Context-Awareness of Software Systems

A Progressive Context-Aware Add-on Architecture

Author:	Sven Vintges
Tutors:	Dr. Ir. J.A.G. Nijhuis
Second Tutor:	Ir. S. Achterop
Supervisors:	Dr. Ir. J.A.G. Nijhuis and Drs. J. Siljee
Institute:	Rijks*universiteit* Groningen
Date:	17 October 2004

Context-Awareness of Software Systems

A Progressive Context-Aware Add-on Architecture

Author:	Sven Vintges
Tutors:	Dr. Ir. J.A.G. Nijhuis
Second Tutor:	Ir. S. Achterop
Supervisors:	Dr. Ir. J.A.G. Nijhuis and Drs. J. Siljee
Institute:	Rijks*universiteit* Groningen
Date:	17 October 2004
Version	Final 1.0

Preface

This thesis is written as a part of my master study software and systems engineering at the University of Groningen. The last two years of the study are the masters phase and this phase is concluded with a thesis and a presentation of the research results described in this thesis.

The target group of this paper are last year bachelor students, master students and the research staff of any software engineering research group (professors, PhDs, masters etc) who are interested in designing and implementing a context-aware software system.

This thesis can be read in total in a chronologic order. Readers familiar with architectural styles can skip paragraph 4.1 to 4.6. Readers who are only interested in creating a context-aware application and therefore the architecture discussed in this thesis could do with reading chapter 5.

Before studying at the University of Groningen I studied at the polytechnics department of the Hanzehogeschool Groningen and received my bachelor degree in software engineering. Because I was still very young and eager to learn I decided I wanted to study for my masters at the University of Groningen. In two years of study I obtained my Bachelor degree at the university and I am concluding my masters' phase (with this paper as a result). After finishing my study I will go to work at Atos Origin as an Application Developer.

I would also like to use this preface to thank some people, for starters I would like to thank my tutors and supervisors Jos Nijhuis, Johanneke Siljee and Sietse Achterop. Also I would like to thank Ivor Bosloper for providing feedback on my thesis and research results. I would like to thank some family and friends; Silvia (for being so patient), Bob, Rikkie, Yuri (for providing the ability to study and all the mental support), Ruben, Marlies, Julian (for all the children laughs and the joy you emit), Gabor, Arno, Natasja and Jeroen for providing feedback on my thesis.

I hope you enjoy reading this thesis as much as I enjoyed writing it.

Kind Regards,

Sven Vintges

Contents

1 Introduction

Software systems are becoming more complex. The research on design issues is growing and huge steps are made the last decades. From structured programming to object-oriented design and from the first if-then-else-statements to the most sophisticated architectural styles. All these research topics help us design better software systems.

Still these solutions are based on static problems; at the design time often a static reflection of the system is used. But the system will be used in a dynamic environment requesting ever changing requirements. One of the problems is that we want the systems to perform optimal in the changing environment, meaning that the quality of the software system is maintained at the highest level possible.

A way to achieve this is by analyzing and adapting dependant on the context. Context-aware applications make use of their context to improve its quality attributes by adapting. The context can be factors outside the system as well as inside the system (extrinsic and intrinsic). By using this information the system will be improving its quality and serve the user (which can be a human or another computer system, application, component etc).

The aim of this thesis is to get a better view on architectural styles in context-aware software systems. This means I will discuss the current architectural styles in perspective of a context-aware system. Though these styles are interesting enough on their own in most cases we will need a composite style to fulfil the problem at hand.

First we will need to decide what exactly the architecture of software systems is. In [15] the most widely used definition is used:

Architecture can be defined as;

> *"a collection of computational components —or simply components- together with a description of the interactions between these components— the connectors."*

The definition states that an architecture consists of an ordered collection of components, these components are linked with connectors. By defining the requirements of these components and connectors we create an architecture. Just like the architecture of a building describes how each component (bricks) are connected (using cement) and how they are ordered (to create the arch).

There are multiple styles for building such architecture, each with their own characteristic. The *pipes and filters* style is often used for building compilers while the *client-server* architectural style is mostly used for, for example, internet applications.

But what is an architectural style? I will adopt the definition given by Garlan and Shaw [15]:

> *"An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition."*

This means that there are multiple architectures using the same patterns. These styles are intended to solve a certain type of problem. In analogy we could see that in the construction of a house the architecture of building a roof has a certain style. But this does not mean every roof is the same, changing the kind roof tiles used creates a complete new look; but still the same pattern or style is used to build the roof.

In this particular case I am interested in the following question:

Which architectural style(s) are suitable for designing Context-aware applications?

To answer this question I will divide it into four sub questions:

- *Which architectural styles are currently used?*
- *What properties are necessary in an architectural style aiming on context-aware application design?*
- *What is the usability of these style(s) with respect to context-aware design and the described properties?*
- *What should the composite style look like?*

The structure of this thesis is as follows:

Chapter 2	In this chapter I will create the foundation for the rest of my thesis. I will start by discussing the definition of context, I will do this by looking at two often used definitions. Here after I will describe my vision on context-awareness and describe the definition of context and context-awareness I will use in this thesis.
Chapter 3	At current autonomous computing and the context of software systems is gaining interest from multiple groups of researchers. In this chapter I will describe IBMs view on autonomous computing, the attempts of Hewlett Packard to create context-aware laptops and I will discuss three context architectures which are used to create context-aware applications.
Chapter 4	In this chapter I will discuss the architectural styles widely used and adopted by the software engineering community. At the end of this chapter I will discuss the usability of these styles and why I think none of these is instantly usable.
Chapter 5	In the fifth chapter I will discuss my own context-aware architecture system. This system is an "add-on" style which can be used for existing software systems but can also be used for new software systems. One of it's strengths lies in it's loose coupling and it's progressiveness towards context-aware systems.

2 Context-Awareness

Today's software systems have their functionality built in quite statically. This often means they are not optimally using the resources they have at hand, that they are sending non-optimized information to the user or using incorrect variations for certain variation points. By using information about the context the system will be able to perform better in specific situations. We call the latter context-aware software systems.

2.1	For starters I will discuss the definition of context. Though there are multiple definitions of context I will only discuss the two most important definitions and my own definition.
2.2	I will discuss my vision on context and context-awareness. At last I will discuss the advantages of context-awareness against conservative designed applications

2.1 Definitions of Context

In a traditional view the computer can be seen as a collection of black-box functions [1], which means that if we provide a certain input the system results the same value directly related to the input parameters. The output is determined directly from the input given. See Figure 1 (a) (adopted from [1]). So when we insert for example the arguments "1" and "2" in a function it answers "3", this is always the case as the context of the system is not influencing the result value.

To make computer systems more useful we should not only use the user interaction but we should also consider the context of the software system (Figure 1 (b) and (c)). Elements used to determine the context could be the output of sensors (temperature, infrared, numbers of cars passing a counter placed in the road etc.) i.e. a representation of the physical world, the virtual world (processor load, hard disk status, network connectivity, network load, server load etc.) and the history of all of these.

There are currently multiple definitions of context. On one side there is a definition that only the implicit input is context, in this case I will discuss the definition from Schilit, Adams and Want [2]. On the other side is a definition that 'everything is context' from Dey, Abowd and Salber [4]. In the next paragraphs I will discuss these two definitions.

2.1.1 Implicit Input is Context

In this paragraph the definition used by Schillit, Adams and Want [2] will be discussed. The definition is as follows:

> *"Three important aspects of context are: where you are, who you are with, and what resources are nearby. Context encompasses more than just the user's location, because other things of interest are also mobile and changing. Context includes lighting, noise level, network connectivity,*

communication costs, communication bandwidth, and even the social situation".

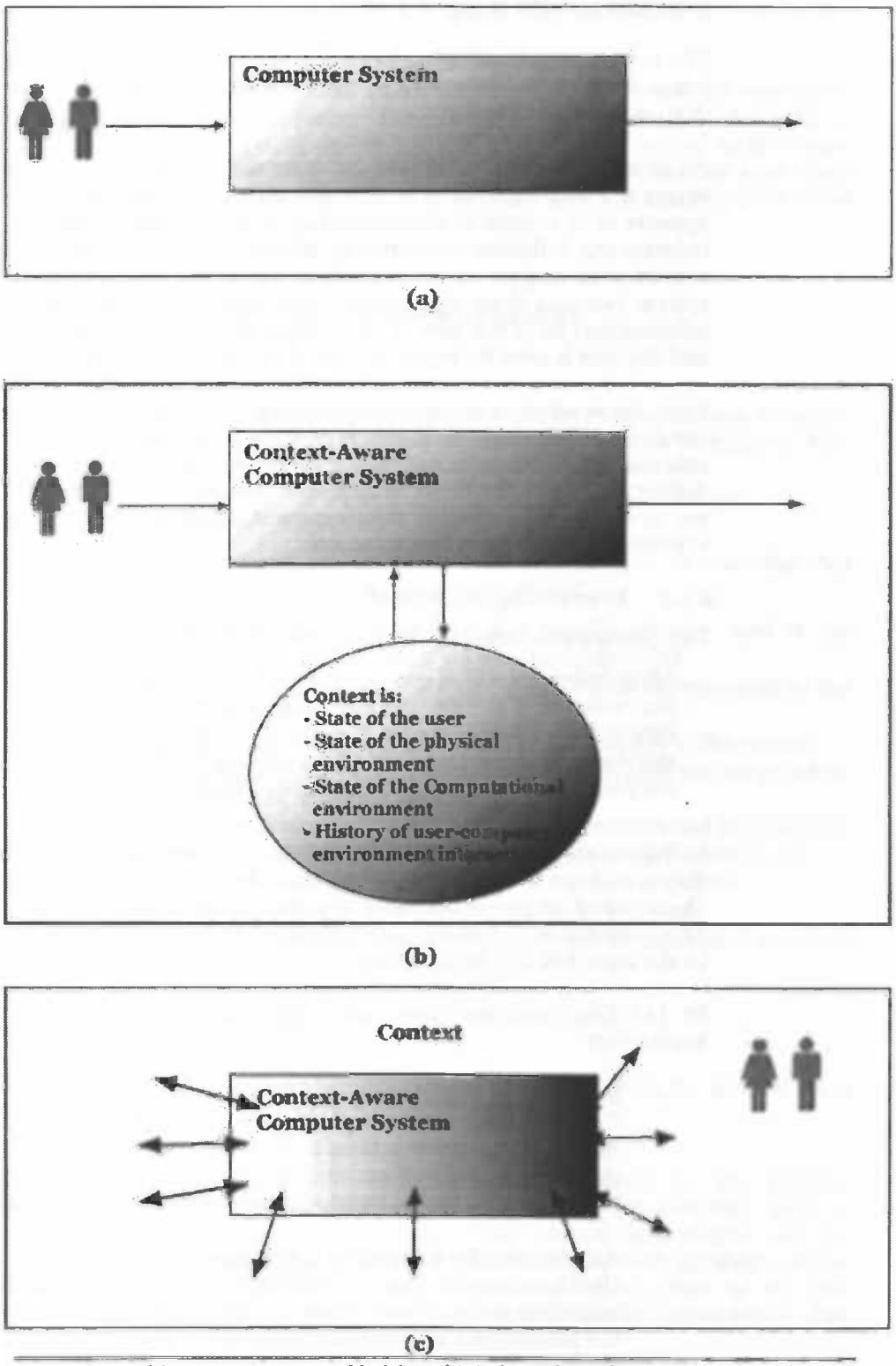

Figure 1. In (a) a system is seen as a black box, this is the traditional view on a computer system. In (b) the computer system is context-aware, this is the case where only implicit input is seen as context. (c) Shows a context-aware system at which everything (including user and the application itself) is considered as context.

There are more of these definitions like the one from, Brown, Bovey and Chan [3] where context is defined as location, identities of the people around the user, the time of day, season, temperature etc. (summary from [4]). An illustration of this definition is found in Figure 1 (b).

The definition is based on information that is not explicitly entered by the user, but is completely determined by information other than the user. We could call this information "implicit".

All these definitions are based on examples i.e. context is location, context is where the user is, context is who the user is with etc. When a new item that appears to be context is discovered (i.e. a new kind of sensor) we would have to redefine our definition of context by adding this item. We should weigh this new subject with respect to the definition. I.e. is the new information part of the system (we can draw the "system box" around the software system and the information) or is it a part of the context (the data is outside the "system box" and the data is used for improvement of the quality of the software system).

Take, for example, a sensor which opens a door when a person steps in front of the door and closes it after a short period of time. Is this context or input from the user? And what if the user waves his hand in front of the sensor to deliberately open the door? Whenever a new kind of sensor delivers new input we have to decide whether this is context. Every new kind of sensor requires arguments whether it is context or not.

2.1.2 Everything is Context

Dey, Abowd and Salber use in their research [4] the following definition:

> *"Any information that can be used to characterize the situation of the entities (i.e. whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity and state of people, groups and computational and physical objects."*

In Figure 1 (c) this definition is illustrated. Everything around the application that is relevant to the interaction between the user and the application is context. There exists interaction between the application and the context of the system. This system is not a black-box system; the system's output is not only determined by the input but also by its context.

In [4] Dey describes three ways of using contextual information in an application:

- **Presenting information and services;**
  Use context for presenting context information to the user or to show the user appropriate actions in the current context. For example, showing a user's position on the map using a car navigation system based on GPS (Global Positioning System).

- **Automatically executing a service;**
  Use the context to trigger commands or reconfigure the system according to the context changes. For example, a service keeps track of the position of the user, if it enters a super market it starts a digital shopping list.

- **Attaching context information for later retrieval;**
  Tag information with context data. For example, a notebook program which takes notes. Whenever the user writes down a note the program adds information about the context like the location, the people in the room when taking the note etc.

A huge advantage of this definition lies in the fact that context is an abstract concept; everything that may be interesting to the application is seen as the context, including the user itself.

A disadvantage of this definition is that the interaction between the user input and the context is rather blurry. The consequence is that while designing a software system the input of the user and the context are treated as the same, this leads to the impossibility of separating concerns (i.e. separate user input from the context, and separating information needed for functionality from those needed for improving quality).

2.2 My vision on Context-Awareness

2.2.1 Context

In this paragraph I will discuss the vision I have on context-awareness. Context-awareness will be an important next step towards autonomous computing. Also quality and user friendliness of software systems will be improved.

ISO 9126 defines software quality as:

> *"The totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs"*

To formalize this definition a quality model is needed. Widely used is the following model, the "ilities-model" (which is about 40 years old):

- Reliability – The way the system can be relied on (often expressed as the mean time to error);
- Modifiability – Degree of work needed to make changes to the system;
- Understandability - The ability to understand the software design when changes/ fixes need to be made to the system;
- Efficiency – The resources the system needs to perform and how well the resources are used by the software system. (Speed, use of memory etc.);
- Usability – Ease of use for the end-user of the software system;
- Testability – The ability to test the working of the software;
- Portability – The ease of porting the software system to another platform (i.e. from windows to unix).

In this thesis I will adopt the following definition of context:

> *"Context is any kind of information which can affect the quality of a software system."*

Context information is not information that is necessary for the systems functionality. The system should still be able to function correctly, with or without information about the context. The context information can be considered optional and is only used for improving the quality attributes of the software system. Because using context information has no effect on the core functionality of the software system, the system can be made context-aware after it is being build (or perhaps after the design phase).

Ordinary information entered and leaving the application is the information offered by the user or another actor (for instance a sensor) which influences, and is needed for, the core functionality of the software system. Context information may be any kind of information that can affect the quality of the software system but is not the ordinary input of the application.

Figure 2 shows three software systems. The three versions all represent a (GPS) car navigation system. In the first (Figure 2 (a)) the user enters the current location and a destination. The system calculates the route the user has to follow to arrive at the destination. This system is not a context-aware system. No extra information is used to improve the quality attributes of the system. The user input is needed for the system's functionality so it cannot be considered contextual information.

In Figure 2 (b) the same car navigation system is shown, except a part of the input is not from a user but from a GPS. The GPS enters the current position, the user enters the destination. If we removed the GPS input, will the system still function? It won't; it has no idea what its current location is. So this system is not a context-aware system. We could address this system to be "location-aware". We could also draw the "system box" around the GPS; the GPS is an essential part of the software system.

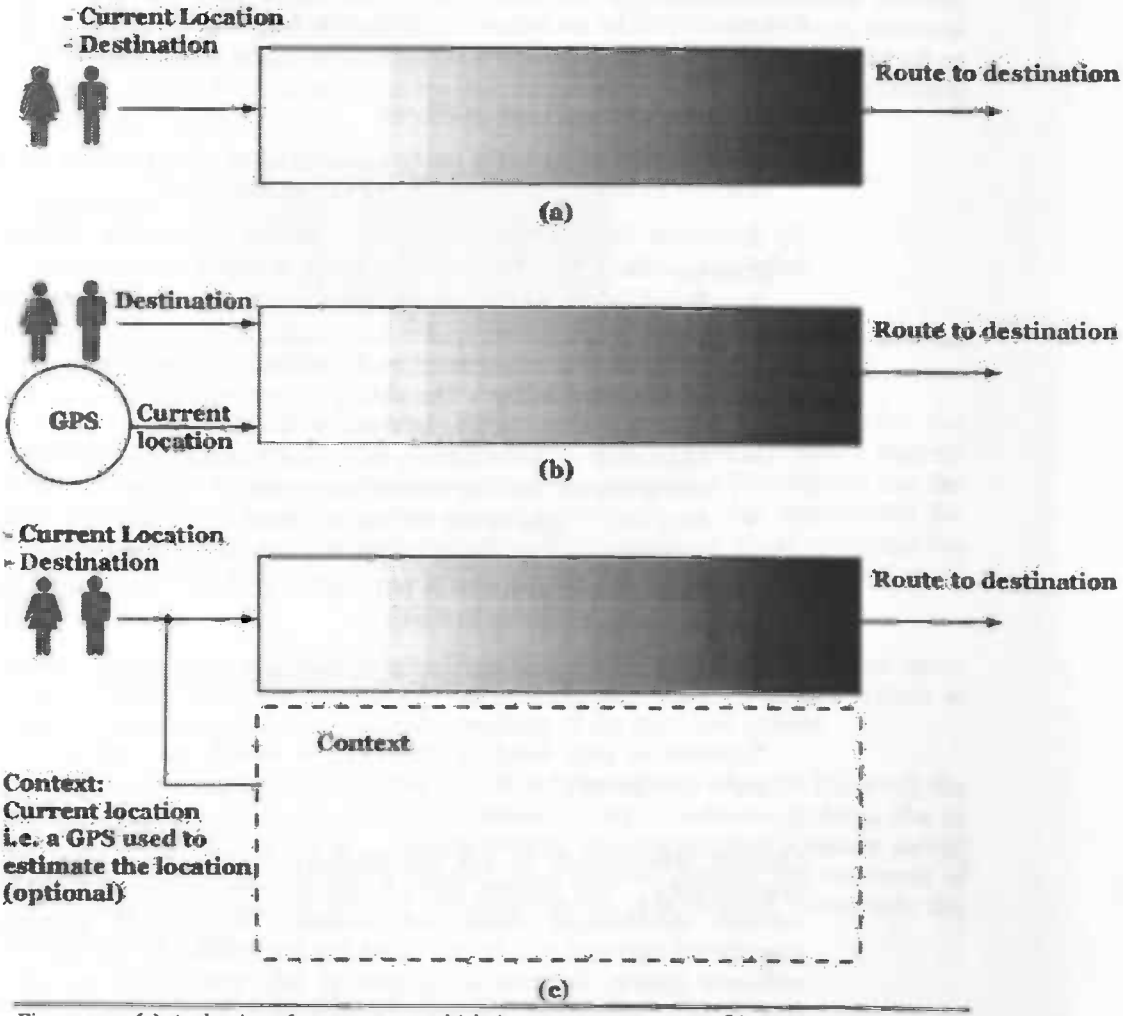

Figure 2. (a) A classic sofware system, which is not context-aware. (b) A non context-aware system, this is rather a location-aware system. (c) Is a context-aware system, the context information is optional.

Figure 2 (c) shows the GPS system but the user is able to enter the current location. In this system the input (information) can also be retrieved from the context. For example the current position can be retrieved from a GPS (but any sensor able to determine the current location can be used). Using this contextual information will improve the quality of the system; the system is able to adapt if

the user misses a turn. The GPS is not necessary for the system to meet its functional requirements; if the context information is removed the system will still function as expected (i.e. it will still fulfil the functional requirements).

Context can be any information; location, processor load, objects in a room, traffic information etc. Context is used to improve quality in the software system.

2.2.2 Context-Awareness

A context-aware application uses the context of the system to improve the quality of the system. Throughout this thesis I will define a context-aware application as follows:

> *A context-aware application is an application that adapts itself to improve the quality attributes of the system by using the context.*

So, a context-aware application uses the context to improve its quality attributes. This can be done by altering parameters in the system, behave differently etc.

Non context-aware applications are always acting the same way, not bearing in mind the context. Thus when a user presents the same input twice the application will give the same response twice. A spreadsheet application is a good example of a non-context aware application; the user enters some formulas and numbers, the spreadsheet fields always give the same result as long as the fields are filled the same way.

2.2.3 Features of Context-Aware Architectures/ Frameworks.

There are some features which we can compare the currently available architectures and frameworks on. In this paragraph I will describe these. The aim of this paper is at an architecture that is useful in a wide variety of software systems.

Separate Context Component: When the analysis of the context is integrated in the system (the intelligence is a part of the software system and mangled with other functionality) it will be harder to extend and maintain. Or perhaps the designers didn't recognize it as context handling. Also the function of the context engine is rather unclear and indicates an ad-hoc system architecture.

Distributed Context System: Not all applications are able to detect all information about the context. Often there is a higher policy than on a local standing. The system should be able to use the information from this higher policy as well as from itself.

Context improves quality: In the definition of a context-aware system the context is used for the improvement of the quality attributes of the system.

Context is not needed for the core functionality: If specific context information is needed for the core functionality of the software system it is not context-aware.

Mechanism of passing changes in the context: There are multiple ways of telling the application that the context has changed. The categories are the following:

- Polling (*P*); the application polls the context engine for changes on a regular basis, i.e. every 5 minutes, each second etc.;
- Call-back or event-based (*CB*); the context engine pushes the information about the context as soon as there is a relevant change to the context;
- Periodical push of information (*PP*); the context engine pushes it's context data in an interval to the application. I.e. every 2 minutes.
- Continuous monitoring (*CM*): The application continuously monitors the context data. Whenever a change occurs it directly acts.

Fail safety: What happens in case the context is not available? Several options are available:

- Backup components; another component which provides the same context information is used.
- Assume unchanged (last state); the last measured state of the context is used i.e. the system will assume no changes in the context.
- Context system is decoupled; the component providing a specific part of the context information is decoupled from the system, in this case the system assumes the default values for this type of information or function as if no context is available. Also the designer could decide the complete context system should be decoupled, ultimately the system can do this by itself (by self-reflection).
- Default values; in this case the complete context system is decoupled from the system. The system uses the default values or values provided by the user.

Context sensing intrinsic or extrinsic: If the context is sensed within the system (intrinsic), the application use changes in the application to fetch context information. If the context sensing is extrinsic the application uses external sensors to changes in the environment of the application (which can be virtual as well as physical). For a schematic representation of an intrinsic and an extrinsic collector see Figure 3.

Figure 3. An application with an instrinsic as well as an extrinsic sensor.

Evaluation of context information: Is there a component responsible for the evaluation of the context? This component is able to identify which context estimator (i.e. an intrinsic or extrinsic sensor that estimates the context) has the best probability of estimating the correct context. It is also responsible for noticing wrong, poor or mis-measurements. If this occurs it could exchange the measuring component, change to another context sensor that produces the same data etc.

Acting on the context information: Does the context system hold one or more actors? These actors are responsible for sending data to the software system at hand or enhancing and improving the working of the software system.

Overriding of contextual data: The user should be able to override (control) the actions based upon the data from the context. Users of software systems like to have the idea that they are in complete control. This means that a steady ascent toward context-aware systems is needed. So a parallel channel for the input of the context information is needed, the user can use this channel to override the context information.

2.3 Benefits of Context-Awareness

There are a lot of benefits to achieve by using information about the context. In this paragraph I will discuss some areas where we can gain significant improvement by using contextual information.

Better User Interface
We can improve user interfaces (UIs) by using context-awareness. For example, we could switch to a user interface dependent on the expertise of the current

user. When a beginner logs in we can hide a lot of expert and advanced functions and add a lot of helpful descriptions to the UI. On the other hand, when an expert logs in a lot of advanced and expert functions can be added to the UI; so this power user is able to do everything in its own way, for example without wizards popping up every minute to "help you out".

Another example is when you have a library index on a pocket pc with GPS. When the user walks in a certain library alley the software will only show the relevant search results for that alley. Or it could lead the user to the right alley for the book he or she is searching for.

Ubiquitous Computing
Ubiquitous computing is the term for "computer everywhere". In short this means that computers will appear everywhere and mostly unseen. Every item from a pencil to a jacket to a telephone will be equipped with a micro processor.

With these computers appearing everywhere we don't want to explicitly tell the computer what to do in a certain situation. The computer should act depending on the context and, for example, what is done in the past.

An electronical wallet (like the one described in [5]) can be used to pay a parking ticket. When parking a car there is a computer in the ground near every parking spot. When we drive the care in the spot it registers the starting time. We are able to pay using our electronical wallet, the first time this could be done by the user. The system learns from the past, in this case the context, so every time when such a situation appears it should pay to the parking lot computer automatically. Nevertheless, if the context is not available the wallet can also be operated manually.

Another example; it is getting dark outside and your child is still playing outside. You can't see your child but you are not worried. You know your child is wearing her "nanny-vest". This unique vest senses the context of the child and notices it is getting dark and directs the child to go home. If the child refuses it will send a SMS to the parents with the GPS coordinates in it. This is another example of a context-aware system. It has a couple of aspects, i.e. sensing it is getting dark, noticing no parent is around, and noticing the child refuses to go home. The vest acts upon these contextual items and makes sure the child is safe.

Autonomous Computing
When the computer system is more dependent on the context it is used in, it will be a more autonomous system. Autonomous systems are able to fully function by itself. They have the ability to make the correct decisions at the right time. Also see paragraph 3.1.

When putting this in perspective of a Traffic Light System (TLS), we could think of changing the green-red light switching algorithm dependent on the traffic at each light. Or perhaps communicate with other TLSs to organize an optimized traffic flow between the different TLSs.

An intelligent router could be an autonomous system. The router not only looks at the packets currently in it's pipeline but also looks at the throughput in the routers the packet should go to. If a certain route is very busy it is able to select another router so the busy router is omitted. In this system de packet rate on certain paths and the nearby routers are considered context.

By using intrinsic sensors the system is able to measure its inner working. With these measurements it can become a self-aware, self-healing etc system. (Also see 3.1.1).

Better Component-Based Software Design

One of the major problems in component-based software design is that the components aren't as reusable as we would like. For every new software architecture a component is placed in, it needs to be adapted to the architecture. Wouldn't it be great if the component adjusts itself depending on the context it is in?

Dynamic Software

As a last example I will use dynamic software. Dynamic software is able to adapt itself at runtime to the wishes of the user. Dynamic software would even be more powerful if it is able to adapt to its context.

For example if we have a ubiquitous computer system on our body (for example a jacket with all sorts of computers) it would be great if it is able to adapt itself to the environment. We could think about less security when you are surrounded by project members, so they are able to read project documentation. But when you are outside your office building no one should be able to read the documentation (at least not without your approval).

3 State of the Art

Context-awareness and autonomic computing are gaining interest as research topics of software engineering and big companies all over the world.

3.1	IBM has been working on autonomic computing for some decades now. In this paragraph I will discuss their vision on autonomic computing and the components of their vision.
3.2	HP has done some research on the usability of context-awareness in laptops and psychological effects it has on the users of the laptops. A short summary appears in this paragraph.
3.3	Multiple research groups are researching context-awareness of software systems and how to create a generic architecture for becoming context-aware. I will discuss three of these including some features of context-aware systems.

3.1 IBMs Autonomic Computing

With the trend of pervasive computing the complexity of computer systems is approaching the limit of the human capability to manage these systems. To overcome this problem there seems to be only one solution; Autonomic Computing.

3.1.1 What Is Autonomic Computing

Autonomic computing systems are systems who can manage themselves according to high-level goals given by their administrator. Creating autonomic systems is a very hard and challenging part of the IT. Autonomic systems are based on the working of, for example, the nervous system of the human.

The nervous system manages the breathing, heart rate, warmth etc. of the human. We don't have to think about how fast we breathe the only thing we have to worry about, is running to get into the bus on time. The same should hold for autonomic systems, we should only tell the system what to do at a high level of abstraction and the system finds out for it self how these goals can be achieved.

There are different aspects of an autonomous system, I will state them in the following part of this paragraph.

Self-management
An autonomic system should be able to manage itself, making sure the system administrator is released from a lot of work with respect to the system.

This means analysing its context etc. The system should be able to act by itself upon the change in the context.

Also the systems should make sure that, for example, the newly downloaded components are functioning well. It should do tests on itself to make this sure.

Self-configuration
Hosting providers commonly have lots of servers, routers, cables, data-entries, data-exits, websites running on the server etc. Most expert administrators are having a hard time administrating these server parks. And commonly business stopped due to a fault by the administrator who tries to install a new system.

Autonomic systems should be able to configure itself in the context it is put in. They should be aware of how they need to be configured and using high-level policy rules and the context to configure itself by.

So, when a new component is introduced and "started", it will integrate itself seamlessly with the rest of the systems and its context.

Self-optimization
Fine-tuning of a large system can cost months of work by a couple of administrators. Most of the time changing one parameter of the system affects other parameters of the system.

An autonomous system should be able to optimize its working; the administrator only needs to tell the high-level policy. This could be to reduce the cost of keeping the system running, making better use of the current resources and giving better/ faster results for the current context of the system.

Self-healing
Whenever an error or failure occurs in a system the administrator has to find the root of the error/ failure. Autonomous systems should be able to isolate failures and errors, detect the root of the problem and repair these problems. If the system is not able to directly fix the failure it will further diagnose the information at hand and eventually update by itself (see "self-management") or inform the programmer of the problem.

Self-protection
Though there are firewalls, virus scanners etc. most users forget or refuse to install these kinds of software. Autonomous systems are able to protect themselves against intrusion and attacks from viruses, hackers etc. Also they will defend the system as a whole from large-scale attacks. Intrusion and virus detection can be done by analyzing the context of the software system.

3.1.2 Context-Awareness in Automatic Computing

To become autonomous the system should be able to inform neighbour components about the information they produce and what information they consume. It will be able to act upon available resources and even be able to interact with other components based on the knowledge of the context of the system.

Context is needed at almost every "self"-item. By analysing the context (being either intrinsic or extrinsic) it will be able to become autonomous and improve its quality attributes in certain circumstances. For example in the self-healing item the system has to detect an failure or error inside the system. By analysing the context (either inside or outside the system) errors or failures will be found more quickly and the analysis of the problem can also be done more easily.

Autonomic systems also need to be able to dynamically find other components which are able to produce the data the current system wants to consume. With context-awareness the system is also able to look for other options to get certain information it needs. For example when a GPS is not working it may want to get

location information by using another component i.e. a system which is able to determine the location based on the use of a mobile phone.

3.2 HPs Context-Aware Systems

HP also has a roadmap for context-aware systems. In this paragraph I will discuss the proof-of-concept for a context-aware laptop. One of the aspects was the psychological issues when a system performs stand-alone actions based on the context. I added this case merely to show the usage of context-aware applications.

The developers made a list of predefined profiles at which the laptop could perform like "Meeting", "Home", "Presentation", "Travel" etc. Though most of these "context-aware" applications focus on saving power when travelling, the developers focussed on response and performance of the system.

At first, the context switching occurred by hand, though the users where able to change the context, almost none of them did. The next step was adding automatic context switching by using the time of day, network traffic and power as the primary sources of information.

After the automatic context switching was implemented, the users found that the notebook adapted to their needs. None of the users had said to change the context manually. This could mean two things, (1) the amount of context information used was adequate, (2) the context switches where too minor to bother the user.

3.3 Architectures and Frameworks

3.3.1 Gauges

In [6] an architecture is discussed for monitoring and adapting of software, based on the use of gauges. The architecture can be used for context-aware applications, but the architecture is not necessarily used as such. For dynamic adaptation the authors suggest three important activities.

Monitoring
Monitoring is used to measure the context of the software system. Monitoring can be intrinsic as well as extrinsic. Of course this monitoring is as free as possible so the widest variety of sensing is supported.

Interpretation
In the monitoring part data about, for example, the context is gathered. Next, the data will be analyzed; this is done by interpretation and aggregation of the data. The system is able to act upon the results from the analysis.

Reconfiguration
The analyzed data is reflected with the defined model, an evaluator measures if the gathered results are acceptable with respect to the model. An actor is responsible for adapting the system and making sure it ends in an acceptable state.

In the article they used the framework for automated system adaptation as seen in Figure 4. This is a 3-layered architecture with the lowest layer of the architecture responsible for monitoring a system's runtime properties.

The *Runtime Management component* is responsible for monitoring the system of its environment and for keeping track of changes to these items. Via the abstraction bridge the data is abstracted in data suitable for analysis at the

architecture layer. The data is "pushed", i.e. continuously monitoring, towards the upper layer when new information is observed.

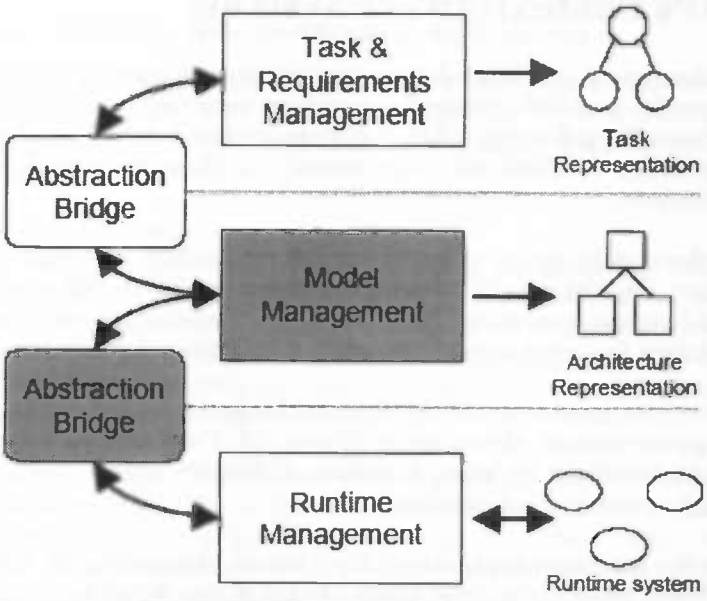

Figure 4. The 3-layered model. The model passes data upwards by using abstraction bridges. At each layer another type of abstraction is used.

The middle layer is responsible for decisions on the architectural level. It decides if data received from underlying layers is in violation with architectural design assumptions. For example bandwidth in a network is analyzed, if the bandwidth is too low for a certain component in the architecture it can decide to exchange the component (for a more efficient one, but with less fancy functionality, like statistical functionality) or look for other communication possibilities. If this layer is unable to handle the problems it is passed up to the next layer.

The top layer has a general overview of the user's tasks and handles severe (unsolvable) problems from the lower layers.

This architecture is very general and can be used in any sort of setting. The authors made a choice on a certain style. They implemented the abstraction bridge from the system layer to the architectural layer by using probes and gauges.

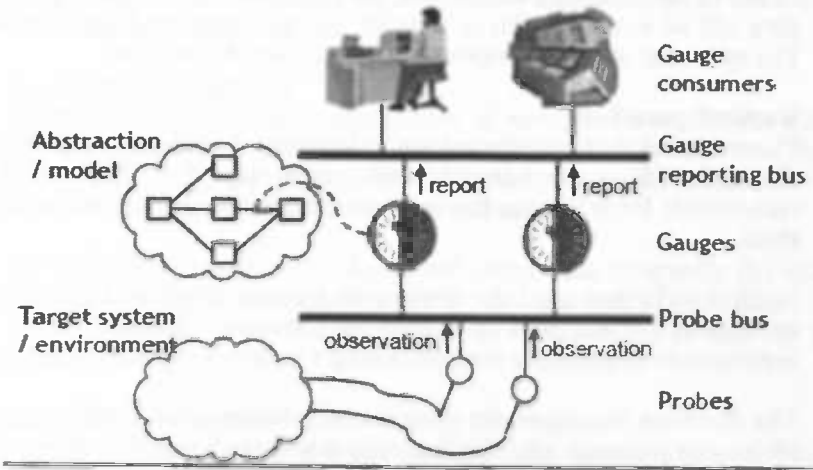

Figure 5. Gauges infrastructure. At the bottom we see the probes and their probe bus, then we see the gauges and their gauge bus.

In Figure 5 the gauge infrastructure is shown. A couple of "probes" are released in the target system or the physical environment. The system has the following components:

Probes
These probes are deployed in the target system and the physical world. The probes are able to measure data and report these data via the "probe-bus" for other components interested in the data.

Gauges
The gauges read the data from the probe-bus. Via the "probe-bus" the data is presented to other components. They interpret the data from the probes into a higher level syntax.

The key features of the gauges are:
- Gauges are decoupled from the system. The gauges aren't a part of the system so they won't affect the performance of the system. It is possible to deploy the probes in a distributed manner;
- Gauges can be mixed and matched. All the data of the different probes and gauges can be intersected, mixed etc.;
- Gauges are isolated from lower-level transport mechanisms, enabling gauges to be deployed over both RPC-based channels and over publish-subscribe mechanisms;
- Gauges can be incorporated into architectural descriptions, enabling automatic generation and execution of gauges.

Gauge consumers
Gauge consumers are consuming the data placed by the gauges. The information can be used, for example, to update models, adapt the software, alert system users, report errors etc. Items at the "model management" component from Figure 4 are components that will consume this data.

Conclusion
The described architecture is a form of pipes and filters. At the bottom we see the probes, these are connected to the gauges filter via a pipe (*probe bus*). These gauges are connected to the gauge consumers via the *gauge bus*.

It seems the user is not able to override the context. At least no component seems responsible. Of course the designer using this system is able to build-in such functionality, for example in the gauge consumer.

The context engine is quite general and can be used in a wide variety of settings. But nothing is said about the fail-safety of the software system. The information about the context send by the probes could be out-dated, a probe could be broken resulting in poor, none or even worse; wrong measurements.

This is not a distributed context system, the probe data could be fetched distributed but the system is not able to use information gathered at a higher standing or using a policy from this higher standing system.

The system has evaluators responsible for reflecting the gathered data with a pre-defined model. This model could hold upper and lower bounds the data should iterate between. When data is measured that is not in the optimal range the evaluators triggers an actor. But these evaluators have very narrow functionality, they are not able to determine which probe sends the best information (perhaps useful information deduced from the data from other sensors) nor are they able to detect wrong, poor sensors.

Nothing is said about the need of the context system for the functioning of the system, will it still function even without the context system? This also results in

the possibility that the context information is needed for the core functionality of the application resulting in not a context-aware system but rather in a specific information aware system.

3.3.2 Widgets, Interpretors and Aggregators

In [8] a framework for building context-aware applications is presented. The framework consists of "Context widgets", "Context Interpreters", "Context aggregators" and "Context-aware services", which I will discuss in this paragraph.

Context Widgets
Dey et. al. in [8] suggests using widgets for gathering and interpretation of context data. These widgets are based on the same idea as the widgets used for user interfaces. User Interface (UI) widgets are components which are usable in a divers set of applications. They provide the programmer with relevant data of the user's actions. It does not matter if the user uses the keyboard, mouse or something else as an input device.

By using widgets for context data the programmer is able to focus on the application itself. The widget provides relevant data; this means that not every change in the context is communicated but only the relevant. For example, when a user moves an inch on his chair this is not relevant for a system tracking people in an office building, but the user moving from one room to another is.

As with UI-widgets, context-widgets will need to be integrated into the application. This means the complete application needs redesign based on the use of context widgets. There is no separate context component the components will be integrated throughout the system.

The widget use an event-based mechanism to inform the information about changes in the context.

Context Interpreters
Interpreters are used to make the data from the widgets more abstract. This can be done by conversion of the context information, conversion can be simple and complex. Simple conversion could be changing the location from a street name to a city name. Complex conversion could be integrating multiple sensors, for example (from the article [8]) using a person counter, volume sensor, location sensor to decide if a meeting is taking place.

The context interpreters are not specifically created and here for it should be able to use these in other applications as well. But as we know at this time from the component- and service-based era this will not happen until there are well defined standards.

Context Aggregation
Aggregators support the aggregation of multiple sensors. The analysis of the context sometimes requires aggregation of multiple sensors readings. The aggregators are facilitating this functionality; they are able to combine information from multiple sensors about one object.

To build an aggregator the developer simply has to specify the entity type and identity. The type refers to the entity the aggregator represents, i.e. a person, location, building etc. The identity is for example the person the aggregator is following.

Context-Aware Services
A context-aware service is a sort of widget; the service is responsible for a certain output (instead of an input like the context widget is). It is responsible for changing something in the state of the environment of the application according

to the context of the application. These can be considered the actors of the engine.

Conclusion
When the measurement of the context fails there is no back-up mechanism to overcome this failure. We will not know what happens when the context measurements fail leaving the system in an undetermined (incorrect) state. This may result in a change for the worse with respect to the quality of the system i.e. the system uses wrong, outdated context information.

There is no evaluator to check if the information about the context is correct with respect to the running system. So the programmer has to decide which context widget to use. If there are two or more widgets providing the same information we don't know which is best or if the data is even useful according to the other context information.

In this case the context is merely used for the functionality of the system, because the context items of this article will be integrated in the application. So we can't say if the context information is used for the improvement of the quality attributes of the software system, the core functionality of the system etc.

The widgets may be either intrinsic of extrinsic.

This system is very tightly coupled to the software system at hand. This results in the system not being able to progressively ascent to a context-aware system, the widget will be seeded everywhere in the software system.

There is no built in functionality for the user to override the information fetched about the context. So the programmer has to build this functionality by itself.

3.3.3 Context Awareness Subsystem

In [7] an "object-oriented, feature-based architecture for Context awareness subsystems (CAS)" is introduced. The system is part of an adaptive mobile learning environment. The MLEARN application is able to adapt on the user's motivation, location, device etc.

The architecture of the CAS is shown in Figure 6. The system contains of a device (which can be a mobile phone, a desktop computer, a PDA etc.), a context engine and a content delivery subsystem.

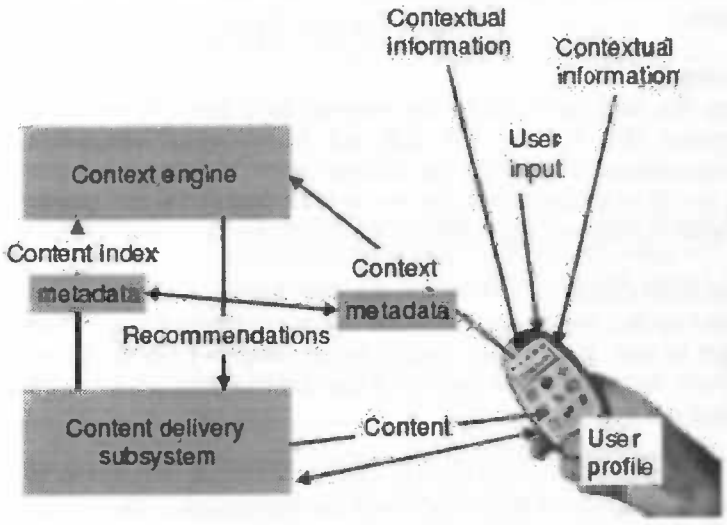

Figure 6. The system overview of the CAS architecture. The device communicates with the context engine and the content delivery subsystem to get the best result.

The mobile phone sends data about the context (i.e. the measurements of sensors, the user's state etc.) to the context engine. The context engine evaluates with use of an evaluator the context and reflects it with the data from the content engine. The context engine has an actor which creates a list of content recommendations to the content engine. This list recommends which content can best be send to the current user in the current context.

In the article the cycle of operations for the context aware system is as follows:

1. Input – of context data
2. Construction – of the context
3. Exclusion – of unsuitable content according to the context
4. Ranking – of remaining content according to the context
5. Output – of ranked list of content according to the context

The architecture in this article is based on two blackboard architectures both are separate from each other. First there is the context engine. This system receives data from the client about the context of the user. The data is from a non-distributed system, only the client itself sends information about the context.

The next step is filtering of relevant data. The engine matches the context data to metadata retrieved from the content subsystem. This system is a blackboard system in its turn. After matching the context to the content engines metadata the context aware subsystem hands out a sorted list of relevant items.

Conclusions

This is a typical system in which the context helps improve the quality of the system and is not necessary for the core functionality of the system. If the context engine is decoupled from the content engine it still delivers the correct information to the user only it will be not as optimized as we may want to (showing a full webpage on a PDA for instance).

There is no built in fail safety, when an error occurs we cannot determine what will happen in the system. Will it use the latest context state? Or perhaps it will use default values? Will it search for other components which are able to provide information a broken sensor cannot provide?

The context engine does not provide functionality for the user to override the information about the context. If this is necessary it needs to be built at the client side of the application.

3.3.4 Conclusions and a comparison

Using the features from 2.2.3 I will make an overview of the discussed architectures.

	Gauges	Widgets	CAS
Seperate Context Engine	+/-	-	+
Distributed Context Engine	-	-	-
Context improves Quality	+/-	-	+
Context is not needed for the core functionality	+/-	-	+
Mechanisms of passing changes in the context	CM	CB	...
Fail safety	-	-	-
Context sensing	IN/EX	IN/EX	EX
Evaluation of context information	-	-	+
Acting on the context information (separate actors)	+	+	+
Overriding of contextual data	-	-	-

Table 1. This table shows the architectures against the features discussed earlier in this chapter. CM = Continuous monitoring, CB = Call-back, IN = intrinsic, EX = extrinsic

Although these architectures are merely ad-hoc designed and very domain- and application- specific most of the architectures and concepts have some abstract requirements in common. With this in mind in the next part of this paper I will try to create an architecture that is able to be used for creating context-aware applications. First I will discuss current architectural styles widely used in software system design.

4 Architectural Styles

To get an idea of the style, we will need to have some basic knowledge in the current widely used architectural styles. A thorough analysis of these styles is beyond the scope of this thesis, so in this chapter I will only briefly describe the styles.

The overview is as follows:

4.1	The pipes and filters style is discussed
4.2	Here the object oriented organization style is described
4.3	The event-based or implicit invocation style is discussed
4.4	In this paragraph the layered system style is introduced
4.5	The repository style is also a widely used style which will be briefly introduced in this thesis.
4.6	The client-server architecture often used on the internet and in network-based application is discussed in this chapter
4.7	In this paragraph I will discuss why we need a composite architectural style to fulfil the requirements of the context-aware architecture.

4.1 Pipes and Filters

In the pipes and filters style each component is responsible for a certain data manipulation. The components (filters) are connected using connectors (pipes). These pipes can be typed (for example ASCII pipes, binary pipes, object streams, XML pipes etc.), bounded (i.e. maximal bandwidth) etc. The pipes don't manipulate the data; they are only responsible for the transport of the data.

Filters have an input and an output; the data that is consumed (from the input) is manipulated and produced at the output. The filter formalizes the data on the input and output side. Filters don't have knowledge about the filter which produces the input or the filter that consumes the output the filter delivers. The pipes and filters architecture is shown in Figure 7.

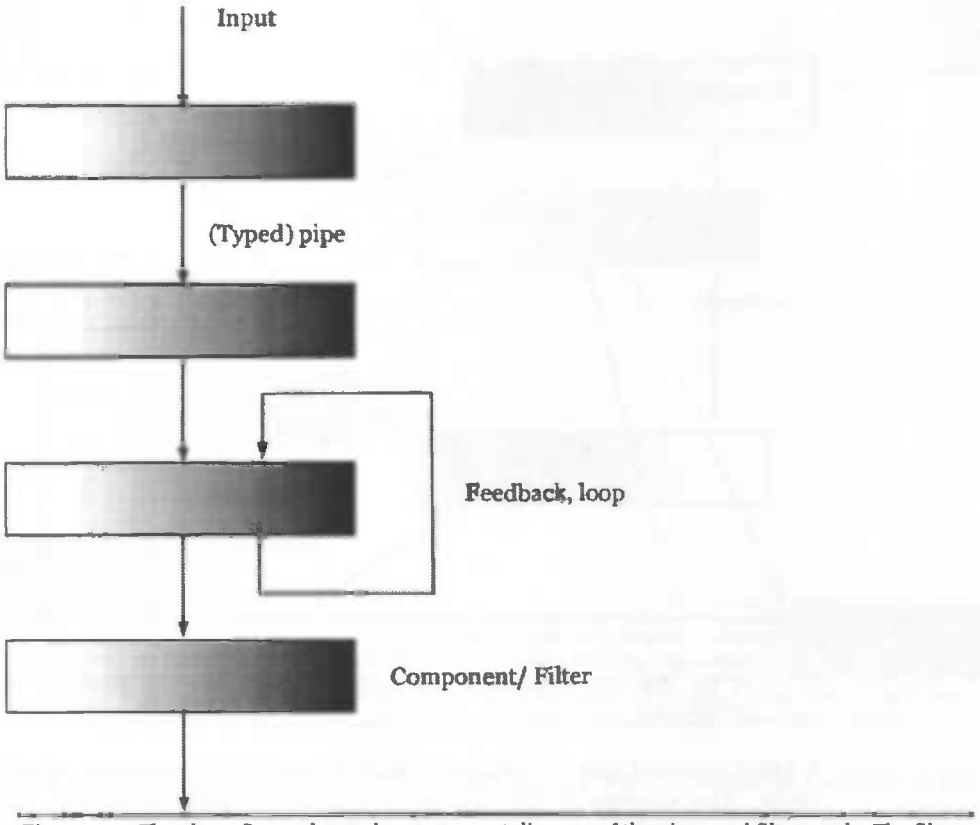

Figure 7. The above figure shows the component diagram of the pipes and filters style. The filters define input and output requirements and the (typed, bounded) pipes transport the data from one filter towards the other.

This architecture has a couple of advantages:

- Filters can be added and replaced by newer, improved versions;
- Input and output can simply be analyzed by the composition of the individual behaviours of the filters;
- Filters can be randomly linked together if they agree on their in- and output parameters;
- Analyses about deadlocks and throughput are easily defined;
- Filters can work concurrently in a threaded system or as separated processes.

This architecture also has some disadvantages:

- Pipes and filters are primarily for batch alike applications. Interactive applications are hard to design using this style;
- With using the pipes data is often serialized and de-serialized in the filters, this is unnecessary overhead;
- Filters may cause failures (which are then passed to other filters)

4.2 Data abstraction and Object oriented organization

The data abstraction and object oriented organization style consists of objects and the abstract data types (ADT) these objects initiate. These elements represent data elements and methods to handle these data elements. The objects are called the managers of the ADTs. The managers have the responsibility to (1) keep the data consistent according to an invariant and (2) to keep the data hidden from the other objects.

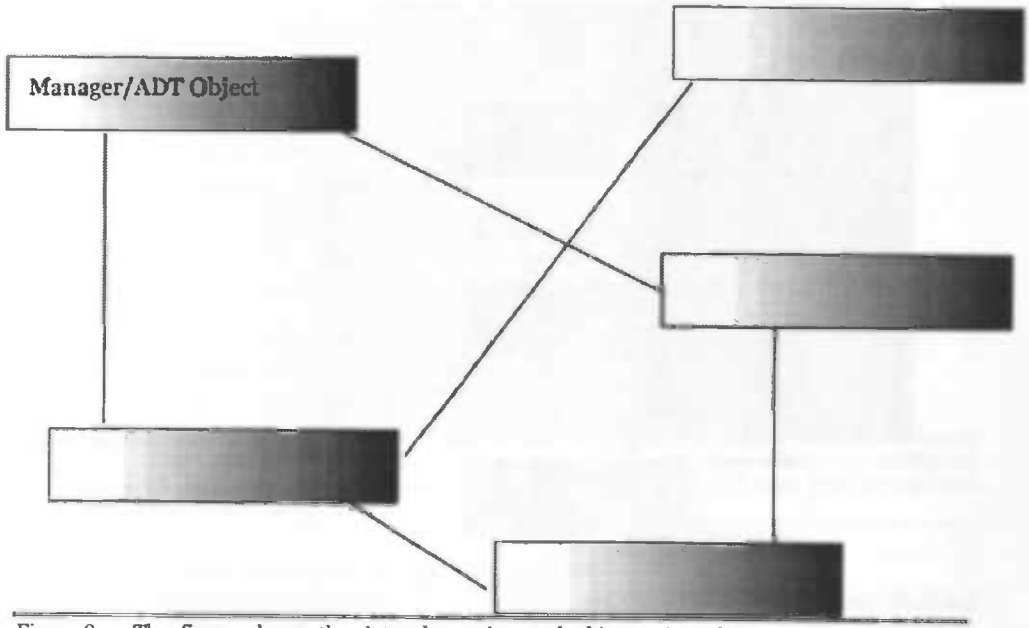

Figure 8. The figure shows the data abstraction and object oriented organization style. The elements represent objects or data abstractions, the elements are responsible for the integrity of the data they represent. The connectors represent the procedure calls or method invocations.

Object oriented system design is a well-known and widespread used architectural style. The objects could represent real world entity systems which are, for example, made persistent using a database system. There are even object oriented compilers [16].

There are very much well-known advantages of object-oriented system including:

- Data hiding; objects can be changed with other ones without affecting the functionality of the system as long as the interface stays the same.
- The ability to build components.
- Real world objects can often be mapped to objects in this style making it easier to discuss about the meaning of objects.

As with every style there are a couple of disadvantages:

- In contrast to the pipes and filters style the objects in this style need to know which objects represent the data it needs.
- Sometimes errors or side effects are rather vague. If object A invokes object B and object C also invokes object B the changes in object B could be mistaken for side effects of the invocations from object A (resulting in a bug report which isn't grounded).

4.3 Event-Based, Implicit Invocation

Event-based or implicit invocation is a loosely coupled style. Objects register a function to an event broadcaster or event dispatcher. The event broadcaster calls all these methods without knowing who the receiver is or what the receiver will do when the event occurs. So instead of knowing the identity of an object the object registers itself for an event. See Figure 9.

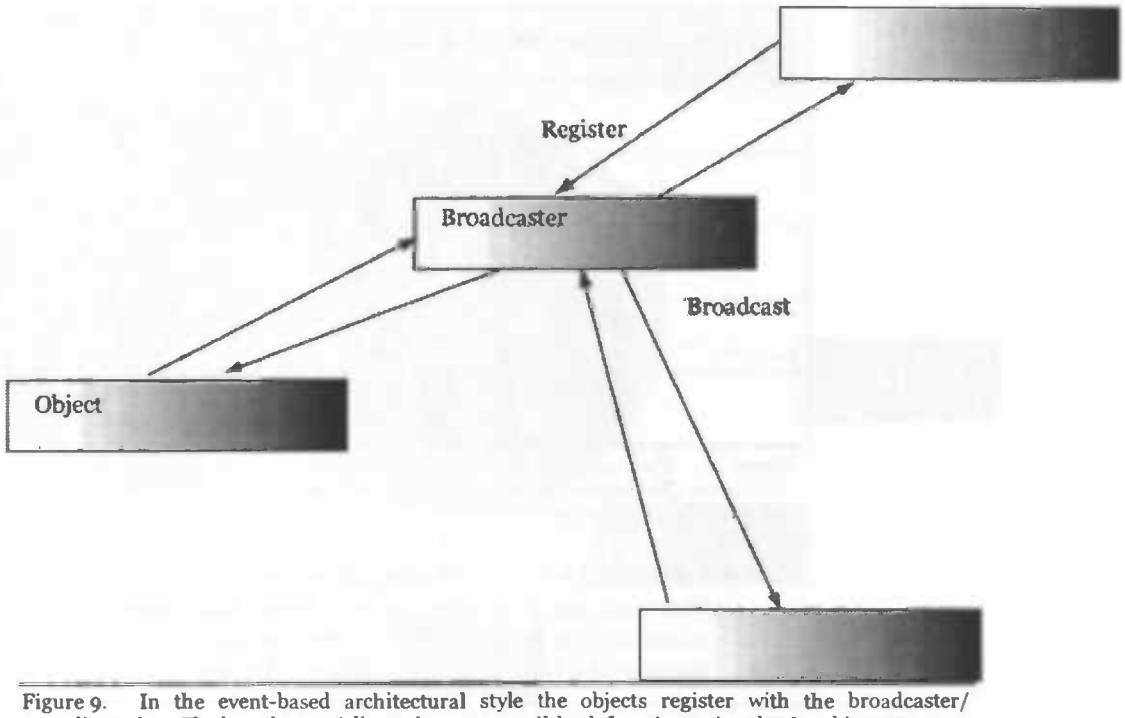

Figure 9. In the event-based architectural style the objects register with the broadcaster/ dispatcher. The broadcaster/ dispatcher uses a call-back function to invoke the objects.

This style is often used in system where modules are dynamically added to the system, in debuggers and user interfaces (model, view and controller).

The advantages of this style are:

- Dynamically adding plug-ins or modules;
- System is expandable by using these plug-ins or modules;
- Components are easily re-used by registering to different systems;
- Evolution of the system is improved; components/objects can be changed without having to change the caller.

The style also has some disadvantages:

- The event broadcaster has no idea of the consequences of broadcasting an event;
- Multiple of the broadcasted objects may need a shared repository, this means there is great risk of faults in the management of resources, good resource management is needed;
- It will be hard to deduce the correctness of the system, at run-time elements could be added, the handling of the events is unknown etc.

4.4 Layered Systems

The layered systems style uses layers to make abstractions at each layer. The layer consists of multiple elements which perform as a client to the lower layer and a producer to the upper layer. The data between the layers is passed using method invocation (for example). Most methods are only visible to the direct upper layer, but some methods are carefully exported to layers above this layer. This is merely for a more efficient approach to a solution (excluding overhead from needless layers). The style is shown in Figure 10.

The layered style is often used in network protocols (ISO OSI model, the TCP/IP protocol), operating systems, database systems etc.

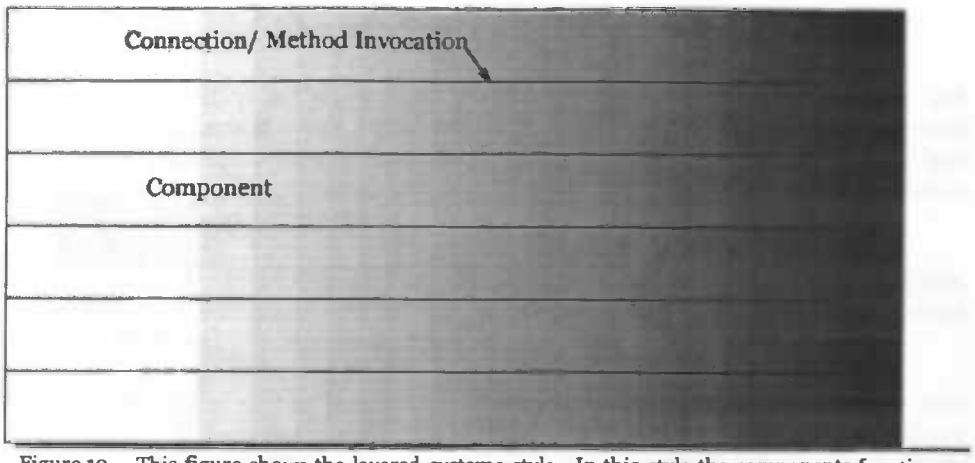

Figure 10. This figure shows the layered systems style. In this style the components function as layers. The bottom layer has the lowest abstraction building up at each layer with the top layer being the most abstracted layer.

The layered system has the following advantages:

- Problems are abstracted in each layer, this means problems are divided in small parts which are better analysable;
- Because the interface of the layers is known, layers can be exchanged;

There are some disadvantages to this style:

- Not every solution can be designed using a layered approach;
- It is hard to decide which abstraction takes place at each layer, often layers are skipped and a lower layer is used;
- The lower layers create an overhead to the functionality we want to use; this makes it hard to design a high performance system.

4.5 Repositories

The black-board or repository style consists of two important elements. The blackboard or repository, which holds the data of the problem which is analyzed, and the components which perform computations on the data. The strength of this style lies in the fact that every component is able to find a solution for a part of the main problem and puts the result back into the repository. With the result from a certain component, another component is able to transform this into another solution which could be an input for another component. An overview of this style is shown in Figure 11.

Though in theory it is possible that multiple components handle the same problem often this is not the case. Many times only one component handles one particular problem.

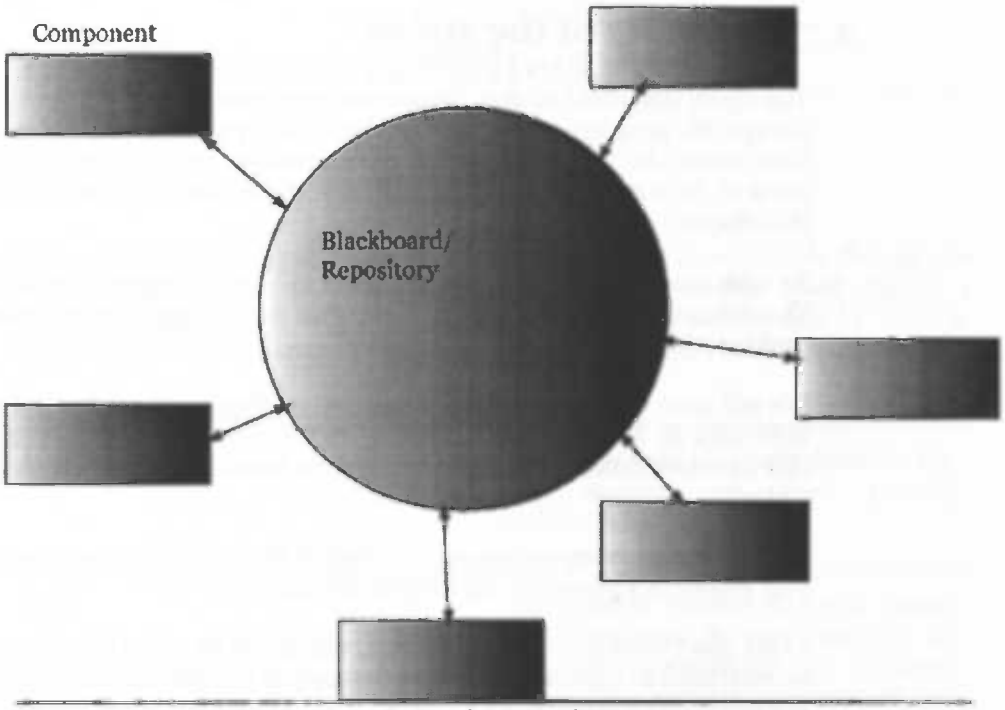

Figure 11. The Blackboard or repository style; this style represents a central data source and components which handle the data to come to a solution for a problem. Mostly each component is specialized in handling a specific problem and puts the solution back into the black-box. The collective of components find an overall solution to the problem.

This style is frequently used in Integrated Development Environments (IDE) where the programming code is the central repository and the elements of the IDE perform handling on this data. Modern compilers are also partially based on this style with the Abstract Data Trees as a repository.

4.6 Client Server Architectures

In the Client-Server architecture there are two main components, the server and the client connecting to this server. The server foresees the clients with some functionality. One server can serve multiple clients.

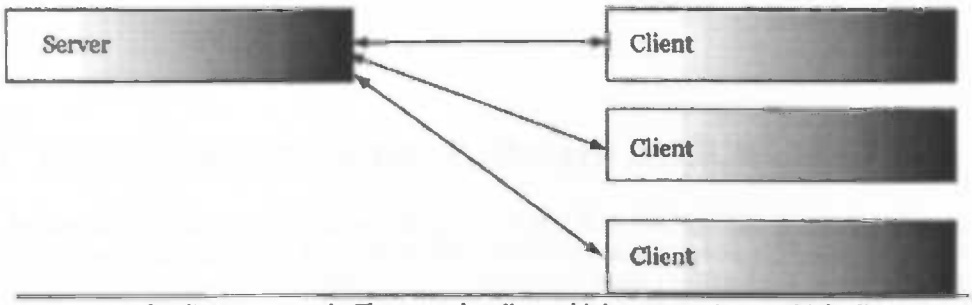

Figure 12. The client-server style. The server handles multiple requests from multiple clients.

The client-server is often used when complex business logic is needed and the client need has to be light weight. Surfing the web is an example of this style there are a lot of client requesting web pages from (web) servers. This style is also often used in business applications where there is a central database (the server) handling all data and clients requesting the data from the database server.

4.7 Usability of the styles

The styles described in this chapter are very general. For our purpose they lack the specific properties which are needed in this problem. The styles will be a solid base upon which the description of the context architecture will be based and parts of the context architecture will have styles related to the styles described in this chapter.

As with most architectures we will need to have a composite of different styles. The strength of using multiple styles is that we will have the best solution to each problem the overall problem consists of.

We will need some sort of blackboard system for all our context collectors to save their data in, but on the other hand there will be some sort of engine analyzing this data and changing the software system based on the data in this black-board system.

But the engine components can be divided into multiple components, which are best placed in a pipes and filters kind of style.

I also don't want the software system, which will be using this context system, to be restricted to a certain style. The designer of the software system will be able to use any architectural style he wants.

So, for context-awareness we will need to define our own specific architecture. This architecture should be general enough to be used in a broad range of software systems, but will need to describe the guidelines for all context-aware systems needed. The above definitions merely lack the latter requirement.

5 Context Aware Architecture

In the previous paragraph I concluded that there is no general style that can be used to create a context aware architecture. In this chapter I will discuss a composite architecture and its components. The architecture can be used in applications that are currently in use but it can also be used in new applications.

The overview of the chapter is as follows:

5.1	In this paragraph I will describe the requirements of the context architecture.
5.2	The context system is not a stand-alone system. It can function at different components at different levels. The context architecture can be used in components or in applications. In this paragraph I discuss the hierarchy of the context aware architecture
5.3	In this paragraph I will discuss the reference architecture. All the components and their responsibilities will be discussed.
5.4	In this paragraph I discuss a progressive implementation scheme to gradually bring an existing application toward an context-aware, self-aware and self-adaptive software system

5.1 Requirements of the Software Architecture

In my definition of context-awareness I state that the system must be able to adapt, as a consequence the system has to be a dynamic system. Therefore Req. 1 Req. 2 Req. 3 Req. 4 and Req. 5 are all adopted from [17] and are required for a Dynamic Software System. I will adopt the definition from this paper; a dynamic software system can be defined as:

> *"A software system where one or more of the variation points have an explicit runtime binding time."*

This means that changes to the system can still be made, even when the system is deployed and running.

The following requirements are needed for utilizing the context-awareness of the software system.

Req. 1 Possibility of choice
In principle a dynamic software system will need to have possibility of choice. If there is no choice this means the system is static and will or cannot adapt at runtime. Choices which are for instance encapsulated are changing the keyboard language settings in windows.

Req. 2 Assessment of the quality in the current choices
Dynamic software must be able to determine the performance of the current choices. This requires a specification of:

1. What needs to be measured in order to asses the quality of each separate choice;
2. How to qualify these measurements.

This qualification is used at run-time to determine the performance of the current choice.

Req. 3 Deciding for change
A mechanism is needed which, at runtime, decides a change is required. The mechanism uses the measured quality of the current choice to determine if a change in the system will increase performance and is therefore needed. The call for change results from changes in user requirements, or –in our case- a change in the context of the system etc.

To be able to estimate the urgency of changing the system it needs to know the requirements of the system. This way it will be able to reflect on itself and is able to decide for change.

Ultimately a system is self-aware meaning (From [17]):

> *"A system is self-aware when it has a built-in assessment and decision mechanism for deciding that it is not functioning properly anymore."*

Req. 4 Determine what to change
We need to be able to determine what to change. In our case we use the data from the context of the system as well as the information from Req. 1 and Req. 3 . Multiple ways of deciding what to change is needed are possible.

For instance we could try all the alternatives to see which performs best. We could build a neural network which learns from previous adaptations and therefore makes better decisions in the future. A pre-programmed decision tree could as well be an option.

Req. 5 Enacting change
A mechanism is needed to enact the change in the system. The designer has to define a strategy to enable the necessary changes. Multiple ways of adapting the system at run-time are thoroughly studied. If the system is able to determine what to change and enacts this change, the system is called self-adaptive.

Req. 6 Measuring context
The context system has to be able to measure the context. There are two ways of measuring the context:

1. Intrinsic – The data is gathered internally only. This means only local data directly available inside the software system is used.
2. Extrinsic – The data is gathered outside the current software system. This could be globally available data, measurement from collectors outside the system etc.

Req. 7 Saving the context information
The context engine must be able to save data of a period of time. This data can be used for analysing changes in the context, the history of the context etc. The saved context information will be available for other components as well.

Saving of this information not necessarily means a database but could also be a pipe or variable in memory.

Req. 8 Analyse context data
Analysing the data is required to estimate the necessity of adapting variation points in the system. The system should be able to save the estimated context for later use and further analyses.

Req. 9 Progressive scheme towards self-adapting system
The context system can be incorporated gradually so the system will flow towards a self-adapting system. The following phases are necessarily:

1. User controls the variation points. (The system's working is changed by the user).
2. User and context-aware system control the variations. At first the user will be in charge gradually flowing towards full responsibility for the context-aware system with the user being able to interfere.
3. Context-aware system completely controls the variations. The ultimate goal is the self-aware and self-adapting system where no user interaction is needed.

For new applications phase 2 or 3 can be directly implemented at design time. For existing software systems the system first has to be made a dynamic software system with the user being able to adjust the system. Then the system should support a gradually increase in context-awareness by shifting towards the 3rd step.

Req. 10 Loose coupling
The context system should be loosely coupled to the software system at hand. This results in the ability to separate the context handling from the software system which eases handling of changing requirements in both the systems.

Also loose coupling results in the ability to design both the components separately.

5.2 Hierarchy of the Context Aware System

Software systems and components often have a hierarchy. For our context-aware systems we can choose from multiple hierarchies: the flat hierarchy where each component has the same level of authority or a tree-based structure.

The flat hierarchy is not suitable for our purpose; it does not allow us to communicate a higher level policy towards other components. This results in a system where each component behaves in it's own interest. If each component only has a scope narrowed to itself, they are most likely to conflict on their policy level. This results in the overall software system performing much worse then with a central component which defines the higher level policy.

In Figure 13 the hierarchy of a context-aware system is shown. This is not an architecture by definition rather it is an hierarchy of context-aware components in a context-aware system. The relations between the components are still free of choice, they could be in a black-board system for example.

There exists one overall context-aware component, in this figure drawn at the very top. This context-aware component has the best overall knowledge about

the system at hand. So, the component is responsible for the decision on a more abstract and overall level than the component beneath this root component.

The root component can pass through evaluation guidance to leafs underneath it; this way adjusting the adaption and function of lower level context-aware components.

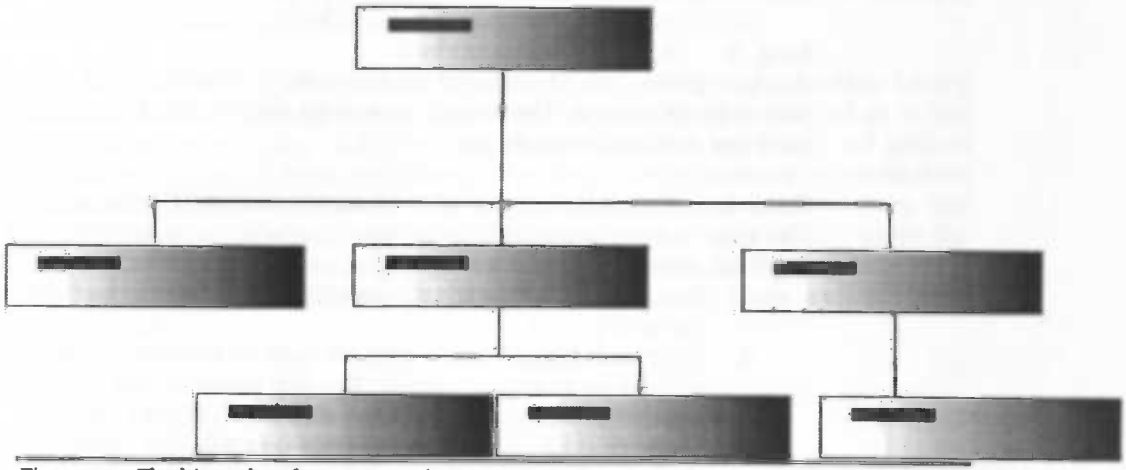

Figure 13. The hierarchy of components in a context-aware system. The top component holds the overall context-awareness.

The root component also has knowledge of all the context-aware components in the system. This is a result of, for example, letting all the context-aware components register at (1) the root node, or letting all the components register at their (2) parent node, giving the root or parent node the ability to pass through higher level policy to child nodes.

The root or parent node also fetches information of the child nodes about the context information they posses and are able to share, this way building a fair amount of context information in a database. Other child nodes are able to ask the parent nodes for certain information and where to get it. If the parent doesn't know where to find the information, the request is transparently passed through to the parent of the parent node which performs the same handling. The component then registers at the component responsible for fetching the kind of context information. The component will be informed about the context information by the producing component from now on.

A context-aware component can exist from multiple context-aware and non-context-aware components. On the other hand a non-context-aware component can not exist of any components that are context-aware, due to the fact that this results in a part of the component adjusting itself according to the context i.e. being a context-aware component.

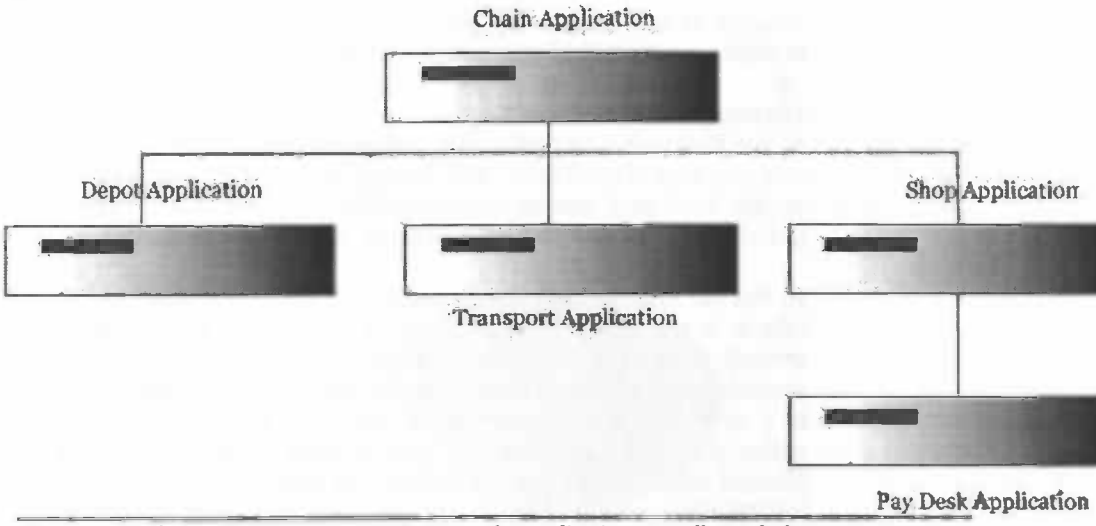

Figure 14. A hierarchy of a commercial chain. The applications are all stand-alone context-aware applications. Though they are able to retrieve context information from higher level applications.

The components in Figure 13 can also be seen as applications. The application could, for example, cooperate in a commercial chain (see Figure 14) where the applications range from storage to administration to pay desk applications. There exists one general context-aware application which has the general overview of the applications. The root application is able to adjust the higher level policy (at chain level) but the shop application is able to adjust the local policy and the pay desk application can have its own policy.

5.3 Context-Aware System Architecture

In this chapter the context-aware system architecture will be discussed. The architecture is responsible for handling context data, evaluate the data and adjust the software system to improve the quality of the software system.

In Figure 15 my first context architecture is shown. This system is very tightly coupled with the software system. It is as if it is embedded in the software system causing a tightly coupled software system. The final architecture is shown in Figure 16.

The responsibilities of the components in the first architecture are the following:

Probes
The probes are responsible for measuring context data inside or outside the application. They are also in charge of passing this data to the context data.

The "probes" suggest real-world measurement devices, but these probes can be intrinsic sensors as well as extrinsic and intelligent as well as raw. In the final architecture therefore the term "collectors" is used. These can be intrinsic and extrinsic, but they are not distributed; later we will see why.

Context data
The "context data" component is responsible for saving data about the context. In this version context data and abstracted context data (i.e. instead of the time the time of day, noon, night etc., is saved) is all saved in one repository. In the final version a distinction is made and a coupling with the engine is created (feedback).

In the first architecture two context data components were necessary, one at the local level and one at the overall level. As we will see in the final architecture this

concept is still used only the external context data sources can be from any component instead of just one overall (root) component.

Interpreter

In the first two interpreters are designed, one at the local level of the application and one at a overall level. This means information can be interpreted twice (one at the local and one at the overall level). Also this means there are multiple policies, at the local and general level, which have to cooperate.

In the old architecture the responsibility of each component seems rather blurry (there is no adequate separation of concern in the components), what is the overall interpreter doing and what is done at the local level? Are all probes available at the overall level, even the probes that measure security sensitive data in a surrounding environment? In the final architecture the data collectors the collectors is only available at the local level in the current data source, while the filtered data (the estimated context) is made publicly available using a context information source (which can be imported and export from and to other components).

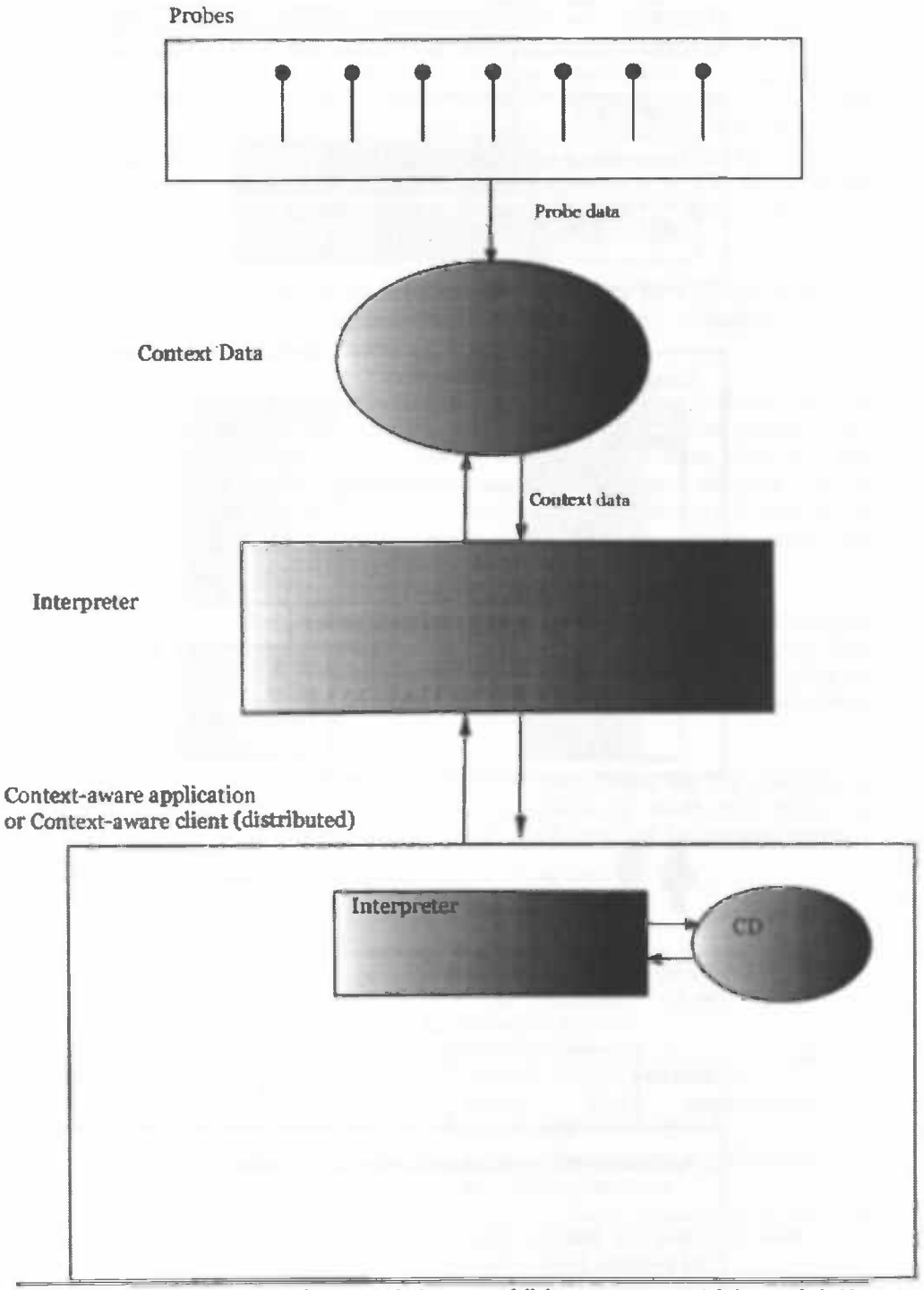

Figure 15. My first context architecture. It is not usefull because it is to tightly coupled. No distribution of the context measurements is possible and the architecture is to general.

In Figure 16 an application is shown which is extended with the context aware system. In this paragraph I will discuss the responsibility, connections and identities of the different components.

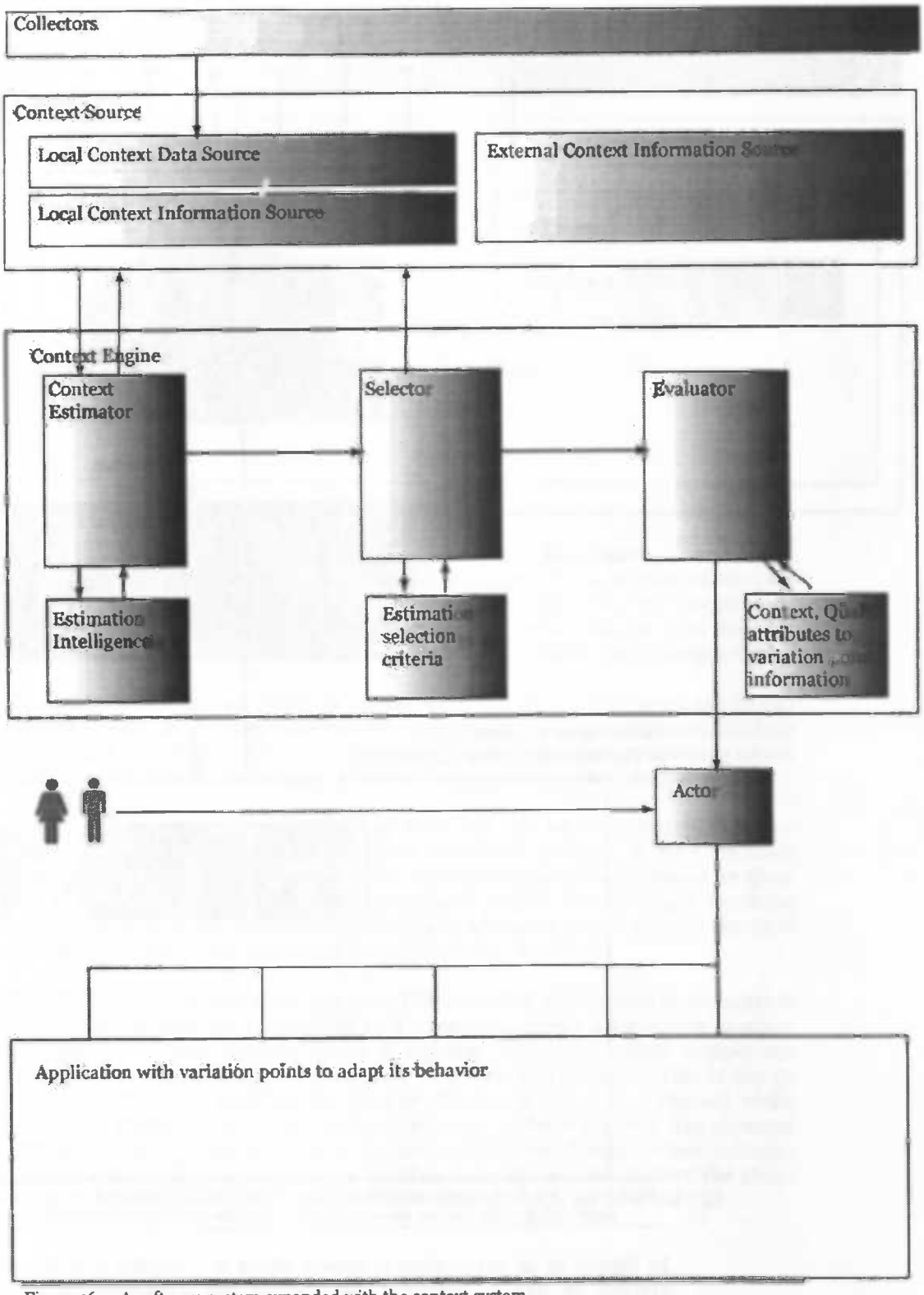

Figure 16. A software system expanded with the context system.

5.3.1 Collectors

Collectors are responsible for fetching data about the context of the software system in this way measuring the context. This can be any sort of information (as all information can be considered relevant for the context). The data can be collected via intrinsic, extrinsic (also see the feature list in 2.2.3) or local collectors.

Looking at these types of collectors one would suspect a distributed collector as well. During my research I also thought this was needed, but it seems they are not; the context information retrieved from these (the collectors which are not local, intrinsic or extrinsic) will be stored in context information sources of other components (or applications). The other component uses these collectors to gather context data (placed in its local context data source) and analyse this and place it in its context information source. The component is able to subscribe to these sources and this way retrieve the information as we will see in 5.3.2. This mechanism makes sure the context data is never analysed twice.

Intrinsic collectors determine the context of a software system by analyzing information in the software system itself, where extrinsic collectors analyze information that is globally available.

Two types of collectors can be defined; raw and intelligent. Raw collectors do not filter or alter the data in any way and place the raw data into context data sources. By using raw filters the context system can, in a later stage (i.e. the context estimator) determine which information is valuable and which is not. As a consequence of sending all data more bandwidth and memory is needed. A larger burden lies within the context engine which needs to perform more data processing.

Intelligent collectors filter or alter the data read. Intelligent collector produce less data, this can be useful in system with limited bandwidth, memory etc. The context engine has to do less filtering on the data. A disadvantage of choosing for intelligent collectors lies in the fact that (valuable) information can be lost due to filtering.

When sending the measured data to the context data source the collector has to provide additional data. The identity of the collector itself, the time the measurement took place are some, other additional data will be implementation specific.

Summary

Responsibility	The responsibility of the collector is to gather data about the context of the software system.
Connections	• Outside world (extrinsic). • Inside the application (intrinsic). • Context sources, saving data to the context sources.
Future research	• What should the interface with the Context sources be? • How can we dynamically add and remove collectors?

5.3.2 Context Sources

The context sources can be seen as blackboard systems. The collectors provide data for the context sources to fill up. The engine analyzes the data and in certain situations adds data to the context source. The context source with all its components is shown in Figure 17.

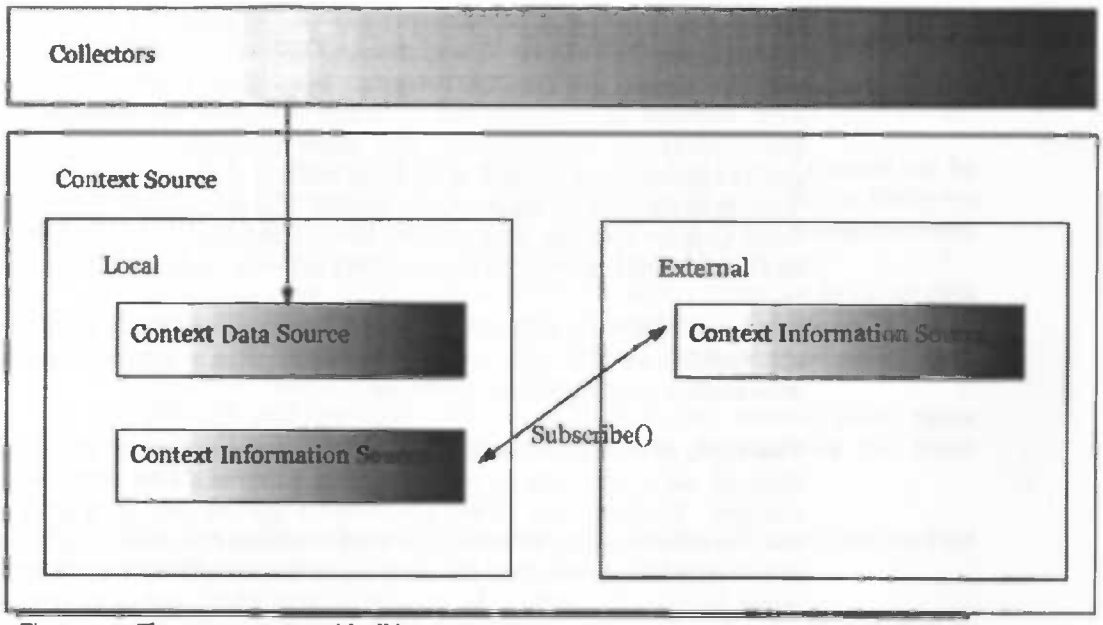

Figure 17. The context source with all its components.

The context source exists of a local and external part. The local part on its turn consists of a context data source and a context information source. The external context source only comprises of a context information source. The external system probably has a context data source but it is not published for external use.

The context data source (CDS) is mostly used by the context estimator in the context engine and by the collectors. The collectors add their measurements into the context data source. This information is read by the context estimator which adds their "abstracted" information into the context information source (CIS).

The CDS is only locally available and thus can not be imported from other components nor can it be exported. This is because the data in the CDS from other components is analyzed by other context estimators and placed in their CIS. It would be clumsy to let every component analyse the same data; resulting in a dozen of different analysis. So only the local engine is able to read the CDS and the information placed in the CIS is also pubicly available.

The CIS can be local as well as external. The stream of information from external CIS is brought forth by subscribing to a stream of context information in other components. I also thought about a concept where the client component requested for data every once in a while (a polling mechanisms), this is not as useful as subscriber mechanisms because the client could do a request while nothing is changed (unnecessary data traffic) or perhaps an item has changed multiple times between the polling moments and these changes where relevant for the client. So by providing the subscriber or notify observer pattern the client component is always up to date with the information from an external CIS.

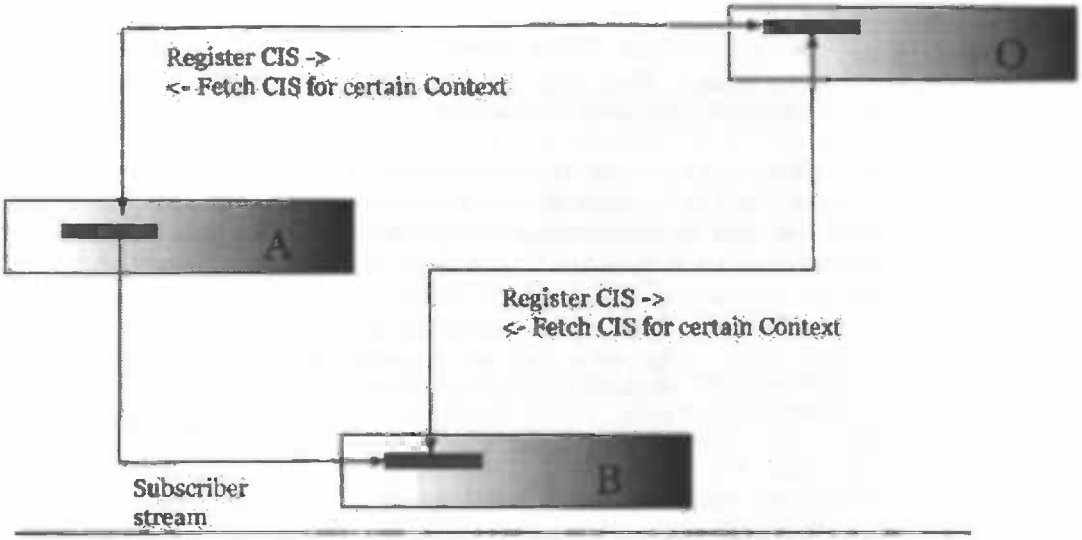

Figure 18. The overall component (O) provides functionality to find components providing certain context information. A and B are common context-aware components, where component B is subscribed to a context information stream of component A.

In Figure 18 an overview of a context-aware application is shown. As we've seen in paragraph 5.2 there exists one overall component in the context-aware software system. This component is named "O" in the figure. One responsibility of this components is that it collects information about the context information other components (in this case component A and B) provide. If a component wants to use this functionality it has to register itself and the context information it provides at the overall component. This way "O" builds up a database with components providing certain context information.

In case a component (in this case component B) is in need of certain context information it asks the overall component which component(s) can deliver such information. It then subscribes itself with the other component (in this case component A) and then A provides component B with new information as soon as it is gathered.

By subscribing to multiple context information sources collecting of certain data only has to be done at a certain level. We could perform Figure 18 at the system level but we could also see it at a higher level where each box represents a different system. By doing so we create a system where the components can fetch data from components at the same level, but also from a higher level and from other applications (because at each level interchange of data can be done).

There are a couple of properties that are dependent on the implementation and the platform we use. In certain cases a tracking of the history can be useful; but this results in higher memory usage. Another option which uses less memory is only saving the current context state resulting in less accurate analyses of the context.

When the context data source is being flooded a policy is needed to clean up the data source. Some policies are:

1. FIFO – The data first entered is first removed i.e. the oldest data will be removed first.
2. Oldest used first– The data that is used the longest time ago is being removed first.
3. Least referenced – The data that is being referenced the least (perhaps in a certain period of time) is being removed first.

The context sources consist of context elements. These are the data from the collectors and the engine. The context elements have certain properties, they are listed next.

Which collector: Which collector is responsible for fetching the data this element represents. This way we can discover which collector is responsible for measuring the data and request its way of collecting data (i.e. raw or intelligent).

Related Collectors: Here the collectors collecting the same type of context can be defined. If the data represented in this element is from poor quality or faulty we are able to find another component which perhaps has better measurements. Related collectors can be defined in multiple ways, two of which are:

- *Reference*; the collected data elements have a reference to other data elements providing the same data. This can be, for example, a linked list or a web where each data elements points to all other date elements providing the same data.
- *Record with multiple collectors;* In this case records exist; these records are a collection of data elements representing the same information.

Update: This flag indicates that the item has been updated. This way the context engine can determine if it will be useful to read the context element.

Error probability: This field indicates the probability that the measured data is faulty. When a selector in the engine indicates that an error or wrong measurement is done it increases the probability that the measurements contain an error. On the other hand if a selector determines that a specific element has to be used it could decrease the error probability. Depended on the intensity of the error or the correctness of the element the probability can be increased or decreased in bigger steps.

The context data source is only receiving data from the collectors. It is not able to adjust the collectors in any way nor is there feedback from the context data source to the collectors. In this way the collectors work in an autonomous way to collect the data. The context data source simply hasn't got the intelligence of dealing with problems in the collectors.

It does expect that the collectors conform to certain requirements. There needs to be an agreement of the update frequency from the collectors. The collector can only adjust the context elements it is responsible for. If there is none it requests an element to represent itself. If there is an element that represents the same data but is assigned to another collector it can request to use this element. This can be handy when a collector is broken and the new collector is replacing this broken version. Negotiation is needed between the two collectors.

If the old collector doesn't exist anymore this will be hard or impossible in this case the context data source locks the element and creates a new element for the collector.

The engine has certain points where it provides feedback to the context data source (as we will see in the later paragraphs). One of these is the estimator, the estimator translates the raw context data to context information. The context information is inserted into the context information source. The estimator provides its identity.

Another feedback moment is from the selector. The selector provides feedback on the probability of errors of the context element. When the element is chosen to be correct the probability is reduced, when it is not correct nor faulty it stays unchanged but when it seems to be faulty the probability is raised. Also see paragraph 5.3.5.

Summary

Responsibility	The context sources are responsible for providing a repository of context data, context information and additional collector information. The context information source can be included from a different external source. The context data source is a local repository for the current context architecture system.
Connections	The context source (CS) has connections with other context-aware components or context-aware applications via the subscribed CIS-es. The CS also has connections with the engine; specifically with the estimator(s) and the selector(s) which read the context data (the estimator) and provide feedback (both).
Future research	• An algorithm for deciding how well the gathered data of information is (probability of error). • Technical solution for the notify observer pattern and how to get the correct context information, i.e. a central component which is responsible for providing components and their context information: i.e. which information can be found where.

5.3.3 Engine

The context engine is responsible for analysis of the context data provided by the collectors in the context data source. It abstracts the data and saves it in the context information source. The engine analyses the data and adjusts the software system at hand according to the current context of the software system. The engine is shown in Figure 19.

In the first concept the engine was one component; it was some sort of spider where the most important decisions had to be made. During the research process different responsibilities where found; these responsibilities where separated to different components resulting in the following engine.

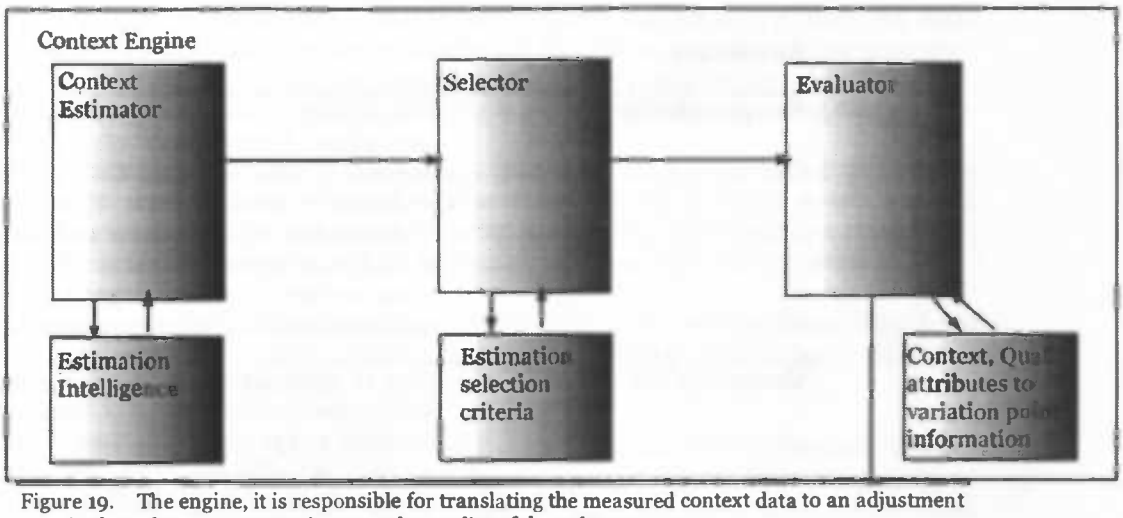

Figure 19. The engine, it is responsible for translating the measured context data to an adjustment in the software system to increase the quality of the software system.

In the next paragraphs the different components of the engine will be discussed. For starters the context estimator will be discussed, followed by the selector and the evaluator.

5.3.4 Context Estimator

In the Context Data Sources the measurements of the context are saved. The context estimator has the responsibility of estimating the context. This means the estimator translates the raw data from the sensors into more useful information. For example by using fuzzy logic the light intensity (measured in lux) is converted to a value like "day", "sunny", "cloudy", "night", "clear night" etc. A decision tree, neural networks etc are all options for measuring the context. But of course the estimator can also pass through the data the way it's kept in the context data source and the context information source.

The estimator collects data from the context data source by requesting for a specific type of context information. There are multiple ways of requesting the data:

- *Request by collector:* The estimator request data from a specific collector. The context data source returns the context elements provided by the requested collector.
- *Request by context type:* The estimator requests a specific type of context on a certain moment (in the past or the present). It gets all elements for the specific context with the collectors responsible for the requested context type.

The estimator could request multiple types of context elements, aggregating these elements results in a more abstract representation of the context. We could, for example, request the time of day, the processor load of the system, the number of processes running and here from decides that it is night, if the system has enough processing power left, the virus scanner could be started.

The results of the estimator can be provided to the context information source. The estimator has to provide the type of context and provide its ID.

The feedback on the information provided by the estimator is saved in the context information source. The result is that the feedback is also available for other context-aware systems.

Summary

Responsibility	The estimator is responsible for estimating a certain context-element (for example, the light intensity). Multiple estimators can estimate the same type of context, a selector is responsible for choosing the estimator with probably the best estimation.
Connections	The estimator uses the context data sources and context information sources to estimate the context. It also provides feedback to the context information source. The selector uses the estimated value to decide which estimator to use.
Future research	• How to transform the context data into context information. • Algorithm for deciding the usefulness of certain measurements. • Mechanism for providing feedback on the workings of collectors.

5.3.5 Selector

The selector determines which estimator has the best result for a specific context type. The evaluator requests the best option for a specific type of context; the selector then looks up the possible estimators and passes through the best context estimation to the evaluator.

There are multiple ways of determining the best option:

- *Previous measurements:* If the context source also stores the history of a certain element (whether abstracted, raw etc) we could trace back the chance that the current estimation is faulty. The estimation with the least probability of being faulty is best.
- *Other context estimations:* We could use the other context estimators. One way to do this is count the estimation that occurs the most (i.e. count how many estimators produce the same value). Choose one of these or use this to narrow for a next evaluation of these estimators (this of course costs more time but comes to more accurate context determination and is strongly dependent on the accuracy needed). Another way is to determine the current context from different type of estimators. For example an estimators tells there are no people at the office and it is dark outside we could narrow the results to "evening" or "night".
- *Probability of error of the estimator:* We could request the estimator to tell the average probability of error of the used context elements and choose the estimator with the lowest probability of error.

The selector should be able to keep track of the estimators who provide the same context information. Thus being able to ask all of these to estimate the context or it could use one of the above algorithms to create a smaller list of estimators it prefers to use this way reducing the number of estimators processing the context.

When the selector finds that a certain context estimator (or better the context element from the context data source it represents) is faulty it will provide feedback on this item towards the context source, this way creating probabilities of errors for the context elements (and their representative context estimators).

The selector could also gather these probabilities in its own database. This is not handy because it results in other selectors not being able to learn from the findings of other selectors.

The selector provides the evaluator with the best context estimation to its current knowledge. The evaluator will use the information of multiple estimators to determine the best way to adapt the software system at hand as we will see later on.

Summary

Responsibility	The selector is responsible for deciding which estimator is most likely to have the best context estimation. It passes the context information to the evaluator for further analysis.
Connections	The selector uses the context estimators and the context information sources to select the best context estimation currently available. It passes this estimation to the evaluator.
Future research	• Decision algorithm techniques for selecting the best estimation. • How to use the external information source to gather additional information.

5.3.6 Evaluator

The first alternative of the evaluator I studied is shown in Figure 20. In this figure a system monitor is included; though this seems logical it is not. In the system we gather context, as well intrinsic as extrinsic, this is why the system monitor is unnecessary; the data is already gathered by the collectors of the context system (and therefrom handled by the engine etc). The context system measures the inner working of the software system; the system monitor is therefore redundant.

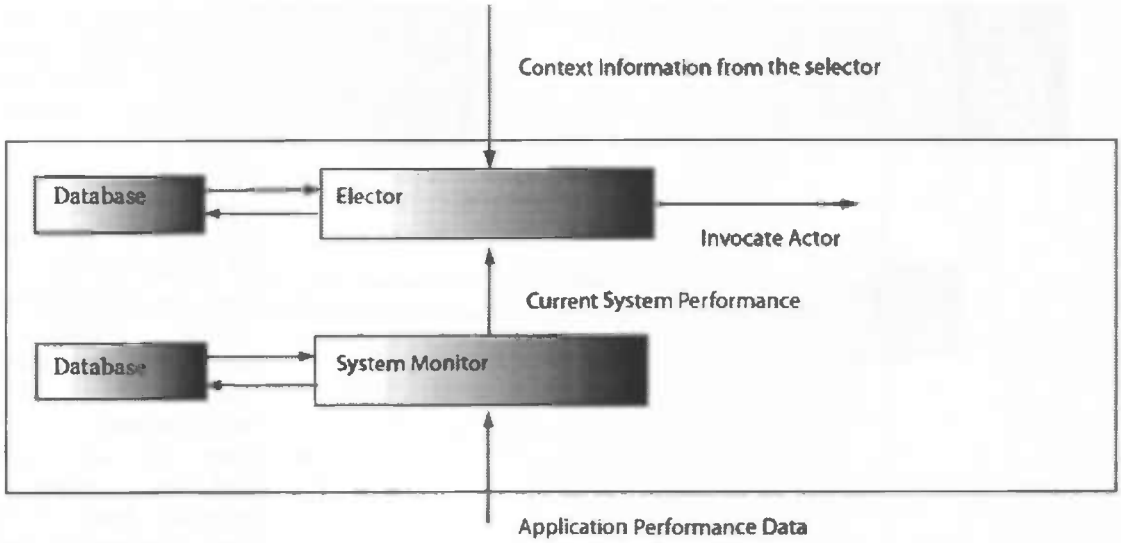

Figure 20. The evaluator of the context engine. The evaluator consists of an elector and a system monitor. The elector is responsible for making changes to the system which is monitored by the system monitor.

The evaluator is responsible for making a choice. This can be either to keep the system running "as is" or it could mean it decides a change is needed. The decision will be based on the current quality attributes of the system, the context and the ability to adjust the system to increase the quality attributes for the best.

To make a choice the evaluator has to decide which choices are available. The evaluator needs to have knowledge about all the possible variations of the software system and what the effect of each is on the quality attributes.

It estimates the improvement of each variation on the software system and decides which is best. Once the best variation is decided it makes sure the changes are made to the software system by invoking the actor.

At first the mapping of the quality attributes on the variations of the system will be static; later on this can be made dynamic. Due to the fact that this lies beyond the scope of this research I will not go into more detail on the methods for doing this.

An interesting topic of research in this field will be how the system responds when the mapping of the evaluator fails; will it be able to dynamically build this mapping?

Summary

Responsibility	The evaluator is responsible for analysing available variations and deciding if a variation is needed.
Connections	The evaluator gets context information from the selector and invokes the actor to change certain variation points in the software system.
Future research	• How to statically map the variations on the quality attributes.

- If the evaluator has no static map how should it proceed? How can it dynamically build the mapping?
- What should be built up when dynamically mapping the variations on the quality attributes.
- How can the outcome of a certain decision be measured and used for future decision improvement.
- How to use external evaluator rules, for example from a higher level component (with a more overall and general policy).

5.3.7 Actor

In one of the earlier design stages the context system was a part of the software system itself. This results in a tightly coupled not progressive architecture. By decoupling the context system from the software system both systems can be designed and built individually.

As we will see in paragraph 5.4 the actor is only a step towards a self-aware self-adapting software system. The actor is responsible for enacting changing in the running software system. The actor performs as the link between the context system, the software system and in the beginning the user.

An actor is responsible for one or more variation points. Invoking an actor results in changing the variations of the system which leads into the software system changing its quality attributes.

An actor is responsible for a certain variation point. It makes sure the variation point switches to a certain variation on the request from the user or the context system.

The relationship between the strength of the user and the context system can be changed over time; resulting in a system that emerges from a system adaptable by the user towards a self-aware and self-adapting system. For instance the choices from the user could weigh for 90% stepping it down towards a system where the context system has full responsibility. The latter algorithm is only usable with cardinal values (measuring the mean of the values "yes" and "no" is rather hard there the software system will not accept a "maybe").

Also it is possible to create a second (or even third to infinite) context system which is coupled to the actor. The actor uses the weight factor to decide which of all the inputs will be invoked and which will be ignored. This also results in a system which is able to use the higher level policy of higher components. See Figure 21.

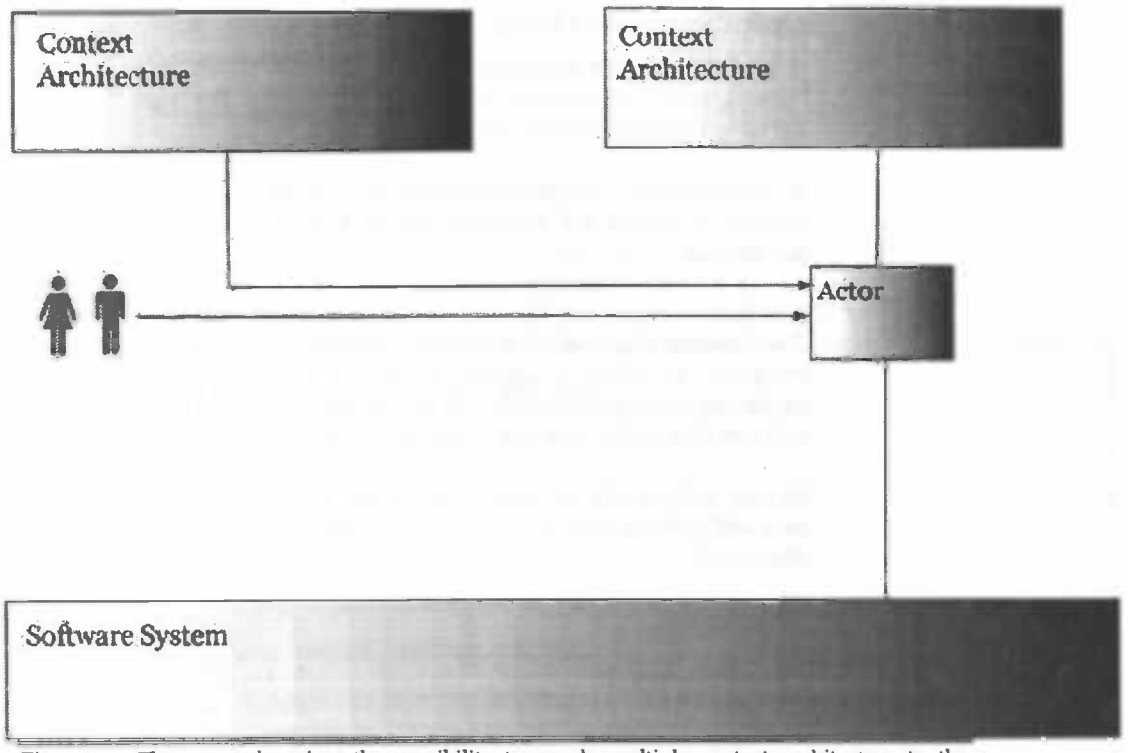

Figure 21. The actor also gives the possibility to couple multiple context architecture to the software system. For example one architecture could be from a more general level than the other.

The actor needs to know all the variation points in the software system for which it is responsible and the variations bind to these variation points. The evaluator only wishes to tell the actor "Change this variation point in this and this way" not exactly telling how the actor should change the variation of the variation point.

Summary

Responsibility	The actor is responsible for invoking variations in variation points. It also decides how a request for invocation weighs and which of these are acted upon and which are ignored.
Connections	The actor combines the information from the user, the context system (zero or more) and invokes variation request on the software system.
Future research	• How can external evaluators be coupled to the local actor? • What algorithms can be used to decide which variation to perform? • How to structurally build an actor which can change dynamically added variation points.

5.3.8 Variation Points

A variation point consists of a "name" by which it can be identified and a "description" which describes the variability the variation point delivers. A variation point can have multiple variants.

A realization of a variant describes the recipe on how to bind the variation to the system. It could for example define how to load a dynamic module or set a parameter.

A variation point can be bind to a requirement or for example a quality attribute. This means that when a quality attributes needs adjustment (i.e. the quality attribute is below or above a certain threshold) we could summon all the variation point associated with the quality attribute to change. In our purpose we will use this in the evaluator and more specific in the elector.

In our system the variation points are invoked by the actor or the user (in the non-self-aware, non-self-adapting version). In [17] more about variation points is discussed.

Summary

Responsibility	A variation point is a point which has multiple variants resulting in different workings of the software system.
Connections	A variation point is a part of the software system and is controlled (in this case) by an actor.
Future research	• How to structurally add variation points to applications. • How to dynamically add variations to variation points and variation points to software system. • How to decide which external "force" is able to change variation points and which isn't.

5.4 Progressive Implementation Scheme

Though we may want to change our system directly into self-aware and self-adaptive systems we are not able to do so instantly; a progressive implementation scheme will be more useful to gradually make the system context-aware. In this paragraph I will discuss a progressive implementation scheme to evolve a software system into a self-aware and self-adaptive software system.

Let's consider a standard software system. The system is static and only accepts user input (as seen in Figure 2 (a)). We will first have to redesign this system to use it with variation points and an actor to invoke the variation points. See Figure 22.

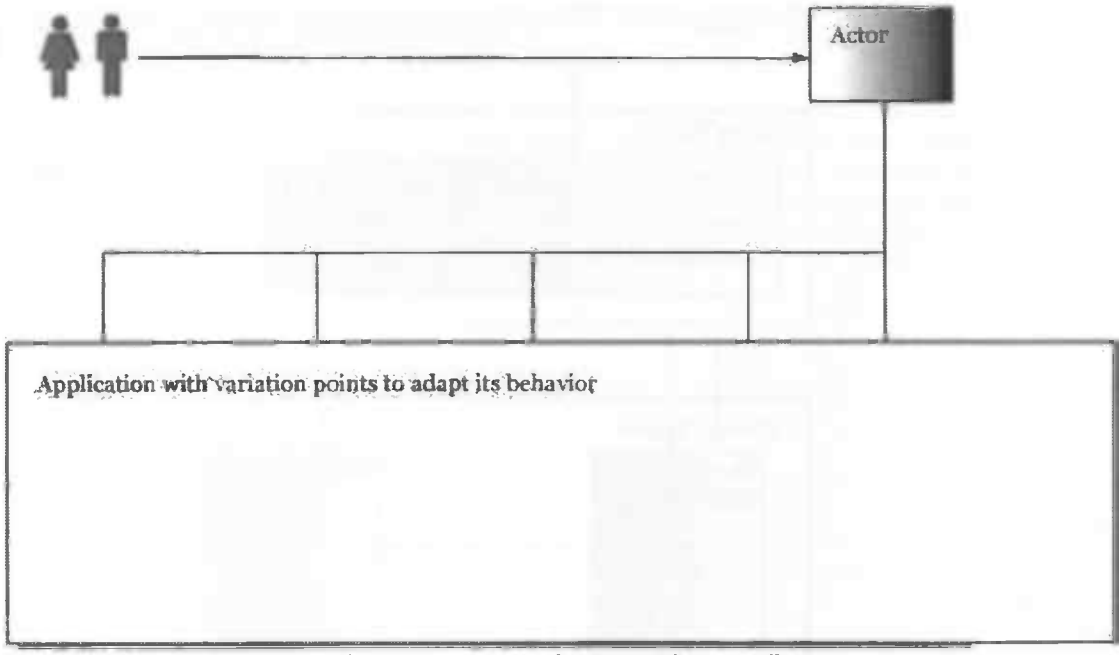

Figure 22. A software system with variation points and an actor. The user tells an actor a certain variation points has to be altered to increase the quality attributes of the software system.

This system has the property that it can be adapted by switching variations at the variation points. This could result in the system performing better, the user is still responsible for enacting these changes. Dozens of examples can be found of applications operating at this level. An example would be, for instance, a Nintendo Gameboy advance SP®, when it is dark we can decide to switch the backlight on so we can see what is happening on the screen.

The next step in the scheme would be to extend the software system with a context system. This context system monitors the software system and the context of the system. It decides which variation points should be changed to which variants for best performance. This variant is shown in Figure 23.

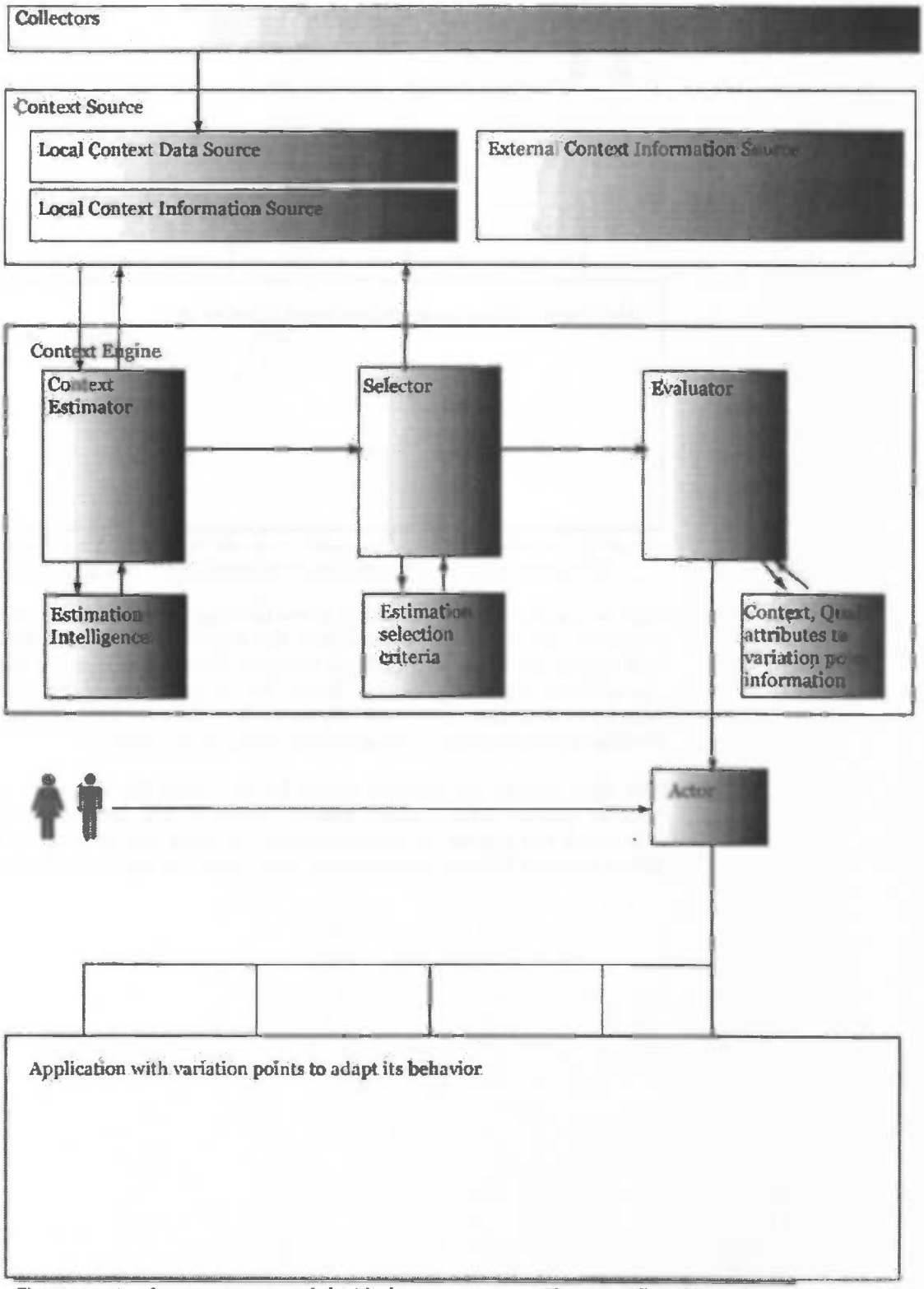

Figure 23. A software system extended with the context system. The user still is able to change variants in variation points but the context system also changes variation points.

The components and workings of the context system are described in earlier chapters. A link between the system and the context system needs to be established, this is the actor's responsibility. The user is still able to adjust variation points in the application and this way able to override the working of the context system. By gradually giving the context system more responsibility (by changing the weight of user input and the architecture input) we are able to

move towards the next step, a context-aware, self-adaptive and self-aware software system.

The gameboy could for example measure the light intensity of the environment. And context estimator changes the data into context information like "light environment" or "dark environment". The selector determines which estimator is best and the evaluator decides what is to be changed in the software system. So when it's dark the backlight is turned on and when it is light it is switched off.

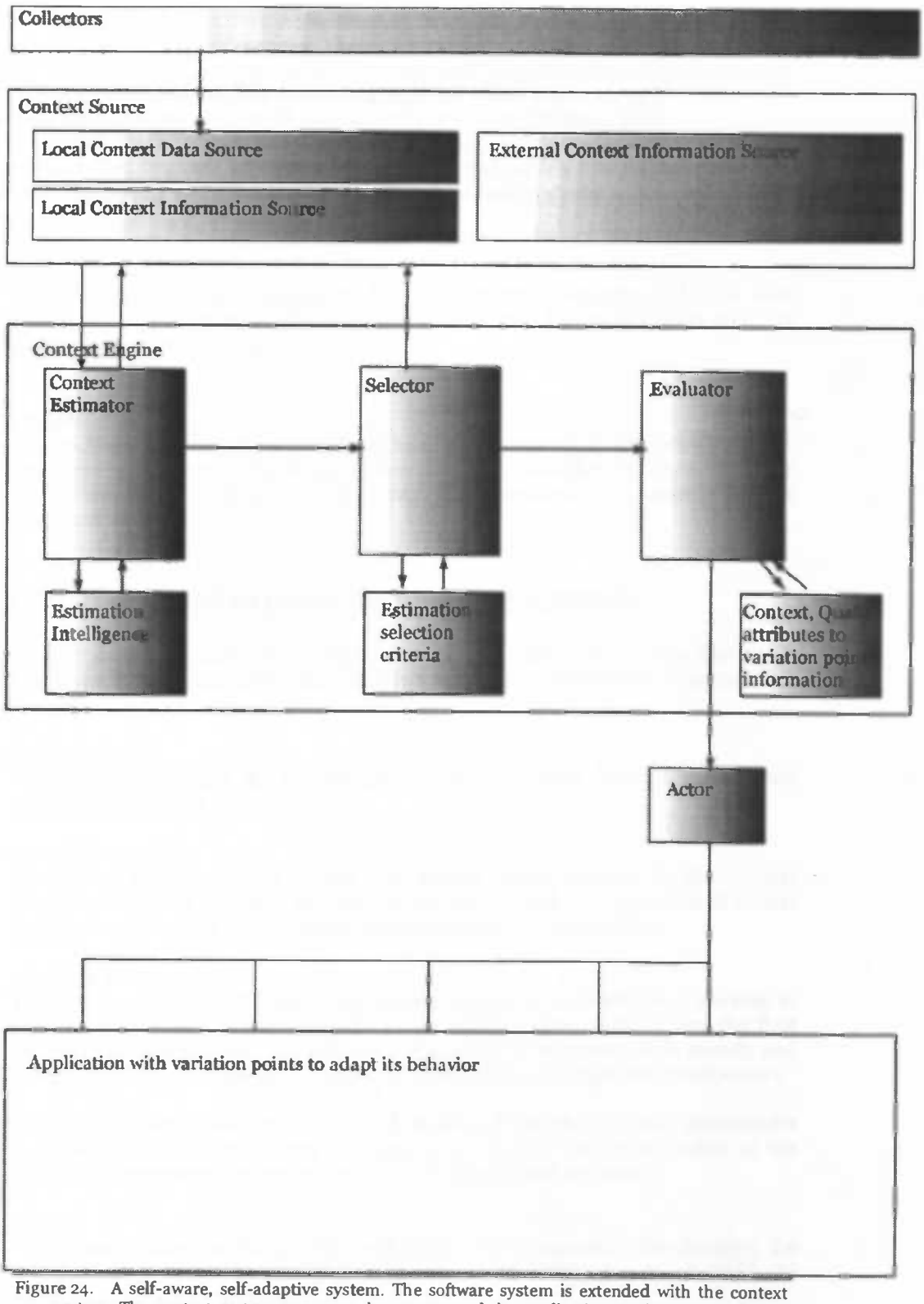

Figure 24. A self-aware, self-adaptive system. The software system is extended with the context system. The context system measures the context and the application performance, when it decides changes to the software system are needed it does so increasing it's quality attributes.

In the last step the user is not able to alter the variants of the variation points anymore. The context system is fully accountable for the changes made to the software system. This results in a self-aware, self-adaptive software system.

The user input should be completely removed in relation to the context information. The context system is now a part of the software system.

6 State of the art vs. Context System

The architecture discussed in the last paragraph is only interesting if it is usable. I tried to create a general version of the versions I discussed in 3.3 and added some additional functionality to improve the usage of a context aware architecture.

6.1	In this paragraph I will discuss the Gauges architecture in reflection with the architecture described in this paper.
6.2	In this paragraph the Widgets architecture is fitted into the context architecture of the previous chapter
6.3	We are also able to fit the CAS system into the designed architecture.
6.4	The requirements from 5.1 define what the context system should be able to fulfill. In this paragraph I will describe why the requirements hold for my architecture and which hold for the other architectures. I will conclude this with an overview in a table.

6.1 Gauges

In paragraph 3.3.1 the "gauges" architecture is discussed. When we study this we can conclude the following.

Collectors
In the gauges architecture "monitoring" is the same as the function of the collectors in my architecture. For the actual collecting of data "probes" are used. These probes correspond to the collectors in my architecture.

Context Source
There is no context source that can be exchanged with other systems with the same architecture. This results in a context-aware application at the local level.

Because the gauge only passes through the (manipulated) data from the probes and there is no information about previous measurements, it will be rather hard to define the performance of certain probes and gauges.

Context Engine
The gauge consumer functions as the engine of the context-aware application. They analyze the data gathered from the gauges and adjust the application.

The gauges can be seen as the estimators, the gauge consumer encapsulates the selector and evaluator. Because these components are not separately described the working of these components is compromised and the gauge consumers have a rather blurry responsibility on this aspect.

Actor
Because the gauge consumers are a part of the application at hand there exists no actor. This also causes a lack of a control point. The user can't override the working of the context system in any way.

Progressiveness
This architecture is not progressive; the architecture is, because of the fact that the gauge consumers are a part of the application, not able to gradually make an application context-aware.

Conclusion
The architecture discussed can be completely designed in the architecture I discussed in this paper. By using the functionality I described in my architecture no functionality in relation with the gauges architecture is lost, merely there is functionality gained.

6.2 Widgets, Interpreters and Aggregators

In 3.3.2 an architecture using widgets, interpreters and aggregators is discussed. I will now discuss this architecture in relation with the architecture I discussed in this paper.

Collectors
Context widgets meet the functionality of the collectors (though the context widgets also contain an estimator).

Context Source
The system has no context source. The data is passed through by the context widgets for instant use. This also results in a system that only performs at a local level and exchange of knowledge by different systems is impossible.

Context Engine
There is no real context engine. Estimation occurs at different level starting at the context widgets, at the interpreters and at the aggregators. Where the first only estimates it roughly, the interpreters use the data to abstract it merely and the aggregators combine the information from different widget and interpreters.

The context-aware service uses the information from the widgets, interpreters and the aggregators to change the system at hand or the environment of the system. It represents the engine and the actor (as we will see later).

Actor
The context-aware service represents the actor. It is responsible for changing the application. There is no description about a user being able to override the decision in the context engine.

Progressiveness
The context architecture is not really progressive. Because the components are scattered among the application it means we have to redesign the application to include the context-aware components. Older applications can not be easily adjusted using this architecture.

Conclusion
Due to the tight coupling the items of the widgets architecture can not be mapped directly onto the architecture I discussed in this paper. But nonetheless the functionality of the widgets can be completely taken care of by the context system I discussed in this paper.

The components of this system are divided in the application. This has as a consequence that from the starting of the design of the architecture we need to bear in mind the context-awareness aspect of the system. It is not a progressive architecture which allows gradually increase of the context-awareness of the software system.

6.3 Context Awareness Subsystem

In 3.3.3 the architecture using a context aware subsystem is used. In this paragraph I will reflect this architecture to the architecture described in this paper.

Collectors
The collectors in this architecture are very low profile. They are described as "contextual information". In the architecture they are also called "context features".

Context Source
The client application holds the context information functioning as a context source. The context information is send as "context metadata" to the context engine.

The context source performs on a local level. This means that the context information is not interchangeable between applications or devices.

Context Engine
There exists a component called the context engine. This matches more or less with the context engine of my architecture. The context engine functions as an evaluator of the context information. It tells the content engine to deliver certain information dependent on the context information it gets from the context source.

Actor
Because of the tight coupling between the content engine and the context engine there is no real actor. The content engine acts on the information from the context engine. This acting results in the adjustment of the data send to the client device.

Progressiveness
Because of the tight coupling between the context engine and the content engine this architecture is not very useful in a wide variety of software systems. The content engine acts upon the information gathered by the context engine. If this coupling becomes looser (like the context system architecture) the architecture will be useful in a more wide variety of systems and become more progressive.

Conclusion
The system is describable in the components of the context system architecture. Still this architecture is a very ad-hoc and implementation specific. This results in it not being very useful in other applications.

6.4 Meeting of Requirements

First of all we see that the architectures that I discussed are all specific implementation and very ad-hoc. The architecture discussed in this paper is an abstract version of these architectures (at least all of them can be implemented in the architecture).

The authors of the other architectures all had more or less the same architecture with only different names. The context architecture covers the requirements of all of the architectures and adds functionality to each of them giving it more value.

In paragraph 5.1 10 requirements are described. I will now compare the 4 architecture using these 10 requirements.

Context System Architecture
The software system using this architecture must have variation points, as a consequence of the application having variation points (and the hereby including variations) the software system is a dynamic one. From [17] we can conclude that this is a dynamic software system so, Req. 1 holds.

The evaluator reflects the current choices by knowing what needs to be measured and requesting this information from the selector. It also has knowledge how to qualify the measurements against the current choices. Req. 2 holds.

The engine has the evaluator as a part of it. The evaluator is responsible for finding better variations in the current context. If it finds a better variation (i.e. the variation improves the quality of the software system) the evaluator decides for a change. The evaluator summons an actor to perform the needed change to the software system. The evaluator (and for Req. 5 also the actor) causes Req. 3 , Req. 4 and Req. 5 to hold.

The collectors are responsible for collecting data and information about the context of the software system. Req. 6 holds.

The context sources are able to save context data (from the collectors) and context information (from and estimator, selector or an external source). The context source makes Req. 7 hold.

The context engine in a whole is responsible for analyzing the context data (and transforming it into context information or an adaption of the software system). Req. 8 is fulfilled.

In paragraph 5.4 a progressive scheme is introduced which is used to convert an existing software system to a context-aware software system. This is described in Req. 9 .

The software system is independent from the context system. The context system can be decoupled from the software system and the software system will still function as expected, this makes Req. 10 hold.

Gauges
Gauges are used to measure the context. The gauges architecture "ends" at the gauge consumer. The gauge consumer can be anything and do anything (including "update an abstraction/ model" etc.), in their case the gauge consumer is the model manager from their integration framework. This framework has the ability to adjust the software system, Req. 1 holds.

When the framework decides at a certain level it doesn't know what to do the information is passed to the upper layers. The layers are responsible for changing

the software system, it is also responsible for assessing the current choices (Req. 2 holds). It also decides for change and determines what to change Req. 3 and Req. 4 hold. The layer also causes the changes to happen which fulfills Req. 5 .

The gauges are responsible for measuring the context, there is not specification about a context database to save context information. Req. 6 holds, but Req. 7 does not hold. The analyses of the context information occurs in the framework layers, Req. 8 holds.

The framework is directly related and integrated in the software system. This means a progressive scheme is not possible. Req. 9 and Req. 10 do not hold.

Widgets

The widgets architecture is more an architecture to gather information about the context than it is an architecture to adapt a software system depending on this context. An application can use uniform widgets to gather information about the context.

As a consequence of the widget architecture only providing a toolkit to gather context information Req. 1 to Req. 4 do not hold (these are all requirements aimed at the software system at hand).

There exist context-aware services (which are able to enact a change in the virtual or physical world), so Req. 5 holds.

The aim of this architecture is measuring the context using uniform widgets, this suggests that Req. 6 holds.

Saving and analyzing context data should be done by the application therefore Req. 7 does not hold. Because of the interpreters we could say that a part of Req. 8 does hold.

The widgets are an integral part of the application, this results in the lack of a progressive scheme and a tight coupling of the context system and the software system, Req. 9 and Req. 10 do not hold.

CAS (Context Aware Subsystem)

In this architecture there is a CAS which consists of an context engine which handles the context information and adjusts the content engine to provide better response to the user.

The context engine chooses the best option from the content engine and decides what is best. Here from Req. 1 Req. 3 Req. 4 and Req. 5 hold. Req. 2 does not hold, the choices made are very ad-hoc and only based on the current context information. It is always analyzed and passed through using the context engine, so the assessment is not relevant in this case.

The client measures its context, how this is done is not said. But with less imagination this can be implemented in the client. Req. 6 holds.

The context information is used in quite an ad-hoc manner; the context information is not saved in any place. Req. 7 does not hold

To decide what information is to be send the context engine has to analyse the context fulfilling Req. 8 .

The coupling of the context engine is quite tight; the context engine has to have very much knowledge about the content engine. The system is not general enough to use it in a wide variety of software system. As a result of the tight coupling a progressive scheme is not possible. Req. 9 and Req. 10 do not hold.

Summary

Summarizing all the analyzed architectures into a table we see that only the current architecture holds all the requirements. This does not mean that the other architectures are useless; the architecture discussed in this paper is a more abstract version of the other architectures. Some of them can be used as a part-of the architecture from this paper while with others a translation can be made from the components.

In the following table the requirements and their fulfillment at the different architectures is shown.

	Req 1	Req 2	Req 3	Req 4	Req 5	Req 6	Req 7	Req 8	Req 9	Req 10
Gauges	+	+	+	+	+	+	-	+	-	-
Widgets	-	-	-	-	+	+	-	+/-	-	-
CAS	+	-	+	+	+	+	-	+	-	-
CSA*	+	+	+	+	+	+	+	+	+	+

Table 2. This table shows the architecture against the requirements from 5.1. As we can see only the Context System Architecture fulfills all the requirements. The other systems are often a specific version of the architecture discussed in this paper.

7 Future Work

In this chapter I will discuss a few besides the topics discussed at the components paragraphs which could be a subject to research in the near future.

- **Create a technical design.** A technical design should be made which defines the inner workings of the different components. Widely used is the Uniform Modeling Language (UML) to do this.
- **Create a prototype.** Create a prototype application using this framework. This prototype should be a proof-of-concept for newly designed applications.
- **Extend an existing application.** Extend an existing application with the context-awareness architecture to test its usage in this case.
- **Run-time dynamic learning.** Research on the possibility of run-time learning in the evaluator. Topics could be the use of neural networks, decision trees, dynamic neural networks etc.
- **Mapping measurements on quality attributes.** Research how the measurements (ultimately the context) can be mapped on the quality attributes of the software system. This could also encompass measuring the current performance of the software system.
- **Mapping variations on quality attributes.** How can the variations influence the quality attributes in the software system and how can we save the mapping of these. This could also include decision making according to this mapping.
- **Evaluator design in more detail.** Separate different responsibilities in more detailed components.
- **Sharing evaluator rules.** Ultimately the overall component is able to share it's policy with it's children. We should research how this influence of higher level policy and the transaction of the bounded rules should be done. This means that the higher level component could interchange rules with lower level of equal level components.

8 Conclusion

In this thesis I aimed at getting a better understanding of architectures for context-aware software systems. I described some existing architectures, and compared them based on the features context-aware systems have in general (these are not the requirements for the architecture).

Next I discussed the most commonly used architectural styles at this moment. Though these styles form a solid base from which architectures can be designed they often aren't fully suitable for a particular problem as is the case in my situation. Complex systems require a composition of the commonly used styles to fully fulfill the requirements of the problem at hand.

In the context architecture we will need parts of the system to be responsible for gathering data (event-based), saving data (repository) and analyzing data (pipes and filter) this results in a composite architecture where each problem has a specific solution in these parts of the architecture.

The next step was to create an architecture style that fulfills the requirements that hold for a context-aware software system. First I had to define these requirements. In short these where:

Req. 1 Possibility of choice
Req. 2 Assessment of the quality in the current choices
Req. 3 Deciding for change
Req. 4 Determine what to change
Req. 5 Enacting change
Req. 6 Measuring context
Req. 7 Saving the context information
Req. 8 Analyse context data
Req. 9 Progressive scheme towards self-adapting system
Req. 10 Loose coupling

The (reference) architecture I designed in my research fulfills all these requirements. Although the other architectures I looked at in this thesis fulfill the requirements for their specific problem they are not general or wide enough. The architectures are aimed at specific problems or are only aimed at gathering context information and not the analysis of these.

The other architecture all had a lack of loose coupling, the systems where all tightly coupled to the software systems they where used by. The architecture I describe in this paper is loosely coupled and can even be removed from the software system. The software system will still perform without the context architecture being available. This is a consequence from the definition of context I used, which is:

> *"Context is any kind of information which can affect the quality of a software system."*

This definition has a subtle difference with the definition from other researchers. The difference lies in the fact that I describe context as any kind of information that affects the quality of a software system, this means that it does not affect the functionality of the software system. If the information used does affect the functionality of the software system, the system is not context-aware rather it is

for example location-aware (if location is the information used to affect the quality of the software system).

The architecture I described in this paper is very general. The architecture can be build separately from the software system. There are few points where direct intervention with the software system is needed. This is the monitoring of the performance of the software system and the actor which is responsible for changing the variation at a certain variation point.

In this research I did not create a proof-of-concept of the architecture, this can be one of the next research topics.

References

1. H. Lieberman, T.Selker, "Out of context: Computer Systems That Adapt To, and Learn From, Context", *IBM Systems Journal Vol. 39, NPS. 3&4, 617-632 (2000)*
2. B. Schilit, N. Adams, R. Want, "Context-aware Computing Applications", *Proceedings Workshop on Mobile Computing Systems and Applications, 8-9 dec 1994*
3. P.J. Brown, J.D. Bovey and X. Chen, "Context-aware application: From the laboratory to the marketplace", *IEEE Personal communications, 4(5), 58-64, 1997*
4. A.K. Dey, G.D. Abowd and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications"
5. T. Selker, W. Burleson, "Context-aware design and interaction in computer systems", *IBM Systems Journal Vol. 39, NPS 3&4, 880-891 (2000)*
6. D. Garlan, B. Schmerl, J. Chang, "Using Gauges for Architecture-Based Monitoring and Adaption", *Working conference on Complex and Dynamic Software Architecture (2001)*
7. P. Lonsdale, C. Baber, M. Sharples, T. Arvanitis, "A Context awareness architecture for facilitating mobile learning", *Proceeding of MLEARN (2003)*
8. A. K. Dey, "Providing Architectural Support for Building Context-Aware Applications", *Thesis for PhD (2000)*
9. M. Ritchie, "Pre- & Post-Processing for Service Based Context-Awareness", *paper University of Glasgow*
10. R.M. Gustavsen, "Condor- an application framework for mobility-based context-aware applications", *Gothenburg, September 29, 2002*
11. J. O. Kephart, D.M. Chess, "The Vision of Autonomic Computing", *IEEE, 41-50, January 2003*
12. P. Horn, "Autonomic Computing: IBM's perspective on the State of Information Technology", *USA October 2001*
13. Stickyminds website http://www.stickyminds.com/sitewide.asp?ObjectId=2909&Function=DETAILBROWSE&ObjectType=COL
14. http://fmserver.sei.cmu.edu/plp-arch-ro
15. D. Garlan, M. Shaw, "An Introduction to Software Architecture", *Pittsburgh January 1994*
16. J. Bosch, "Delegating Compiler Objects - An Object-Oriented Approach to Crafting Compilers", *proceedings Compiler Construction '96*
17. I. Bosloper, J. Siljee, J. Nijhuis, "Modeling Dynamic Software Systems with Variation Points", *Configuration in Industrial Product Families 14 march 2004*
18. M. Weiser, "Hot topics in ubiquitous computing", *IEEE Computer "Hot Topics", October 1993.*
19. A. Shalloway, J.R. Trott, "Design Patterns Explained, A new perspective on object oriented design", *Addison-Wesley Professional July 2001*
20. T. Winograd, "Architectures for Context", Human Computer Interaction 2001 volume 16