SAABNet: Managing Qualitative Knowledge in Software Architecture Assessment

Jilles van Gurp & Jan Bosch
[Jilles.van.Gurp|Jan.Bosch]@ipd.hk-r.se
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
Soft Center, S-372 25 Ronneby
http://www.ipd.hk-r.se/[jvg|bosch]

Abstract

Quantitative techniques have traditionally been used to assess software architectures. We have found that early in the development process there is often insufficient quantitative information to perform such assessments. So far the only way to make qualitative assessments about an architecture, is to use qualitative assessment techniques such as peer reviews. The problem with this type of assessment techniques is that they depend on the knowledge of the expert designers who use them. In this paper we introduce a technique, SAABNet (Software Architecture Assessment Belief Network), that provides support to make qualitative assessments of software architectures.

1. Introduction

Traditionally the software development is organized into different phases (requirements, design, implementation, testing, maintenance). The phases usually occur in a linear fashion (the waterfall model). The phases of this model are usually repeated in an iterative fashion. This is especially true for the development of OO systems.

At any phase in the development process, the process can shift back to an earlier phase. If, for instance, during testing a design flaw is discovered, the design phase and consequently also the phases after that need to be repeated. These types of setbacks in the software development process can be costly, especially if radical changes in the earlier phases (triggering even more radical changes in consequent phases) are needed. We have found that non-functional requirements or quality requirements often cause these type of setbacks. The reason for this is that testing whether the product meets the quality requirements generally does not take place until the testing phase [1].

To assess whether a system meets certain quality requirements, several assessment techniques can be used. Most of these techniques are quantitative in nature. I.e. they measure properties of the system. Quantitative assessment techniques are not very well suited for use early in the development process because incomplete products like design documents and requirement specifications do not provide enough quantifiable information to perform the assessments. Instead developers resort to qualitative assessment techniques. A frequently used technique, for instance, is the peer review where design and or requirement specification documents are reviewed by a group of experts. Though these techniques are very useful in finding the weak spots in a system, many flaws go unnoticed until the system is fully implemented. Fixing the architecture in a later stage can be very expensive because the system gets more complex as the development process is progressing.

Qualitative assessment techniques, like the peer review, rely on qualitative knowledge. This knowledge resides mostly in the heads of developers and may consist of solutions for certain types of problems (patterns [2][6]), statistical knowledge (60% of the total system cost is spent on maintenance), likely causes for certain types of problems (“our choice for the broker architecture explains weak performance“), aesthetics (“this architecture may work but it just doesn’t feel right“), etc. A problem is that this type of knowledge is inexplicit and very hard to document. Consequently, qualitative knowledge is highly fragmented and largely undocumented in most organizations. There are only a handful known ways to handle qualitative knowledge:

• Assign experienced designers to a project. Experienced designers have a lot of knowledge about how to engineer systems. Experienced designers are scarce, though, and when an experienced designer resigns from the organization he was working for, his knowledge will be lost for the organization.

• Knowledge engineering. Here organizations try to capture the knowledge they have in documents. This method is especially popular in large organizations since they have to deal with the problem of getting the right information in the right spot in the organization. A major obstacle is that it is very hard to capture qualitative knowledge as discussed above.

• Artificial Intelligence (AI). In this approach qualitative knowledge is used to built intelligent tools that can assist personnel in doing their jobs. Generally, such tools can’t replace experts but they may help to do their work faster. Because of this less experts can work more efficiently.

In this paper we present a way of representing and using qualitative knowledge in the development process. The technique we use for representing qualitative knowledge, Bayesian Belief Networks (BBN), originates from the AI community. We have found that this technique is very suitable for modeling and manipulating the type of knowledge described above. Bayesian Belief Networks are currently used in many organizations. Examples of such organizations are NASA, HP, Boeing, Siemens [8]. BBNs are also applied in Microsoft’s Office suite where

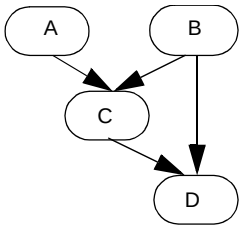

Figure 1. A BBN: qualitative spec.

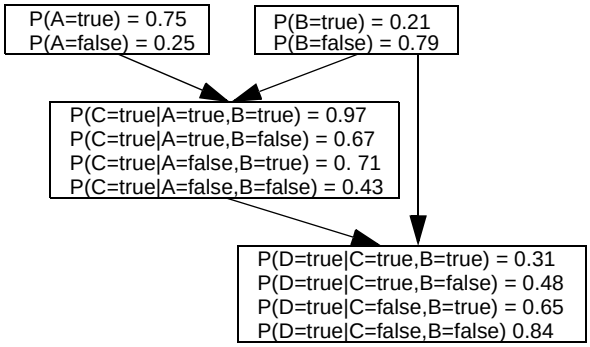

Figure 2. A BBN: quantitative spec.

they are used to power the infamous paperclip [13].

We created a Bayesian Belief Network, called SAABNet (Software Architecture Assessment Belief Network), that enables us to feed information about the characteristics of an architecture to SAABNet. Based on this information, the system is able to give feedback about other system characteristics. The SAABNet BBN consists of variables that represent abstract quality variables such as can be found in McCall's quality factor model [12] (i.e. maintainability, flexibility, etc.) but also less abstract variables from the domain of software architectures like for instance inheritance depth and programming language. The variables are organized in such a way that abstract variables decompose into less abstract variables.

A BBN is a directed acyclic graph. The nodes in the graph represent probability variables and the arrows represent conditional dependencies (not causal relations!). A conditional dependency of variable C on A and B in the example in figure 1 means that if the probabilities for A and B are known, the probability for C is known. If two nodes are not directly connected by an arrow, this means they are independent given the nodes in between (D is conditionally independent of A). Each node can contain a number of states. A conditional probability is associated with each of these states for each combination of states of their direct predecessors (see figure 2 for an example).

A BBN consists of both a qualitative and a quantitative specification. The qualitative specification is the graph of all the nodes. The quantitative specification is the collection of all conditional chances associated with the states in each node. In figure 1 a qualitive specification is given and a quantitative specification is given in figure 2.

By using a sophisticated algorithm, the a priori probabilities for all of the variables in the network can be calculated using the conditional probabilities. This would take exponential amounts of processing power using conventional mathematical solutions (it's a NP complete problem). A BBN can be used by entering evidence (i.e. setting probabilities of variables to a certain value). The a priori probabilities for the states of the other variables are then recalculated. How this is done is beyond the scope of this paper. For an introduction to BBNs we refer to [16].

The remainder of this paper is organized as follows. In section 2. we discuss our methodology, in section 3. we will introduce SAABNet. Section 4. discusses different ways of using SAABNet and in section 5. we discuss a case study we did to validate SAABNet. Related work is presented in section 6. and we conclude our paper in section 7.

2. Methodology

The nature of human knowledge is that it is unstructured, incomplete and fragmented. These properties make that it is very hard to make a structured, complete and unfragmented mathematical model of this knowledge. The strength of BBNs is that they enable us to reason with uncertain and incomplete knowledge. Knowledge (possibly uncertain) can be fed into the network and the network uses this information to calculate information that was not entered. The problem of fragmentation still exists for this way of modeling knowledge, though.

To build a BBN, knowledge from several sources has to be collected and integrated. In our case the knowledge resides in the heads of developers but there may also be some knowledge in the form of books and documentation. Examples of sources for knowledge are:

• *Patterns.* The pattern community provides us with a rich source of solutions for certain problems. Part of a pattern is a context description where the author of a pattern describes the context in which a certain problem can occur and what solutions are applicable. This part of a pattern is the most useful in modeling a BBN because this matches the paradigm of dependencies between variables.

• *Experiences.* Experienced designers can indicate whether certain aspects in a software architecture depend on each other or not, based on their experience.

• *Statistics.* These can be used to reveal or confirm dependencies between variables.

To put this knowledge into a BBN, a BBN developer generally goes through the following steps: (1) Identify relevant variables in the domain. (2) Define/identify the probabilistic dependencies and independencies between the variables. (this should lead to a qualitative specification of the BBN). (3) Assess the conditional probabilities (this should lead to a quantitative specification of the BBN). (4) Test the network to verify that the output of the network is correct.

We have found that the last two steps need to be iterated many times and sometimes enhancements in the qualitative specification are also needed.

The only way to establish whether a BBN is reliable (i.e. is a good representation of the probabilistic distribution of its variables) is to perform casestudies. Performing such case studies means feeding evidence of a number of selected cases to the network and verifying whether the output of the network corresponds with the data available from the case studies. The network can be relied upon to deliver mathematical correct probabilities given correct qualitative and quantitative specifications of the BBN. If a BBN doesn't give correct output, that may be an indication that the probabilistic information in the network is

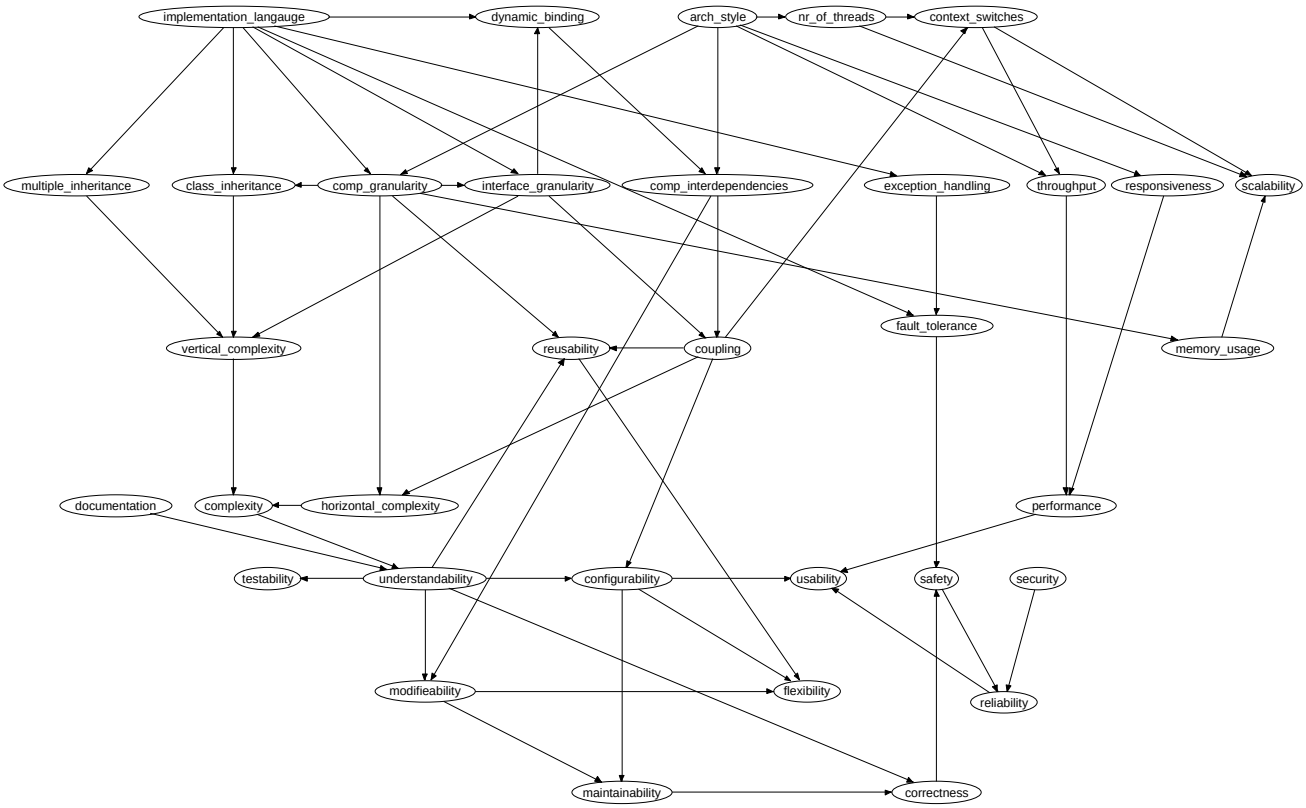

Figure 3. Qualitative specification of SAABNet

wrong or that there is something wrong with the qualitative specification of the network.

Problems with the qualitative specification may be missing variables (over-simplification) or incorrect dependency relations between variables (missing arrows or too many arrows). Problems with the quantitative specification are caused by incorrect conditional probabilities. Estimating probabilities is something that human beings are not good at [4] so it is not unlikely that the quantitative specification has errors in it. Most of these errors only manifest them in very specific situations, however. Therefore a network has to be tested to make sure the output of it is correct under all circumstances.

3. SAABNet

Based on a number of cases we have created a BBN for assessing software architectures called SAABNet (Software Architecture Assessment Belief Network) which is presented in figure 3. The aim of SAABNet is to help developers perform qualitative assessments on architectures. Its primary aim is to support the architecture design process (i.e. we assume that requirements are already available). Consequently, it does not support later phases of the software development process.

3.1. Qualitative Specification

The variables in SAABNet can be divided into three categories:

- Architecture Attributes
- Quality Criteria
- Quality Factors

This categorization was inspired by McCall's quality requirement framework [12], though at several points we deviated from this model. In this model, abstract quality factors, representing quality requirements, are decomposed in less abstract quality criteria. We have added an additional decomposition layer (not found in McCall's model), called architecture attributes, that is even less abstract. Architecture attributes represent concrete, observable artifacts of an architecture.

In figure 3, a qualitative representation of SAABNet is given (i.e. a directed acyclic graph). Though at first sight our network may seem rather complicated, it is really not that complex. While designing we carefully avoided having to many incoming arrows for each variable. In fact there are no variables with more than three incoming arrows. The reason that we did this was to keep the quantitative specification simple. The more incoming arrows, the higher the number of combinations of states of the predecessors. The cleverness of a BBN is that it organizes the variables in such a way that there are few dependencies (otherwise the number of conditional probabilities becomes exponentially large). Without a BBN, all combinations of all variable states would have to be considered (nearly impossible to do in practice because the number rises exponentially). In addition to limiting the number of incoming arrows we also limited the number of states the variables can be in. Most of the variables in our network only have two states (i.e. good and bad or high and low etc.). We may add more states later on to provide greater

arch_style *(pipesfilters, broker, layers, blackboard):* This variable defines the style of the architecture. The states correspond to architectual styles from [2].

class_inheritance_depth *(deep, not deep):* This variable detemines whether the depth of the inheritance hierarchy is deep or not.

comp_granularity *(fine-grained, coarse-grained):* This variable acts as an indicator for component size. A component, in our view, can be anything from a single class up to a large number of classes [5]. In the first case we speak of fine-grained component granularity and in the other case we speak of coarse-grained granularity.

comp_interdependencies *(many, few):* This indicates the amount of dependencies between the components in the architecture.

context_switches *(many, few):* A context switch can occur in multi threaded systems when data currently owned by a particular thread is needed by another thread.

coupling *(static, loose):* This indicates whether the components are statically coupled (through hard references in the source code) or loosely coupled (for instance through an event mechanism).

documentation *(good, bad):* Indicates the quality of the documentation of the system (i.e. class diagrams and other design documents).

dynamic_binding *(high, low):* Modern OO languages allow for dynamic binding. This means that the program pieces are linked together at run time rather than at compile time. Programmers often resort to static binding for performance reasons (i.e. the program is linked together at compile time).

exception_handling *(yes, no):* Exception handling is a mechanism for handling fault situations in programs. This variable indicates whether this is used in the architecture.

implementation_language *(C++, Java):* This variable indicates what programming is used or is going to be used to implement the architecture.

interface_granularity *(coarse-grained, fine-grained):* In [5] we introduced a conceptual model of how to model a framework. One of the aspects of this model is to use small interfaces that implement a role as opposed to the traditional method of putting many things in a single interface. We refer to these small interfaces as fine-grained interfaces and to the larger ones as coarse-grained interfaces. This variable is an indication of whether fine-grained or coarse-grained interfaces are used in the architecture.

multiple_inheritance *(yes, no):* This variable indicates whether multiple inheritance is used in the architecture design.

nr_of_threads *(high, low):* Indicates whether threads are used in the application or not.

Figure 4. Architecture attributes variable definition

fault_tolerance *(tolerant, intolerant):* The ability of implementations of the architecture to deal with fault situations.

horizontal_complexity *(high, low):* We decomposed the quality factor comlexity (see figure 6) into two less abstract forms of complexity (horizontal and vertical complexity). With horizontal complexity the complexity of the aggregation and association relations between classes is denoted.

memory_usage *(high, low):* Indicates whether implementations of the architecture are likely to use much memory.

responsiveness *(good/bad):* Gives an indication of the responsetime of implementations of the architecture.

security *(secure, unsecure):* This variable indicates whether the architecture takes security aspects into account.

testability *(good, bad):* Indicates whether it is easy to test the system

throughput *(good, bad):* This variable is an indication of the ability of implementations of the architecture to process data.

understandability *(good, bad):* This variable indicates whether it is easy for developers to understand the architecture.

vertical_complexity *(high, low):* Earlier we discussed horizontal complexity (the complexity of aggregation and association relations between classes). Vertical complexity measures the complexity of the inheritance relations between classes.

Figure 5. Quality criteria variable definitions

complexity *(high, low):* This variable indicates whether an architecture is perceived as complex.

configuration *(good, bad):* This indicates the ability to configure the architecture at runtime (for compile time configurability see the variable modifiability).

correctness *(good, bad):* This variable indicates whether implementations of the architecture are likely to behave correctly. I.e. whether they will always give correct output.

flexibility *(good, bad):* Flexibility is the ability to adapt to new situations. A flexible architecture can easily be tuned to new requirements and to changes in its environment.

maintainability *(good, bad):* the ability to change the system either by configuring it or by modifying parts of the code in order to meet new requirements.

modifyability *(good, bad):* The ability to modify an implementation of an architecture on the source code level.

performance *(good, bad):* This variable indicates whether implementations of the architecture perform well.

reliability *(good, bad):* Good relieability in SAABNet means that the architeture is both safe and secure.

reusability *(good, bad):* The ability to reuse parts of the implementation of an architecture.

safety *(safe, not safe):* An architecture's implementation is safe if it does not affect its environment in a negative way.

scalability *(good, bad):* With scalability we refer to performance scalability. I.e. the system is scalable if performance goes up if better hardware is used.

usability *(good, bad):* Usability in SAABNet is defined in terms of performance, configurability and relieability. I.e. usable architectures are those architectures that score well on these quality attributes.

Figure 6. Quality factor variable definitions.

accuracy. A short description of all the variables is given in figure 4, figure 5 and figure 6. For complexity reasons, we omitted a full description of all the relations between the variables.

3.2. Quantitative Specification

Since quantitative information about the attributes we are modeling here is scarce, our main method for finding the right probabilities was mostly through experimentation. Since our assessment did not provide us with detailed information, we provided the network with estimates of the conditional probabilities. Since the goal of this network is to provide qualitative rather than quantitative information, this is not necessarily a problem.

A complete quantitative specification of our network is

beyond the scope of this paper. A reason for this is that there are simply too many relations to list here. Our network contains 30+ variables that are linked together in all sorts of ways. A complete quantitative specification would have to list close to 200 probabilities. As an illustration we will show the conditional probabilities of the configurability variable in SAABNet.

Table 1. Conditional probabilities configurability

understandability	good		bad	
coupling	loose	static	loose	static
good	0.9	0.2	0.7	0.1
bad	0.1	0.8	0.3	0.9

Configurability depends on understandability and coupling. In table 1 the conditional probabilities for the the two states of this variable (good and bad) are listed. Since there are 2 predecessors with each two states, there are 4 combinations of predecessor states for each state in configurability. Since we have two states that is 8 probabilities for this variable alone. Note that the sum of each column is 1.

The precision for the output of our model is one decimal. Instead of using the exact probabilities we prefer to interpret the figures as trends which can be either strong if the differences between the probabilities are high or weak if the probabilities do not differ much in value

4. SAABNet usage

It is important to realize that any model is a simplification of reality. Therefore, the output of a BBN is also a simplification of reality. When we designed our SAABNet network, we aimed to get useful output. I.e. output that stresses good points and bad points of the architecture.

The output of a BBN consists of a priori probabilities for each state in each variable. The idea is that a user enters probabilities for some of the variables (for instance P(implementation_language=Java)=1.0). This information is then used together with the quantitative specification of the network to re-calculate all the other probabilities. Since also probabilities other than 1.0 can be entered, the user is able to enter information that is uncertain.

Though the output of the network in itself is quantitative, the user can use this output to make qualitative statements about the architecture ("if we choose the broker architecture there is a risk that the system will have poor performance and higher complexity") based on the quantitative output.

Sometimes the output of a BBN contradicts with what is expected from the given input. Contradicting output always can be traced back to either errors in the BBN, lack of input for the BBN, unrealistic input, confusion about terminology in the network or a mistake of the user. In other cases the BBN will give neutral output. I.e. the probabilities for each state in a certain variable are more or less equal. Likely causes for this may be that there is not enough information in the network to favour any of the states or that the variable has no incoming arrows.

If the output is correct, the structure of the BBN can be used to find proper argumentation for the probabilities of the variables. If for instance SAABNet gives a high probability for high complexity, the variables horizontal and vertical complexity (both are predecessors of complexity in SAABNet) and their predecessors can be examined to find out why the complexity is high. This analysis may also suggest solutions for problems. If for instance maintainability problems can be traced back to high horizontal complexity, solutions for bad maintainability will have to address the high horizontal complexity.

Though the ways in which a BBN can be used is unlimited, we have identified four types of usage strategies for SAABNet:

• *Diagnostic use*. One of the uses of SAABNet is that as a diagnostic tool. When using SAABNet in this way, the user is trying to find possible causes for problems in an architecture. Usually some architectual attributes are known and possibly also some quality criteria are known. In addition there are one or more Quality Factors which represent the actual problem. If, for instance, the implementation of an architecture has bad performance, the performance variable should be set to "bad".

• *Impact analysis*. Another way to use SAABNet is to evaluate the consequences of a future change in the architecture on the quality factors. To do so, the architecture attributes of the future architecture have to be entered as evidence. The network then calculates the quality criteria and the quality factors that are likely for such architecture attributes.

• *Quality attribute prediction*. In this type of use, as much information as possible is collected and put in the SAABNet. From this information, the SAABNet can calculate all the variables that have not been entered. This is ideal for discovering potential problem areas in the architecture early on but can also be used to get an impression of the quality attributes of a future architecture

• *Quality attribute fulfillment*. The first three approaches all required an architecture design. Early in the design process when the design is still incomplete, these approaches may not be an option. In this stage SAABNet can be used to help choose the architecture attributes. This can be done by entering information about the quality factors into SAABNet. The probabilities for all the architecture attributes are then calculated. This information can be used to make decisions during the design process. If, for instance, the architecture has to be highly maintainable, SAABNet will probably give a high probability on single inheritance since multiple inheritance affects maintenance negatively. Based on this probability, the design team may decide against the use of multiple inheritance or use it only when there's no other possibility.

The four mentioned usage profiles can be used in combination with each other. A quality attribute prediction usage of SAABNet can for instance reveal problems (making it a diagnostic usage). This may be the starting point to do an impact analysis for solutions for the detected problems. Alternatively, if there are a lot of problems, the quality attribute fulfillment strategy may be used to see how much the ideal architecture deviates from the actual architecture.

5. Validation

As a proof of concept, we implemented SAABNet using Hugin Lite [7] and applied it to some cases. The tool makes it possible to draw the network and enter the conditional probabilities. It can also run in the so called com-

piled mode where evidence can be entered to a network and the conditional probabilities for each variable's states are recalculated (for a complete specification of SAABNet in the form of a Hugin file, please contact the first author).

All tests were conducted with the same version of the network.

5.1. Case1: An embedded Architecture

For our first case we evaluated the architecture of a Swedish company that specializes in producing embedded software for hardware devices. The software runs on proprietary hardware. We were allowed to examine this company's internal documents for our cases.

The software, originally written in C, has been rewritten in C++ over the past years. Most of the architecture is implemented in C++ nowadays. The current version of the architecture has recently been evaluated in what could be interpreted as a peer review. The main goal of this evaluation was to identify weak spots in the architecture and come up with solutions for the found problems. The findings of this evaluation are very suitable to serve as a testcase for our BBN.

5.1.1. Diagnostic use. The current architecture has a number of problems (which were identified in the evaluation project). In this case we test whether our network comes to the same conclusions and whether it will find additional problems.

Table 2. Diagnostic use

Entered evidence	
documentation	bad
class_inheritance_depth	deep
comp_granularity	coarse_grained
comp_interdependencies	many
complexity	high
context_switches	few
implementation_language	C++
interface_granularity	coarse_grained
Output of the network	
arch_style	layers (0.47)
configurability	bad (0.76)
coupling	static (0.76)
horizontal_complexity	high (0.66)
maintainability	bad (0.71)
multiple_inheritance	yes (0.77)
vertical_complexity	high (0.87)
modifiability	bad (0.90)
reusability	bad (0.68)
understandability	bad (1.0)

Facts/evidence. We know several things about the architecture that can be fed to our network:

• C++ is used as an implementation language

• The documentation is incomplete and usually is not up to date

• Because of the use of object-oriented frameworks, the class inheritance depth is deep.

• Components in the architecture are coarse-grained

• There are many dependencies between the modules and the components

• The whole architecture is large and complicated. It consists of hundreds of modules adding up to hundreds of thousands lines of code.

• Interfaces are only present in the form of header files and abstract classes form the frameworks

• There are very few context switches (this has been a design goal to increase performance)

Based on these architectual attributes we can enter the evidence listed in table 2.

Output of the network. In table 2 some of the output variables for this case are shown. The results clearly show that there is a maintainability problem. There is a dependency between configurability and maintainability and a dependency between modifiability and maintainability in figure 3. So, not surprisingly, modifiability and configurability are also bad in the results. Reusability (depends on understandability, comp_granularity and coupling) is also bad since all the predecessors in the network also score negatively. The latter, however, conflicts with the company's claims of having a high level of reuse.

In SAABNet, reusability depends on understandability, component granularity and coupling. Clearly the architecture scores bad on all of these prerequisites (poor understandability, coarse-grained components and static coupling) so the conclusion of the network can be explained. The network only considers binary component reuse. This is not how this company reuses their code. Instead, when reusing, they take the source code of existing modules, which are then tailored to the new situation. In most cases the changes to the source code are limited though. Another reason why their claim of having reuse in their organization is legitimate despite the output of SAABNet is that they have a lot of expert programmers who know a great deal about the system. This makes the process of adapting old code to new situations a bit easier than would normally be the case.

The network also gives the layers architectural style the highest probability (out of four different styles). This is indeed the architectual style that is used for the device software. As can be deduced from the many outgoing arrows of this variable in our network, this is an important variable. Choosing an architectural style influences many other variables. It is therefore not surprising that it picks the right style based on the evidence we entered.

5.1.2. Impact analysis. To address the problems mentioned, the company plans to modify their architecture in a number of ways. The most important architectural change is to move from a layers based architecture to an architecture that still has a layers structure but also incorporates elements of the broker architecture. A broker architecture will, presumably, make it easier to plug in components to the architecture. In addition, it will improve the runtime configurability.

Apart from architectural changes, also changes to the development process have been suggested. These changes should lead to more accurate documentation and better test procedures. Also modularization is to be actively promoted during the development process. In this test we used the impact analysis strategy to verify whether the predicted quality attributes match the expected result of the changes.

Table 3. Impact analysis

Entered evidence	
arch_style	broker
class_inhertance_depth	deep
comp_granularity	coarse_grained
interface_granularity	coarse_grained
context_switches	few

Table 3. Impact analysis

documentation	good
implementation_language	C++
Output of the network	
configurability	good (0.52)
maintainability	good (0.64)
modifiability	good (0.66)
reusability	bad (0.65)
understandability	good (0.64)
coupling	loose (0.54)
correctness	good (0.75)
comp_interdependencies	few (0.79)

Facts/evidence.
• C++ is still used as a primary programming language.
• Documentation will be better than it used to be because of the process changes.
• The inheritance depth will probably not change since the frameworks will continue to be used.
• The component granularity will still be coarse-grained.
• The component interfaces will remain coarse-grained since the frameworks are not affected by the changes.
• There are still very few context switches.
• The architecture is now a broker architecture.

Output of the network. One of the reasons the broker architecture has been suggested was that it would reduce the number of interdependencies. SAABNet confirms this with a high probability for few component interdependencies. However, the network does not give such a high probability for loose coupling (as could be expected from applying a broker architecture). The reason for this is that the involved components are coarse-grained. While the relations between those components are probably loose, the relations between the classes inside the components are still static.

A second reason for using the broker architecture was to increase configurability. In particular, it should be possible to link together components at runtime instead of statically linking them at compiletime. The low score for good configurability is a bit at odds with this. It is an improvement of the higher probability for bad configurability in the previous case, though. The reason that it doesn't score very high yet is that the influencing variables, understandability and coupling, don't score high probabilities for good and loose. The improved documentation did of course have a positive effect on understandability but it was not enough to compensate for the probability on high complexity. So, according to SAABNet, configurability will only improve slightly because other things such as complexity are not addressed sufficiently by the changes.

5.2. Case2: Epoc32

Epoc32 is an operating system for PDAs (personal digital assistants) and mobile phones. It is developed by Symbian. The Epoc32 architecture is designed to make it easy for developers to create applications for these devices and too make it easy to port these applications to the different hardware platforms EPOC 32 runs on. Its framework provides GUI constructs, support for embedded objects, access to communication abilities of the devices, etc.

To learn about the EPOC 32 architecture we examined Symbian's online documentation [17]. This documentation consisted of programming guidelines, detailed information on how C++ is used in the architecture and an overview of the important components in the system.

5.2.1. Quality attribute prediction. In this case we followed the quality attribute strategy to examine whether the design goals of the EPOC 32 architecture are predicted by our model given the properties we know about it. The design goals of the EPOC 32 architecture can be summarized as follows:
• It has to perform well on limited hardware
• It has to be small to be able to fit in the generally small memory of the target hardware
• It must be able to recover from errors since applications running on top of EPOC are expected to run for months or even years
• The software has to be modular so that the system can be tailored for different hardware platforms
• The software must be reliable, crashes are not acceptable.

Table 4. Quality attribute prediction

Entered evidence	
class_inheritance_depth	deep
comp_granularity	coarse-grained
comp_interdependencies	few
exception_handling	yes
implementation_language	c++
interface_granularity	coarse-grained
memory_usage	low
multiple_inheritance	no
Output of the network	
complexity	low (0.62)
configurability	high (0.55)
correctness	good (0.73)
fault_tolerance	tolerant (0.70)
flexibility	good (0.55)
maintainability	good (0.65)
modifiability	good (0.66)
reliability	reliable (0.74)
reusability	bad (0.64)
usability	good (0.65)
understandability	good (0.52)

Facts/evidence. We assessed the EPOC architecture using the online documentation [17]. From this documentation we learned that:
• A special mechanism to allocate and deallocate objects is used
• Multiple inheritance is not allowed except for abstract classes with no implementation (the functional equivalent of the interface construct in Java).
• The depth of the inheritance tree can be quite deep. There is a convention of putting very little behavior in virtual methods, though. This causes the majority of the code to be located in the leafs of the tree. The superclasses can be seen as the functional equivalent of Java interfaces.
• A special exception handling mechanism is used. C++ default exception handling mechanism uses too much memory so the EPOC 32 OS comes with its own macro based exception handling mechanism.
• Since the system has to operate in devices with limited memory capacity, the system uses very little memory. In several places memory usage was a motivation to choose an otherwise less than optimal solution (exception handling, the way DLLs are linked)
• Components are medium sized.

• There are few dependencies between components. In particular circular dependencies are not allowed.
• Generally components can be replaced with binary compatible replacements which indicates that the components are loosely coupled.

Output of the network. The output of the network confirms that the right choices have been made in the design of the EPOC 32 operating system. Our network predicts that low complexity is probable, high reliability is also probable. Furthermore the system is fault tolerant (which partially explains reliability.). The system also scores well on maintainability and flexibility. A surprise is the low score on reusability. Unlike the previous case, the EPOC 32 features so called binary components. What obstructs their reuse is the fact that the components are rather large and the fact that the interfaces are also coarse-grained.

Also of influence is the fifty fifty score on understandability (good understandability is essential for reuse). The latter is probably the cause of a lack of evidence, not because of an error in the network. The available evidence is insufficient to make meaningful assumptions about understandability. The reason for the bad score on reusability lies in the fact that even though EPOC components are reusable within the EPOC system, they are not reusable in other systems (such as the PalmOS or Windows CE).

5.2.2. Quality attribute fulfillment. Though its certainly interesting to see that the architectural properties predict the design goals, it is also interesting to verify whether the design goals predict the architectual properties. To do so, we applied the quality attribute fulfillment strategy.

Table 5. Quality attribute fulfillment

Entered evidence	
configurability	good
fault_tolerance	tolerant
memory_usage	low
modifiability	good
performance	good
reliability	reliable
Output of the network	
class_inheritance_depth	not deep (0.52)
comp_granularity	coarse-grained (0.83)
comp_interdependencies	few (0.75)
exception_handling	yes (0.80)
implementation_language	java (0.66)
interface_granularity	fine-grained (0.58)
multiple_inheritance	no (0.77)

Facts/evidence. In this case we entered properties that were presumably wanted quality attributes for the EPOC architecture:
• Fault tolerance and reliability are both important for EPOC since EPOC systems are expected to run for long periods of time. System crashes are not acceptable and the system is expected to recover from application errors.
• Since the system has to operate on relatively small hardware, performance and low memory usage are important
• Since the system has to run on a wide variety of hardware (varying in processor, memory size, display size), the system must be tailorable (i.e. configurability and modifiability should be easy)

Output of the network. It is unreasonable to expect our network to come up with all the properties of the EPOC 32 OS based on this input. The output however once again confirms that design choices for EPOC 32 make sense. One of the interesting things is that our network suggests a high probability on Java as a programming language. While EPOC 32 was programmed in C++, its designers tried to mimic many of Java's features (also see [17]). In particular they mimicked the way Java uses interfaces to expose API's (using abstract classes with virtual methods), they used an exception handling mechanism, they created a mechanism for allocating and deallocating memory which is safer than the regular C++ way of doing so. Considering this, it is understandable that our network picked the wrong language.

SAABNet also predicts coarse-grained components which is correct. In addition to that it gives a high probability for the presence of exception handling which is also correct. The network is also correct in predicting no multiple inheritance and few component interdependencies. It is wrong, however, in predicting an low inheritance depth and predicting fine-grained interfaces. The latter two errors can easily be explained since, as we pointed out in the previous case, virtual classes in EPOC can be compared to Java interfaces. This makes the inheritance hierarchy much easier to understand.

6. Related Work

Important work in the field of BBNs is that of Judea Perl [16]. In this book the concept of belief networks is introduced and algorithms to perform calculations on BBNs are presented. Other important work in this area includes that of Drudzel & Van der Gaag [4] where methodology for quantification of a BBN is discussed.

We were not the first to apply belief networks to software engineering. In [14] and [15], BBNs are used to assess system dependability and other quality attributes. Contrary to our work, their work focuses on dependability and safety aspects of software systems.

The qualitative network we created could be perceived as a complex quality requirement framework as the one presented by McCall [12]. Apart from our model being more complex, there are some structural differences with McCall. In our model abstract attributes like flexibility and understandability are decomposed into less abstract attributes (follow the arrows in reverse direction). McCall's decomposition is far more simple than ours is: it only has three layers and there are no connections within one layer. We think that his decomposition is too simplistic for our goal which is to make useful qualitative assessments about software architecture using a BBN. Mc Call's decomposition does not model independencies very well (which essential for a BBN). Many criteria like "modularity" show up in the decomposition of nearly every quality factor. In a BBN that would lead to many incoming arrows. We feel that our model may be a better decomposition because it tries to find minimal decompositions and groups simple quality criteria into more abstract ones. An example of this is our decomposition of complexity into vertical and horizontal complexity. However, continued validation is required to prove our position.

Lundberg et al. provide another decomposition of a limited number of quality attributes [9]. Like McCall's de-

composition, their decomposition is a hierarchical decomposition. We adopted and enhanced their decomposition of performance into throughput and responsiveness. However, we did not use their decomposition of modifiability into maintainability and configurability as we needed a more detailed decomposition. Rather we adopted Swanson's decomposition of maintenance into perfective, adaptive and corrective maintenance [18]. We mapped the notion of perfective and corrective maintenance onto modifiability while adaptive maintenance is mapped onto configurability. A reason for this difference in decomposition is that we prefer to think of modifiability as code modifications and of configurability as run time modifications.

The SAABNet technique, we created, would fit in nicely with existing development methods such as the method presented in [1] which was developed in our research group. In this design method, an architecture is developed in iterations. After each iteration, the architecture is evaluated and weaknesses are identified. In the next iteration the weaknesses are addressed by applying transformations to the architecture. Our technique could be used to detect weak spots earlier so that they can be addressed while it is still cheap to transform the architecture.

SAABNet could also be used in spiral development methods, like ATAM (Architecture Tradeoff Analysis Method) [10], that also rely on assessments. It is however not intended to replace methods like SAAM [11] which generally require an architecture description since SAABNet does not require such a description. Rather SAABNet could be used in an earlier phase of software development.

7. Conclusion

In this paper we have presented SAABNet, a technique for assessing software architectures early in the development process. Contrary to existing techniques this technique works with qualitative knowledge rather than quantitative knowledge. Because of this, our technique can be used to evaluate architectures before metrics can be done and can even assist in designing the architecture.

We have evaluated SAABNet by doing four small case studies, each using one of the four usage strategies we presented in section 4.. In each of the cases we were able to explain the output of SAABNet. There were some deviations with our cases. The most notable one was the low score on reusability in both evaluated systems. We explained this by pointing out that in both cases the companies idea of reuse is different from what SAABNet uses. In general the output of SAABNet is quite accurate, given the limited input we provided in our cases. This suggests that extending SAABNet may allow for even more accurate output.

The sometimes rather obvious nature of the conclusions of SAABNet are a result of the fact that the current version of our belief network is somewhat simple. We intend to extend SAABNet in the future to allow for more detailed conclusions. We also intend to develop a tool around SAABNet that makes it more easier to interact with it. A starting point for building such a tool are the usage strategies we identified. Although our small case study shows that this is a promising technique, a larger, preferably industrial, case study is needed to validate SAABNet.

8. References

[1] J. Bosch, P. Molin, "Software Architecture Design: Evaluation and Transformation", in Proceedings of the 1999 IEEE Conference on Engineering of Computer Based Systems. March 1999.

[2] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, "Pattern-Oriented Software Architecture - A System of Patterns", John Wiley & Sons, 1996.

[3] J. Daly, A. Brooks, J. Miller, M. Roper, M. Wood, "The effect of inheritance on the maintainability of object oriented software: an empirical study", Proceedings of the international conference on software maintenance, pp. 20-29, IEEE computer Society Press, Los Alamitos, CA, USA, 1995.

[4] M. J. Drudzel, L. C. van der Gaag, "Elicitation for Belief Networks: Combining Qualitative and Quantitative Information", Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95), pp. 141-148, Montreal August 1995.

[5] J. van Gurp, J. Bosch, "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", submitted July 1999.

[6] J. Gosling, B. Joy, G. Steele, "The Java Language Specification", Addison Wesley, 1996. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns - Elements of Reusable Object Oriented software", Addison-Wesley, 1995.

[7] Hugin "Hugin Expert A/S - Homepage", http://www.hugin.dk.

[8] Hugin, "General Information", http://www.hugin.dk/gen-inf.html.

[9] L. Lundberg, J. Bosch, D. Häggander, P. O. Bengtsson, "Quality Attributes in Software Architecture Design", Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications, pp. 353-362, October 1999.

[10] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The architecture Tradeoff Analysis Method", Proceedings of ICECCS, August 1998, Monterey, CA.

[11] R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A Method for Analyzing the Properties Software Architectures", pp. 81-90, Proceedings of ICSE 16, May 1994.

[12] J. A. McCall, "Quality Factors", encyclopedia of Software Engineering, vol 2 O-Z pp. 958-969, John Wiley & Sons New York 1994.

[13] Microsoft Research, "Machine Learning and Applied Statistics", http://research.microsoft.com/research/mlas.

[14] M. Neil, B. Littlewood, N. Fenton, "Applying Bayesian Belief Networks to Systems Dependability Assessment", Proceedings of Safety Critical Systems Club Symposium, Leeds, Springer-Verlag February 1996.

[15] M. Neil, N. Fenton, "Predicting Software Quality using Bayesian Belief Networks", Proceedings of 21st Annual Software Engineering Workshop, 1996.

[16] J. Pearl, "Probabilistic Reasoning in Intelligent Systems", Morgan Kaufmann Publishers, Inc. San Mateo 1988.

[17] Symbian, "EPOC World Library", http://developer.epocworld.com/EPOClibrary/EPOClibrary.html.

[18] E. B. Swanson, "The dimensions of maintenance", proceedings of the 2nd international conference on software engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos 1976.