to incorporate the nonfunctional requirements early in the lifecycle and this approach integrates them in a useful way. Also, this approach provides the necessary information to the problem solver, without restricting their freedom to apply various strategies.

The intent specification is based on the basic systems engineering process as shown in the figure:

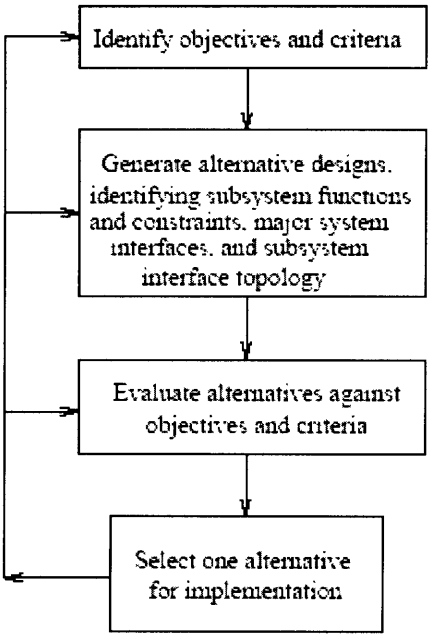


Figure 3.8: System Engineering Process

The specification is also based on three aspects of interface design advocated by cognitive psychologists: (1) content, (2) structure, and (3) form. The content and structure aspects will now be described.

The content of the specification is vital, since "people tend to ignore information during problem solving that is not represented in the specification of the problem."[40] Incomplete specifications can actually leave the users worse off than having no document at all. The content in a common specification should define the following: the system boundary, inputs and outputs, components, structure, interactions between the components, and the goals of the system. Finally, intentional information behind the decisions made above must be included.

The structure of the specification also strongly affects the performance of the development team. If it is difficult for a user to extract the appropriate information from the document, it will not be very useful. A key to creating this structure is to understand

that the complexity of the system depends on its observed complexity. Depending on how detailed the model is, any object can be extremely simple or complex. The aim is to give the user the ability "zoom in or out" with regards to the complexity. The intent specification utilizes a means-ends hierarchy to do this (the means-ends hierarchy was introduced in section 3.2.1). This provides useful support for decision-making. "Consideration of purpose or reason has been shown to play a major role in understanding the operation of complex systems."[55]

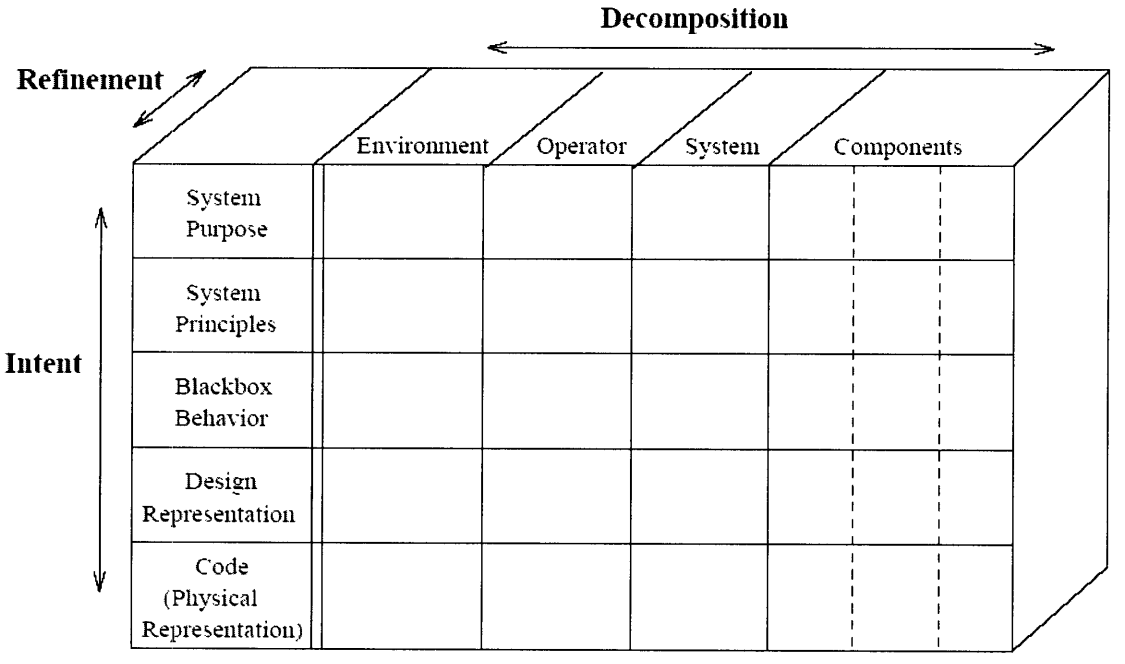


Figure 3.9: Blank Intent Specification [40]

This approach was successfully used to extend the formal TCAS II aircraft collision avoidance system requirements specification. Intent information and other info was included that cannot be expressed formally but is necessary in a complete system requirements specification. This gives a great example to work from for software systems.

There are many aspects of intent specifications that will be useful. The Cognitive Work Analysis exhibited that the structure of the specification is very important to having a good understanding of the system. It assists in discovering problems that may have otherwise been hidden. It also helps to see the intent related dependencies so that users can attempt to minimize them. This approach shows that it can be usefully applied to software engineering. This is also an example of a specification that many groups of people can understand and use. Much effort in projects can be wasted on building a common understanding. The importance of intent information to a system cannot be overlooked. This information is key to communicating the main aspects of the system. The means-end hierarchy assists by simplifying search as well as by providing traceability. This ability to track design decisions between multiple levels helps the users to cope with the cognitive demands induced by the complex system. The proposed approach builds upon the intent specification as will be seen in chapter 4.

CHAPTER FOUR: SUGGESTED APPROACH

The last chapter introduced approaches that have been used to design and develop quality products, as well as approaches tailored to assist in the development of complex systems. This chapter will introduce the proposed approach to improving upon the generation of software specifications. This approach is based upon the positive aspects of each of the previous models and extends their benefits to the design of software. This can be done by taking a customer centric design approach (represented by the hardware processes) and integrating these benefits with cognitive support tools. In this chapter, more detail will be given on what the proposed approach entails. A few examples will then be given of some of the benefits such tools may provide.

4.1 Customer Centric Design with Cognitive Support

There is a rich body of literature on approaches to engineering requirements. The suggested approach adds to these approaches by supplying tools with the intent of developing a robust, learning culture. Only with such a culture will key practices be passed on and expanded upon. To accomplish this, new design and cognitive support tools will be created to assist in the generation of software specifications. This can be done by keeping designers and developers focused on the key goal: satisfying the customer. (or as Chris Peters has said, at Microsoft, our job is not to write code, it is to ship products. [10]). Using key aspects from the product and cognitive approaches, engineers can build well-structured documents that are prioritized with the customer in mind. This will help in decreasing the amount of errors, improving comprehension and communication, and motivation. By improving upon the decision support resources software engineers have access to, the quantity of errors can be reduce early in the process, and save in the costliest of the life cycle phases, maintenance. The tools will also make it easier to comprehend the system for those designing, implementing, and maintaining the system.

The difficulty in creating quality software lies in the creation of the software specification. Software is inherently a complex construction. Engineers still have tremendous difficulty representing the software specification in intuitive ways, and communicating this representation to others. These difficulties can be overcome. The assumptions and constraints can be known in the user's work environment, the difficulty is identifying them and referencing them when a decision relies on them. Experts in the field are able to build complex maps of the software in their head. These cognitive support tools will assist designers in building these maps. With these tools, one can minimize the amount of requirement changes late in the life cycle since they incur such tremendous costs to the project. Likewise, these tools can supplement the incomplete understanding of the requirements set a user has to ensure that the design is robust.

The proposed approach is to utilize the customer-centric approaches successful in hardware systems and create new tools that will support the users cognitive load. The hardware approaches are adept at taking customer input and transforming this data into technical requirements. The cognitive tools assist in organizing information in ways that they will allow the user to work more productively. Both of these aspects should be utilized to create a requirements framework: a map to help the designer to visualize the system so that they are able to spot problem areas. This will help to gain a better understanding of the structure of the set of requirements for the user's particular system. Having this understanding would facilitate the discovery of any holes that need to be filled in the requirements set.

The first step is to have a solid understanding and representation of the work domain. It is well known that outside influences can play a huge part in the success of a software project [64]. Aspects of approaches such as the cognitive work analysis and the intent specification will assist in documenting constraints and assumptions that may have been missed in the work environment. Not only in identifying them, but in structuring them in a useful way (i.e., not just written in the back of the document, but as a constraint, a boundary that the cognitive tool can make others aware of) Also crucial to supplying

decision support is to ensure that the intent behind the previous decisions made are incorporated into the documents and tools.

In the next step, identifying the necessary requirements for the system, decisions must be based off of the information derived from the work domain analysis. A number of different approaches for identifying requirements were shown. Successful approaches in the product domain exhibit the importance of concentrating on the customer requests. More so, prioritizing the requirements gives greater insight into what the structure of the system should look like. The importance of considering the environmental and social aspects of the system was also shown, since they can have as large an effect on success as the technical aspects. These processes also must be incorporated to reinforce good practice and promote a healthy work environment. The metric thermostat illustrated that it is possible to shape the culture of the company in a productive manner. Decision support tools must be build that can provide as complete a picture of the environment as possible; not every detail, as every model is incomplete, but the details important to each level of the system.

One aspect of the benefits these tools will provide can be seen as similar to those found from the Ada programming language. A study conducted in 1995 showed that Ada had lower development costs than the C programming language. They found that Ada encouraged its users to spend more time in writing documentation for their code. This extra information derived a number of benefits. Since it can be somewhat difficult to compile a program in Ada, it has a higher probability of successfully executing once it does. Also, there is a general tendency to apply a quick fix to software. "For example, C allows users to create a global variable without registering it in a ".h" file. C allows users to avoid strong typing easily." [71] The proper constraints and incentives can create the right environment. The motivation can be provided for users to document the structure and intent of the system. Documentation has been a laborious task, particularly as it seems developers dislike this task (and it is exceptionally difficult). Automatic documentation (javadoc) has been a goal for some time, and again works well in some instances. Much of the intent will still be missing however. The increased use of chat

windows and bug logs (passive documentation) has been able to capture some of this intent in a non-intrusive manner.

The current practice in requirements generation involves structuring the requirements in useful ways, as well as proposing to use Quality Function Deployment to assist in identifying missed requirements. How is the suggested approach different? This approach incorporates aspects of cognitive engineering with the best practices of a number of successful product approaches. For example, the affinity graph, hierarchy trees, and the house of quality (of QFD) are useful for organizing information, but they do not help to construct a map of the quality (and breadth) possible using cognitive approaches. Cognitive engineering gives one the ability to build into the specification the structure and the content vital to making sound decisions in complex systems (work domain and worker competencies for example).

The proposed approach is an important step to being able to comprehend complex systems, to build better requirements specifications, to be able to identify requirement 'holes' and errors. Those required to maintain these systems would be able to create a map more quickly than current practice. Those who develop these systems will be able to identify trouble areas more easily. And those who will use these systems will have there needs matched more frequently and have their opinions documented in an efficient (product proven) way. It will lead to better designs, lower bug rates, lowering total cost, as well as the most significant cost, maintenance.

One way to measure the promise of this approach is to see how it can improve upon current practice when looking at Brooks' list of difficulties: complexity, conformity, changeability, and invisibility. Expert developers are able to build accurate maps of the software systems they create. While it may not be known how exactly they do it, it is known to be possible. Can this situation be likened to navigation; can one create software maps that can be used by 'novices' to understand the landscape? The key to software comprehension lies in constructing and describing the system at appropriate levels of abstraction. A tool built upon the abstraction-decomposition matrix could be the first step to building a software map. And with a solid understanding of the work domain

constraints, as well as established standards, engineers will know the constraints from within which the software can be manipulated. Problems generally arise from an incorrect mental model or incorrect assumptions. Many problems will be created if one does not take into account the whole socioeconomic system: the environment, system architecture, user requirements, software design options, and business needs.

There are a number of quantitative ways to check how successful this approach is. First and foremost, the amount of errors in the software specification should markedly decrease. The cost to maintain these software systems should decrease as well. These results are directly related to the level of comprehension the users have of the system. To see how this approach can useful in developing quality requirements specifications, the next section will look at examples of a few possible cognitive tools.

This approach is a guide that may lead to tools that will make software manageable. Every model has limitations and the key is to know these limitations, and apply the correct model at the correct time. It is necessary for every engineer to develop a toolset from which to they can choose the right tool for the task at hand. The proposed approach may help users to visualize complex systems.

4.2 Possible Tools

Now that the concepts behind the suggested approach have been presented, decision support tools built on the approach can be described. The purpose of this section is to introduce these tools and explain how they will be able to facilitate requirements generation. The first tool will help to construct a prioritized map of the system. Utilizing features from CWA and QFD, it will construct a complete picture of the work environment and detail the prioritized constraints and assumptions. The second tool will help to identify types of functionality needed at a higher level. By prioritizing the resources needed, complex systems will be easier to interact with. The third tool is an

extension of the House of Quality matrix. It places emphasis on the cognitive aspects of both the customer and technical requirements.

4.2.1 Prioritized Map

To improve upon current approaches to requirements generation, a decision support tool is needed to assist the user in creating a mental model of the system. QFD performs well at transforming customer requirements into technical requirements in a structured way, but it does not incorporate many of the constraints necessary to consider. There may be numerous assumptions of which the customer and the designer are not aware. How can it be ensured that all (as many as possible) of the requirements are captured?

In section 3.2.1, the necessity of performing a work domain analysis was shown. The constraints must be known to generate the requirements since the control tasks receive information from the work domain, and using this info, perform actions on the work domain. Before the tasks to perform can be determined, all of the environmental constraints imposed upon the system must be well understood. Rasmussen's abstraction-decomposition space displays this information well. One is able to use this tool to help determine what tasks should be performed in the particular environment. .

The affinity graphs and hierarchy trees in QFD can be useful in uncovering missed requirements, but they do not have the inherent, intuitive structure that the abstraction-decomposition space contains however. One can spot a critical requirements hole in the abstraction-decomposition space, while unable to in the affinity graph. A key aspect of QFD though, is the customer-centric foundation. The House of Quality, by ranking and prioritizing the requirements, can quickly determine the importance of different components.

This is incorporated into the prioritized map. Based on the intuitive structure of the abstraction-decomposition space, the requirements and the interconnections are assigned weights. These weights represent their relative importance to the customer and the system being constructed. With this prioritized map of the work domain, the control tasks can be identified. It will give constraints as to what requirements are feasible (just as a city map dictates where an automobile can drive).

4.2.2 Prioritized Resources

In order to correctly identify the necessary requirements for complex systems, the cognitive load imposed upon the participants must be well understood. A useful framework would be able compare tools and determine whether they meet the needs to perform the required work. The resources model provides an interesting approach to represent the types of cognitive resources necessary for particular tasks.

One approach to utilize this model would be to determine how much of each resource is required for the user to achieve their goals for particular tasks. For example, is it more important to have a very detailed description of the current state, and not as important to consider the possible affordances (set of possible actions) for a particular application? It is easy to see that a system could need extensive detail with regard to a couple of information structures (plans, goals, affordances, history, action-effect relationships, current state) and need little to zero information on others.

While sitting at a restaurant for example, it may seem obvious that affordances (provided through the menu) are a necessary support for decision-making. How could the menu be improved? Consider a computerized menu at the center of the table. It could elicit customer input to determine the goal state, current state, and history for example. The customer could tell the menu that they are somewhat hungry, would like a light lunch, and prefer to have chicken. From this, a tailored set of affordances can be generated.

Meals with ingredients not in stock could be taken off the menu (current state). Even with the standard menu, one can see how these resources are used (when categorizing the meals by dominant meat type for example).

The resources model can be quite useful to compare different cognitive tools, and see where they differ on particular aspects. The resources model can also be a great guide to ensuring the correct structures are built into a tool. (imagine web browsers without the ability to look up visited websites: history) It can help to determine where the resources are allocated among users and artifacts. The difficulty lies in determining what selection and 'quantity' of information structures are most appropriate for the user's system. The Value Curve may assist with this problem. This curve does a great job of quickly displaying the strengths and shortcomings of a particular product. Requirements similar in context can be analyzed according to the information structures detailed in the resources model. Once this base line has been established, the Value Curve can show the engineer where possible improvements may lie. The Value Curve can also help to filter out the resources that the industry takes for granted and should be eliminated. They can see which resources should be reduced well below the current standard, or raised well above the current standard. And they can identify the resources that should be created of which the industry has never offered.

4.2.3 The Cognitive Customer

The purpose of this tool is to expand upon the success of QFD; to incorporate into the House of Quality cognitive aspects that may be overlooked. To do this, the information structures from the resources model will be included both in the customer and technical requirements. As was seen in section 3.1.1, both the left side (customer requirements) and the top (technical requirements) are divided into common groups. The cognitive requirements, referencing the resources model, could form one such group. The customer would be able to prioritize which resources are most important to them. For an

application such as an automobile cruise control, the requirement to display the current and goal states may have a high priority and a plan a low priority. The customer could then prioritize the requirements similarly to that which is done in the standard House of Quality. This improves upon the standard House of Quality by explicitly recognizing the cognitive resources necessary for the customer requirements. Without this ability, key attributes could be looked over.

And likewise, focusing on the cognitive requirements, along the top of the House of Quality with the other technical requirements, would be useful. This will ensure that these aspects as well as constraints are considered in the design. Perhaps more important however, is the ability to determine if there are conflicts between the technical and cognitive requirements. This can be determined by using the Technical Correlation Matrix (the triangular matrix located at the top of the house).

This improves upon the standard House of Quality by explicitly stating the cognitive resources necessary for the technical requirements. QFD could miss important requirements. There is a necessity to support QFD with a construct that assists in discovering possible 'holes' in the design. The resource model could be one approach to discovering these holes.

4.2.4 Other Tools

Besides the three tools described above, many others can be constructed grounded on the suggested approach. Aspects of the decision ladder (mentioned in section 3.2.1) could also be successfully incorporated with the customer-centric approaches. There is great promise in using cognitive support tools to improve upon the generation of software specifications. To see this, the next chapter will apply the proposed approach to an automotive software application: adaptive cruise control.

CHAPTER FIVE: AUTOMOTIVE SOFTWARE

There is no need to speak of the importance of the automobile in our lives; with over 200 million cars on the road in the US alone, there may soon be approximately one car per person. What may not be well known though is the growing importance of software in our autos. "Within the automotive industry, more and more innovations are based on electronics and software to enhance the safety of the vehicles, but also to improve the comfort of the passengers and to reduce consumption and emission." "DaimlerChrysler experts estimate that 80 percent of all future automotive innovations will be driven by electronics, 90 percent thereof by software." [17]

Automobile companies are in the process of shifting their vehicles from a mechanical to software- and electronics-based vehicle design development. To gain a better perspective of this change, in 1980, electronics accounted for 2% of a car's total cost. In 1997, this figure grew to 10 to 15%, and in some present-day cars it has reached almost 30%. [11] "A current example for the high degree of software-related complexity of today's cars is provided by the current S-class Mercedes-Benz which contains more than 50 controllers, more than 600,000 lines of code, three different bus systems and approximately 150 messages and 600 signals." [17] The Peugeot 607 has the calculation power of a 1982 Airbus A310." [11]

Software has become a vital aspect in determining a company's ability to be competitive. As time passes, customers will demand more and more software-intensive systems, such as robust communication systems, navigational tools, adaptable engines, intelligent cameras and sensors, and perhaps soon an 'autopilot.'
Software has also given them the ability to add functionality in a much shorter time frame. It is viewed as the best place to make updates since changing the mechanical parts is so cost prohibitive.

With all its benefits, one must recall the many difficulties we've had and continue to have building complex software systems, particularly in critical systems. These problems are currently creating adverse affects in the automotive industry as well. Tools are desperately needed, particularly in "the software development process, software quality management including supplier cooperation, the overall architecture of the in-vehicle software, as well as the ability to specify, to integrate and to test the system." [17]

With the increasing size and complexity of these programs, managing the interaction of the different software functions has been described as both an opportunity, and a nightmare. The job of defining system function and specifications, consequently, has become the job of system engineers. [11]

Automobiles are plagued with the same problems seen previously. "In house experience has shown that, on average, more than 40 percent of errors which occur during the use of an automotive software-based function are attributable to requirements errors caused by immature specifications."[17]

There is a need to find a way to control the complexity of software. This complex sociotechnical environment is exactly the type of system the proposed approach is meant to help. With a set of cognitive support tools, the requirements specification process can be improved upon and reap large cost savings.

Many of the automotive industry's needs are similar to those throughout the software industry. Traceable, un-ambiguous, maintainable designs and specifications are desired. There is a need to have system-level design methodologies to encourage component interoperability, as well as the capability to reuse application level code. Finally, there must be ways to facilitate communication between the different groups involved, which is particularly important when delivering important automotive assumptions to the software engineers who will incorporate them.

There are also a few needs unique to automotive software. As electronics replace more traditional mechanical systems, this transition must be imperceptible to the driver.

Companies have been successful thus far as the increase in electronic systems has been largely unnoticed by the public, except in terms of the benefits it brings through increased safety, driving ease, lower fuel consumption, and emissions. Also of importance to the companies is brand recognition. Just as the design of the car's body is vital to projecting the company's image, the functionality software adds can be just as important.

5.1 Adaptive Cruise Control

Adaptive cruise control (ACC) is a feature that is currently being incorporated into many modern cars. Mercedes-Benz is a leader in this area with their ACC system called Distronic.

The ACC system is an extension of the standard cruise control in most cars today. With the use of forward-looking radar, it scans the area ahead of the car for objects - mainly other cars - and applies the brakes automatically if a collision is likely to occur. If the distance to a vehicle ahead is below a pre-set value, the ACC system is designed to slow the car down, using brakes if required, to match the speed of that vehicle, then returning the car to its pre-set speed once the lane ahead is clear. Steering angle and yaw rate sensors also detect lanes and predict road curves, checking whether any vehicle ahead is in the same lane as the car with ACC.

The controls to the standard ACC system are designed to be similar to conventional cruise controls. The driver sets the speed as usual, but can use a thumbwheel on the center console to select a desired following distance within a safe range, which is displayed on the dashboard within the speedometer. The system may also be configured to correspond to separation distances based on time (2 seconds for example). The system also alerts the driver with an audible chime and flashing red triangle on the display if the car ahead slows rapidly.

The controller uses a feedback and feed-forward control law of

the form:

$$a_d = \dot{v}_d - k(v - v_d)$$

where: a_d is the desired acceleration of the vehicle, v is the speed of the vehicle, v_d is the desired speed of the vehicle and k is a gain set to 0.75.

The software utilizes information taken in from the radar sensor, vehicle speed sensors, and vehicle steering and yaw rate sensors. The Adaptive Cruise Control interacts with other sub-systems of the vehicle in order to perform the overall functionality required such as braking, engine management, and vehicle stability. Thus, the control module is integrated with the software for the anti-lock brakes, traction and stability control systems, and body, engine, and transmission control modules.

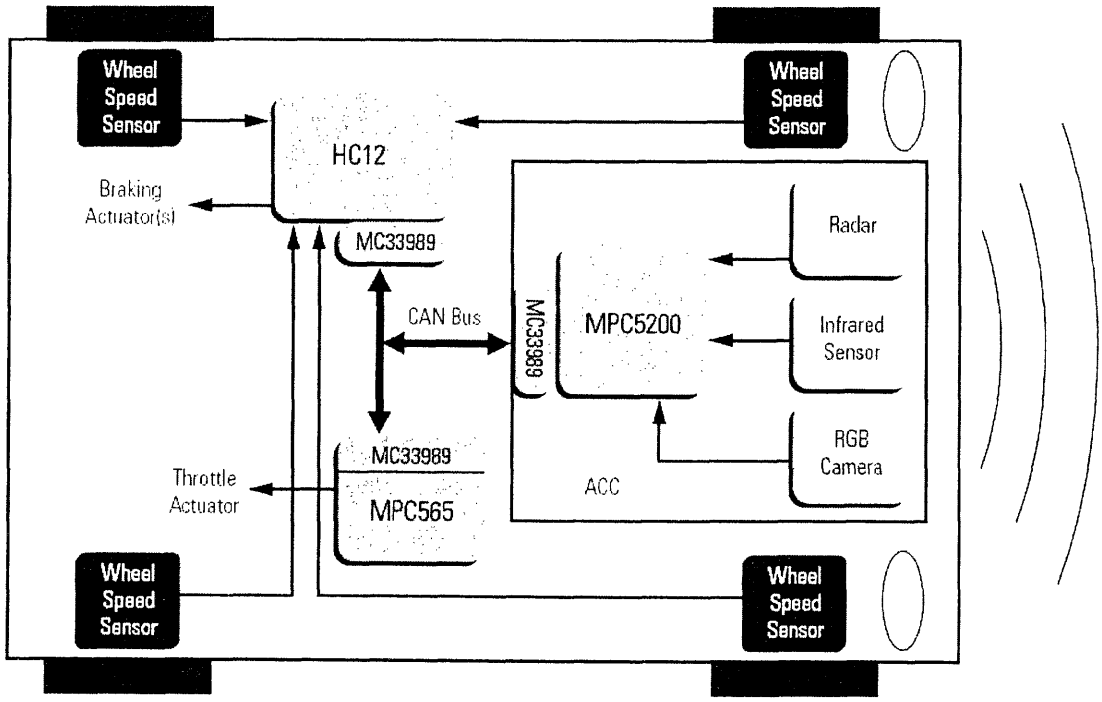


Figure 5.1: Motorola's Adaptive Cruise Control Block Diagram [49]

There are a number of benefits that ACC provides. It reduces the rate of automobile accidents, reduces driver fatigue, and increases fuel efficiency (due to gradual accelerations and decelerations in traffic).
There are also many design challenges. The most important is the cost of building the robust system. There must be safeguards built in to ensure against the ACC not functioning properly. As a safety critical machine, this involves a substantial amount of software validation and verification. To help ease this burden, automobile companies are marketing ACC as a comfort feature:

Holger Meinel, senior researcher at Daimler Benz Aerospace AG (Ulm, Germany), was quick to make a point stressed by all the companies working in this field. "This is not anti-collision radar," he said. "It's not a safety feature, it's a comfort feature." The source of this distinction is concern that if adaptive cruise control is marketed as a safety feature, the first accident that occurs involving a vehicle equipped with millimeter-wave radar will bring a damaging liability suit. That's why companies are at great pains to point out that the driver retains control and responsibility. [13]

Another challenge is in integrating the very different mechanical and software work environments. "They have different expertise, language, reasoning, and most importantly, very different development cycles. The mechanical life cycle is a decade and today there are standard solutions that don't change much. In electronics, the life cycle is more like six months to a year, so even if we agree on a function, six to twelve months later we can give a new function, or make the system less expensive or smaller by changing the hardware. Mechanical engineers aren't used to this speed of change." [11]

Finally, automobile makers have had the rethink how often they should add functionality to their products. "At Fiat, we used to think about establishing a car for approximately six years and only rethinking functionality when we do a redesign. Now we have to think about having maintenance throughout the car life, upgrading the software, with scalable hardware." [11]

5.2 Improvements upon Adaptive Cruise Control

To gain more perspective as to what improvement this approach can make, this section will look at how the proposed approach can be utilized with an application such as adaptive cruise control.

The Adaptive Cruise Control is quite a complex application. How can cognitive tools assist in ensuring that it works properly? As mentioned in Chapter 4, the users must have a deep understanding of the constraints and assumptions in the work environment, in this case, automotive software. And these constraints and assumptions must be documented in a structured and intuitive manner. A prioritized map as discussed in section 4.2.1 would be a good start to building a system "topological map". This would assist the engineer in quickly locating key functionalities at the right "magnification." With the adaptive cruise control software, the user can zoom in from viewing the details of the high-level controller (with modes such as standard cruise control, adaptive cruise control, and off) to viewing the low-level controller (whether to apply the brakes or throttle, and in what amount).

The software for the adaptive cruise control is also coupled to a number of systems. Again, with the use of the prioritized map, the engineer can determine what components and functionalities these interconnections link between. Perhaps more significant however, is the intent and prioritization information provided. There will be many occasions where an expert will have to make a crucial engineering decision and having the intent information, as well as knowing the relative importance, can greatly facilitate these decisions and minimize costly re-work. Particularly for the software engineers who are not as knowledgeable about automotive systems as their mechanical engineering counterparts, the prioritization gives them added insight they otherwise would not have. This will also help to reinforce the goal of the system, to satisfy the customer, as well as raise the comprehension level and ease communication between the engineers.

The prioritized map could also be useful in maintenance applications. Imagine the following scenario has occurred. It has been discovered that the adaptive cruise control applies the brake at apparently random times. How will the engineer go about tackling this problem? An engineer with great experience who had participated in the design of the software may know exactly where the problem lies. This person is often not the one making the repair. But the goal is to be able to capture this expertise in a way that other engineers can utilize. Working with the prioritized map, the software engineer can obtain information they may otherwise have trouble locating. For example, when a radar unit is damaged and needs to be replaced, proper alignment is vital to ensure that performance does not fall. A misalignment of greater than one degree may cause system malfunction [26] This relatively obscure detail may not be known to the software engineer, but the constraint will appear in the prioritized map.

What type of requirements could be discovered utilizing the prioritized resources tool? With a standard cruise control, it is most important to be aware of the current and goal state. The driver will press on the accelerator until reaching the goal speed, and will then press a button on the panel ("Set Speed") to set the current state as the goal state. Pressing the brake pedal will deactivate the cruise control, and pressing a second button on the panel ("Resume") will bring the car back to the goal speed. In this process, there is generally no plan. There is a sequence of actions to follow, represented internally, but the common strategy undertaken is goal matching. What about the adaptive cruise control? The ACC also focuses on the current state and the current state. In this case however, a plan is necessary as braking and acceleration are involved. This approach is also one of goal matching (a set speed, unless following by either a set distance or amount of time), but there are more possible affordances to consider. Not only can the driver apply the brake or accelerator, but the software can initiate these actions as well. New modes must be taken into account. For example, while following from a set distance, what will happen when the accelerator is pressed? One would hope that the break wouldn't be applied to maintain the distance. The software must take into account all such modes. The software could disengage the ACC when either pedal is pressed. This may create more confusion however, as standard cruise control allows the driver to

speed up and then return to the goal speed. Many other Action-effects could be identified. One resource that is useful to incorporate is history. Often times, the driver prefers to maintain a set speed (the speed limit) so it is useful for the software to allow one or possibly multiple set speeds in memory.

This system does place a heavier cognitive load onto the driver than the standard cruise control. There are more possible modes to consider and more controls to interact with. With the standard cruise control, this is generally a minimal amount of information displayed (perhaps a light to inform the driver that the cruise control is on). The display for adaptive cruise control systems are more complicated however. Distronic, Mercedes-Benz ACC offering, shown below, displays a digital reading of the trailing distance within the speedometer.

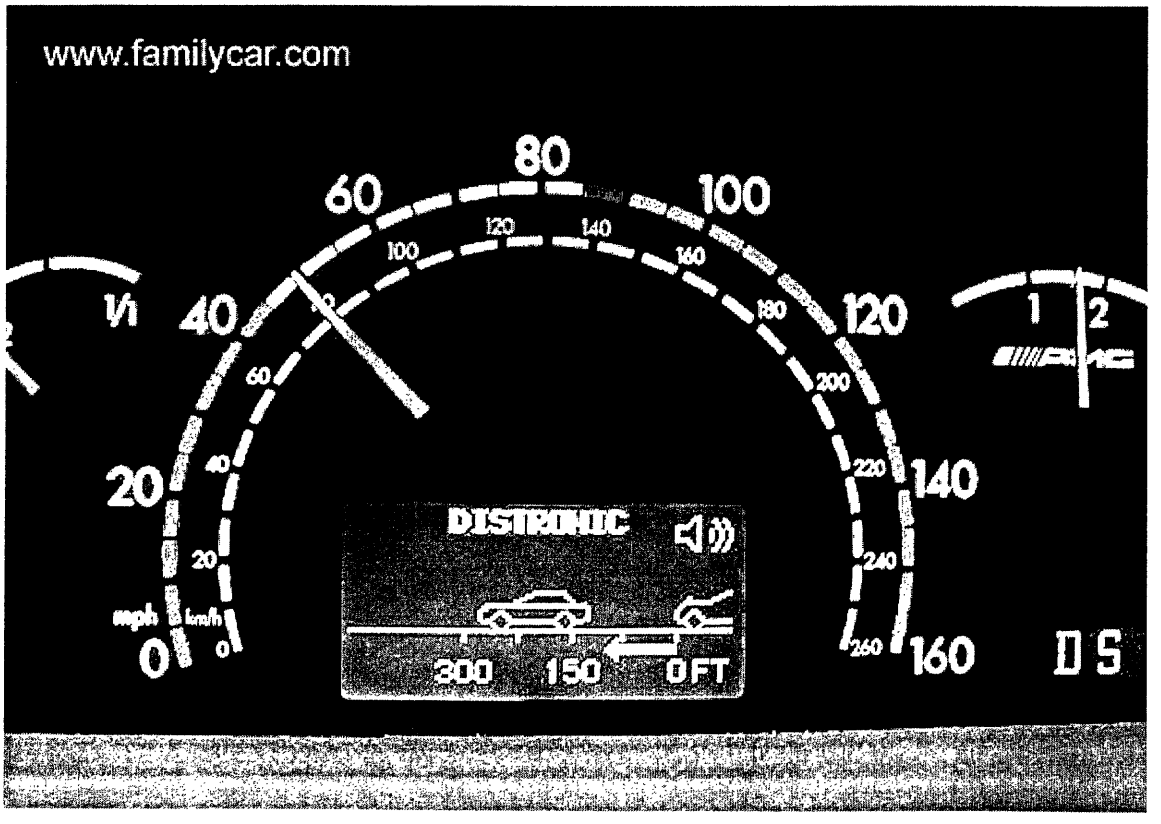


Figure 5.2: Mercedes-Benz Distronic Adaptive Cruise Control System [http://www.familycar.com]

This is an extra burden on the driver and another reason to keep the eyes in the "cockpit" rather than on the road. On the other hand, speed and distance information is now displayed rather than only having access to speed information. The prioritized resources tool could help to evaluate how much of an improvement this is.

Finally, there is the reduced reaction time to consider. Now that the automobile is able to increase and decrease speed at a distance, the driver will interact less with the system, which can reduce fatigue. However, the driver's situational awareness may also be impaired by the inaction. To account for this, perhaps a minimum following distance that is greater than standard will be of assistance.

CHAPTER SIX: CONCLUSION

The challenge in creating quality software is well known. This is largely due to the difficulty in capturing requirements in complex sociotechnical environments. It is understood that software is inherently difficult, and that it is essential to have a solid requirements processes in place. There is tremendous value in reducing the requirement errors early in the process. The need to develop a supportive learning culture has also been shown. Without it, the tools available to software engineers may be of little assistance.

To help to develop this culture, three successful product development approaches were analyzed: Quality Function Deployment, Value Innovation, and the Metrics thermometer. They each gave a unique perspective on how to convert customer input into the system the customer actually wants. QFD provides the House of Quality matrix, which helps to take prioritized customer requirements and create relevant technical requirements, structured in an intuitive fashion. Value Innovation helps to look outside the competitive mindset and evaluate the task from the customer's viewpoint. The Value Curve displays important features of the product (as regards to the product, service, and distribution aspects) in a very simple manner. The Metrics Thermostat described how to shape the company's culture to improve upon key areas of interest (time to market for example) through implicit rewards.

To supplement these tools, three cognitive support approaches were analyzed: Cognitive Work Analysis, Resources Model, and Intent Specifications. The Cognitive Work Analysis assists in identifying and incorporating the work domain constraints into the design, as well as provide useful models to better visualize the system. The Resources Model identifies key cognitive resources and strategies, which can be used to determine the best representation to use for certain tasks. The Intent Specification contributes a

well-structured map of the software system and importantly documents the intent behind the engineering decisions made.

The proposed approach was then presented. Its goal is to improve upon the quality of software specifications with the use of cognitive support tools. With key aspects from successful product approaches and cognitive engineering, decision support tools can be built to improve upon the comprehension of complex systems. The customer-centric focus, championed by the product approaches, is invaluable to the success of the software project. Likewise, the assumptions and constraints identified with the cognitive tools help to ensure that vital requirements are not overlooked early in the lifecycle. Examples of such tools were described, as was the manner in which they may be useful in the effort to improve upon the requirements process. Finally, these tools were applied to a particular automotive application: adaptive cruise control. For the failure rate of software projects to match that of other engineering fields, well-structured models and tools must be developed in order to comprehend these complex systems. Decision support tools can assist with this task as well as assist in developing a culture similar to the other fields.

REFERENCES

[1] Aerotech News and Review. "Dryden, Boeing Use MATRIXx AutoCode in first quad-redundant avionics application." June 30th, 1999. http://www.aerotechnews.com/starc/1999/073099/Boeing_NASA-Dryden.html

[2] Akao, Y. Quality Function Deployment: Integrating Customer Requirements into Product Design, Translated by Glenn Mazur. Productivity Press. 1990

[3] Andersson, Eve, Phillip Greenspun, Andrew Grummet. Internet Application Workbook. 4/2001. http://philip.greenspun.com/internet-application-workbook/

[4] Augustin, Larry, Dan Bressler, Guy Smith. Accelerating Software Development Through Collaboration.

[5] Brooks, Jr., Frederick P. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley. 1982.

[6] Brooks, Jr. Frederick P. "No Silver Bullet." Computer Magazine. April 1987.

[7] Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18(6):543–554.

[8] Carson, Ronald S. et al. "Requirements Completeness." INCOSE Symposium. 1998.

[9] Crow, Kenneth. "Performing QFD Step by Step." DRM Associates. 2000

[10] Cusumano, Michael A. and Richard W. Selby. Microsoft Secrets. Simon & Schuster. 1995.

[11] Design News. "Automobiles merge electronics, mechanics." October 1, 2000. http://www.designnews.com/article/CA138041

[12] Dorner, Dietrich. The Logic of Failure. Perseus Publishing. 1996.

[13] EETIMES. "Adaptive cruise control takes to the highway." October 20, 1998. http://www.eetimes.com/story/OEG19981020S0007

[14] Glass, Robert L. Frequently Forgotten Fundamental Facts about Software Engineering. IEEE Software. May/June 2001.

[15] Glass, Robert L. "Reuse: What's Wrong with This Picture?" IEEE Software. March/April 1998.

[16] Goh, Shaun and Charles P. Coleman. "Sustainment of Commercial Aircraft Gas Turbine Engines: An Organizational and Cognitive Engineering Approach. American Institutue of Aeronautics and Astronautics, Inc. 2003.

[17] Grimm, Klaus. "Software Technology in an Automotive Company – Major Challenges." International Conference on Software Engineering. 2003.

[18] Hales, Robert. "Adapting Quality Function Deployment to the U.S Culture." IIE Solutions. October 1995.

[19] Hatton, Les. "Does OO Sync with How We Think?" IEEE Software. May/June 1998.

[20] Hatton, Les. "Some notes on software failure." October 21, 2001. http://www.leshatton.org/Documents/RSNotes.pdf

[21] Hauser, John R. "Metrics Thermostat." July 2000.

[22] Hauser, John R. and Don Clausing. "The House of Quality." Harvard Business Review. May-June 1988.

[23] Holcomb, Lee. Software Past, Present, and Future: View from the NASA CIO. NASA Software Engineering Workshop, Dec. 1999.

[24] Howard, Alan. "Software Engineering Project Management." Communications of the ACM. May 2001.

[25] Humphrey, Watts S. Managing Technical People. Addison-Wesley. 1997.

[26] I-CAR. "Adaptive Cruise Control." Inter-Industry Conference On Auto Collision Repair. February 16, 2004. www.i-car.com

[27] IBM Business Consulting Solutions. "Automotive Software Foundry Solutions." ibm.com/industries/automotive. 2003.

[28] Jones, Caper. Applied Software Measurement: Assuring Productivity and Quality. McGraw Hill Text; 2nd edition. August 1996)

[29] Jones, Caper. Patterns of Software Systems Failure and Success. International Thomson Publishing. 1996.

[30] Kar, Pradip, and Michelle Bailey. "Characteristics of Good Requirements." INCOSE Symposium, 1996.

[31] Kim, W. Chan and Renee Mauborgne. "How SouthWest Airlines Found a Route to Success." Financial Times. May 13, 1999.

[32] Kim, W. Chan and Renee Mauborgne. "Value Innovation: The Strategic Logic of High Growth." Harvard Business Review. January 1997.

[33] Kim, W.C. and R. Mauborgne, "When Competitive Advantage Is Neither." Wall Street Journal. April 21, 1997

[34] Lauesen, Soren. "Task Descriptions as Functional Requirements." IEEE Software. March/April 2003.

[35] Leffingwell, Dean and Don I. Widrig. Managing Software Requirements: A Unified Approach. Addison-Wesley. 1999.

[36] Lethbridge, Timothy C. and Robert Laganiere. Object-Oriented Software Engineering: Practical Software Development using UML and Java. McGraw Hill. 2001

[37] Leveson, Nancy. Class Notes on Advanced Software Engineering. Massachusetts Institute of Technology. Sept. 1999.

[38] Leveson, Nancy G.. "System Safety in Computer-Controlled Automotive Systems." SAE Congress. March, 2000.

[39] Levenson, N.G. "Evaluating Accident Models using Recent Aerospace Accidents: Part I. Event-Based Models". MIT Technical Report, 2001. http://sunnyday.mit.edu/accidents/nasa-report.pdf

[40] Leveson, N. G. "Intent specifications: An approach to building human-centered specifications." IEEE Transactions on Software Engineering 26 (2000), p15-35.

[41] Leveson, Nancy G.. System Safety in Computer-Controlled Automotive Systems. SAE Congress. March, 2000.

[42] Leveson, Nancy G.. Safeware: System Safety and Computers. Addison-Wesley. 1995.

[43] Lowe, A.J. "Qualtify Function Deployment." 2001. http://www.shef.ac.uk/~ibberson/qfd.html

[44] Mayrhauser, A. von and A.M. Vans. "Comprehension Processes During Large Scale Maintenance." Proceedings of the 16th International Conference of Software Engineering. 1994.

[45] Mazur, Glenn H. "QFD for Small Businesses: A Shortcut through the Maze of Matrices." The Sixth Symposium on Quality Function Deployment. June 1994.

[46] Meyer, Bertrand. "A Really Good Idea." Computer Magazine. Dec. 1999.

[47] Microsoft Corporation. "Trustworthy Computing." White paper. October 2002. http://www.microsoft.com/mscorp/innovation/twc/twc_whitepaper.asp

[48] Monk, A.F. "Action-effect rules: A technique for evaluating an informal specification against principles." Behavior and Information. Vol. 9, pp. 147-155. 1990.

[49] Motorola. "Adaptive Cruise Control." SG2025/D. September 2003. www.motorola.com/semiconductors

[50] National Institute of Standards and Technology. "Software Errors Cost U.S. Economy $59.5 Billion Annually." June 2002. http://www.nist.gov/public_affairs/releases/n02-10.htm

[51] Navarro, Israel, Nancy Leveson, and Kristina Lundqvist. "Reducing the Effects of Requirements Changes through System Design." SERL report. 2000.

[52] Nuseibeh, Bashar and Steve Easterbrook. "Requirements Engineering: A Roadmap." International Conference on Software Engineering. 2000.

[53] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules." Communications of the ACM, Vol. 15, No. 12, December 1972.

[54] Rasmussen, J. "Skills, rules, knowledge: Signals, signs, and symbols and other distinctions in human performance models. IEEE Transactions on Systems, Man, and Cybernetics." 13(3):257–267. 1983.

[55] Rasmussen, J. Information Processing and Human-Machine Interaction: An Approach to Cognitive Engineering. New York: North-Holland. 1986.

[56] Samad, T. and J. Weyrauch. Automation, Control, and Complexity. John Wiley & Sons Ltd. 2000

[57] Simonyi, Charles. "Intentional Programming – Innovation in the Legacy Age." IFIP WG 2.1 meeting. June 4, 1996.

[58] Sommerville, Ian and Pete Sawyer. Requirements Engineering: A Good Practice Guide. John Wiley & Sons Ltd. 1997.

[59] Sharon, David. Meeting the Challenge of Software Maintenance. IEEE Software. January 1996.

[60] Standish Group. "Chaos: The Standish Group Report." 1995. http://www.projectsmart.co.uk/docs/chaos_report.pdf

[61] Sullivan, L.P. Quality Function Deployment. Quality Progress, June, 1986,

[62] Thomas, Bill. "Meeting the Challenges of Requirements Engineering." SEI Interactive Mar. 1999.

[63] Tully, Colin. Improving Software Practice: Case Experiences. John Wiley and Sons. 1998.

[64] Vicente, Kim J. Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work. Lawrence Erlbaum Associates. 1999.

[65] Watkins, Robert and Mark Neal. "Why and How of Requirements Tracing." IEEE Software. July 1994.

[66] Walenstein, Andrew. "Observing and Measuring Cognitive Support: Steps Toward Systematic Tool Evaluation and Engineering." 11th International Workshop on Program Comprehension. May 2003.

[67] Wiegers, Karl E. Software Requirements. Microsoft Press, 2003.

[68] Wright, P., R.E. Fields, and M. Harrison. "Distributed information resources: A new approach to interaction modelling." In Green, T.R.G., Cañas, J.J. and Warren, C.P. (Eds.) Cognition and the worksystem. Proceedings of the 8th European Conference on Cognitive Ergonomics (ECCE 8). EACE Press pp. 5-10. 1996.

[69] Wright, P., R.E. Fields, and M. Harrison. "Analysing Human-Computer Interaction as Distributed Cognition: The Resources Model." Draft-Revised. August 1999.

[70] Zagal, Jose Pablo, Raul Santelices Ahues, Miguel Nussbaum Voehl. "Maintenance-Oriented Design and Development: A Case Study." IEEE Software. July/August 2002.

[71] Zeigler, Stephen F. "Comparing Development Costs of C and Ada." Rational Software Corporation. March 30, 1995. http://www.adaic.org/whyada/ada-vs-c/cada_art.pdf

[72] Zrymiak, Dan. "Software Quality Function Deployment." iSixSigma LLC. 2004. http://www.isixsigma.com/tt/qfd/