High level performance analysis and dependency management

Matthijs Wessels

July 2010
High level performance analysis and dependency management

*Applied at Philips Healthcare in Best*

M.F. Wessels

*Master thesis*

Department of Mathematics & Computer Science
Software Engineering & Technology
Eindhoven University of Technology
Abstract

Quality is an aspect in software that plays an important role in Philips Healthcare. This thesis describes a research project that helped Philips Healthcare on two fronts in producing high quality software. Firstly, a solid software architecture is an important factor in the creation of quality software. Managing the dependencies within a system is one aspect that contributes to this. This thesis presents a survey of methods that are available for Philips Healthcare, to manage their dependencies. Secondly, response times are another factor that contributes to the quality of software. This thesis introduces a method to analyze the performance of a large software system on a higher level than conventional methods do. Before, finding the cause of a performance problem would be a time consuming activity due to the large scale of the system. Using the introduced method, the performance problem can be localized very fast, after which the engineer can focus on a smaller part of the system.
1 Introduction

This thesis is the result of a graduation project for the master Computer Science & Engineering at the Eindhoven University of Technology. For this project, research has been done into software analysis. It has been carried out at Philips Healthcare in Best under the supervision of the Software Engineering & Technology research group at the university. Quality is an important aspect at Philips Healthcare. The aim of this project is to improve and/or help to maintain the quality of software developed at Philips Healthcare. Dependencies between different subsystems can influence the quality of a system [11]. A survey has been done for a way for Philips to monitor the dependencies between the building blocks of their architecture. Next to this survey, a new method has been developed and implemented for localizing the cause of performance problems.

This chapter will start by talking about software analysis in Section 1.1. After this, Section 1.2 will treat the Philips organization. In Section 1.3, Philips Informatics Infrastructure, a standard framework for medical imaging systems, is introduced after which Section 1.4 will talk about the subsystem on which the analyses described in this thesis have been performed. Then, Section 1.5 will explain what software analysis can do for Philips Healthcare. Lastly, Section 1.6 will provide the reader with an overview of the rest of this thesis.

1.1 Software analysis

When developing software in a professional context, a design is always made before a system is realized. However, as development goes on, it is not uncommon that deviations from the original design sneak into the implementation [10][6]. These deviations are usually small and are overlooked due to multiple people working on the same large code base. Over time, even though the deviations are small, the gap between the original design and the implementation grows. This not only results in the source code becoming less comprehensible for developers, but also can mean that certain design goals, set by architects, are no longer achieved. An example of such a design goal could be: “Subsystem A should not depend on Subsystem B”.

The research field of software analysis aims to solve this problem in several ways. One branch is to reverse engineer existing systems by deriving a design from the implementation. This will allow developers to understand the software better. It also allows architects to compare the generated design to the original design and verify their architectural goals. The most common way to do this reverse engineering is by basing the analysis on the source code. This type of analysis is known as static analysis.

Static analysis has some limitations however. For example, to keep different subsystems separated, links between these subsystems can be made dynamically (i.e. only after the system has started). This means that when looking only at the source code, one cannot always derive these links. This results in an incomplete view of the system. Furthermore, analyzing the behavior of a system becomes more difficult when only looking at the source code. There are so many factors that play a role when executing a software program, that it is not possible to predict all possible behavior.
This is where dynamic analysis comes into play. Dynamic analysis differs from static analysis in that it
looks at systems at runtime. Dynamic links between subsystems can be detected in this way for
example. Furthermore, using dynamic analysis, performance aspects like memory usage, timing, etc. can
be analyzed. Dynamic analysis however, also has some limitations. Running a system once or even
several times only shows a subset of all possible behavior.

1.2 Philips Healthcare organization
Currently, Philips is active in three sectors, Healthcare, Lighting and Consumer Lifestyle. With sales of 23
Billion euro in 2009, Philips is leading in cardiac care, acute care and home healthcare, energy efficient
lighting solutions and new lighting applications, as well as lifestyle products for personal well-being and
pleasure.

Philips Healthcare has most of its business in imaging systems like CT, MRI and x-ray scanners. Its aim is
to be “people focused” which means that they do not just provide the machines. They design their
products to accommodate the medical procedures in such a way to allow the patient to be as
comfortable as possible and the doctor to work efficiently.

In the business unit Interventional X-Ray (iXR), Philips develops cardio vascular interventional x-ray
machines to support minimally invasive surgery (see Figure 1.1). By means of inserting a catheter into
the patient’s blood stream, a doctor can perform minimal invasive cardio and vascular procedures. The
system uses x-ray to provide the doctor with live images of the patient’s blood vessels during the
procedure. Figure 1.2 shows how iXR is positioned in the organization.
Figure 1.1: An x-ray machine developed by Philips Healthcare

Figure 1.2: Organizational diagram highlighting where Interventional X-Ray is located within Philips.
1.3 Philips Informatics Infrastructure

Philips Informatics Infrastructure (PII) is a medical platform, used by multiple medical imaging devices within Philips Healthcare. This system provides common functionality and a similar look and feel for all imaging products. This section will provide some background information on how PII is used within Philips and how it has grown over the years.

1.3.1 History

PII started out around 1998 under the name Easy Vision Modules (EVM), only providing components for archiving, connecting and printing (ACP). The technology used at the time was Java 1.0. As time went on, more functionality was added to ACP and it was renamed to Medical Imaging Platform (MIP). Around 2001-2003, more viewing components were added to MIP. Due to problems between Microsoft and Sun regarding Microsoft’s implementation of the Java Virtual Machine, a move was made from Java to J# which is based on .NET. Gradually J# was being phased out and more and more code was written in C#.

In 2007 Philips Medical Work spot (PMW) was released. With PMW the scope of PII products was changed from just providing components with commonly used functionality to providing an almost-finished product. End users now only need to develop plug-ins on PMW to implement functionality specific to their end-product requirements.

Earlier in 2005, a company providing a solution called iSite was acquired by Philips. iSite is a solution that supports remote management and usage based billing. This will allow a hospital to outsource all its data management. This is where the name PII comes into play. The target of PII is to integrate iSite with PMW. Current product releases of PII do not contain this integration yet, so now PII just provides a common environment for the development of Philips Healthcare software.

1.3.2 Development with PII

Figure 1.3 shows three models of software development used within Philips Healthcare. The leftmost one (a) is how applications were traditionally developed. Here the application is responsible for everything and common functionality is grouped in reusable libraries. The second model in Figure 1.3 (b) shows the way development is done using for example MIP. An application chooses a set of components and just adds glue-code for the components next to application logic specific to the application. The model on the right (c) shows how development with the latest version of PMW in PII works. All the interfaces and mechanisms connecting the components are standardized. A working application can now be developed just by defining the connections between the components. These connections can be expressed as a protocol using a domain specific language and can be saved in an XML file. This allows for rapid development of medical applications.
This allows for rapid development of medical applications. To support this, PII offers a visual application builder in which links between components can be specified using diagrams. An example of an application allowing a user to view and edit medical images can be seen in Figure 1.4.

Figure 1.3: Three stages in the evolution of software development at Philips Healthcare

Figure 1.4: The visual application builder on the left and the resulting application on the right.
1.3.3 Public and private interface
Within Philips Healthcare, PII is developed by a separate business unit (Healthcare Informatics-Patient Monitoring in Figure 1.2). The customers of this business unit are the other business units that build applications on top of PII. To avoid breaking their customers’ code when changing or fixing something, the components in PII are split up into two categories: public and private. Customers should only use public functionality of PII. For public interfaces of components, a detailed description is made describing exactly what the interface does and how it is to be used. These public interfaces will almost never change.

If for any reason PII does want to change a public interface or its behavior, PII will first have to go through a strict process to ensure that all stakeholders agree. Changing private interfaces can be handled internally. Because PII is still being developed, some functionality is still undergoing a lot of changes. The interfaces for this functionality are kept private to avoid having to go through the lengthy process of approving the change. If a customer would like to use an interface that is currently private, it is possible to file a request which might lead to the interface becoming public.

1.4 Back-end X-ray
The research for this project has been carried out in the team that builds BeX, which is short for Back-end X-ray. For the x-ray machines described in Section 1.2, Philips is using a new architecture called Common X-ray Architecture (CXA). CXA can be divided into three main subsystems, a front-end (FE), image processing (IP) and a back-end (BE or BeX). These parts communicate with each other through standardized “CXA” interfaces (see Figure 1.5) allowing the development and the lifecycle of the three subsystems to be completely decoupled.

![Figure 1.5: The CXA Architecture where an arrow represents the direction of data flow between two subsystems.](image)

The research for this project will use BeX as a case study to verify its results. The BeX system is responsible for patient administration, connectivity to hospital information systems, most graphical user
interfaces and a number of imaging applications. BeX is one of the systems at Philips Healthcare that is based on PII. The primary focus of the case study is on BeX specific code. Nonetheless, PII was not considered as a black box and the developers from PII were also consulted during the project.

1.5 Research problem
The development of software is usually divided into several phases. After completing some of these phases it is possible to go back and reiterate earlier phases. A common division would be:

- Specify requirements
- Specify a design
- Implement the design
- Test the implementation

Any of these phases can be reiterated after which the following phases are repeated as well. The test and implementation phases are usually interleaved in the way that a small piece of functionality is implemented and immediately tested before the next piece of implementation is added. When a test fails the implementation has to be adjusted until the test succeeds. However, when the implementation is changed, the design phase is not always reiterated; causing the implementation and design to drift apart. This is a common phenomenon as described earlier in Section 1.1 and in [10]. Architectural rules that have been set up during the design phase are then in danger of being violated. For this reason, it is useful to have some way to verify these architectural rules after an iteration of the implementation phase has been completed. This results in the first research question:

Q1 How can the software architecture be guarded using dependency checking?

Furthermore, during the test phase the implementation is tested to comply with functional and non-functional requirements. Performance is one of these non-functional requirements that is tested. When looking at the requirements from a performance perspective, the system can be divided into several use cases. A use case can contain one or more steps that are performance sensitive (usually not more than one). In this context, the term use case is not used to indicate the performance sensitive step in the use case. Every performance critical use case is defined in the requirements phase where it also receives a time budget. These budgets are verified during a test phase, usually after a new feature is delivered by an engineer or after integration.

If a use case takes more time than the budget assigned to it, the implementation needs to be changed to fix this problem. The information that “Use case A is over its budget” is rarely enough for a developer to find out what exactly is causing the problem. Available tools that analyze the performance of a system in more detail, provide information on the level of source code (e.g. time spent in method x or number of calls to method y). For large systems such as the ones being developed at Philips Healthcare, the information provided by such tools can be overwhelming. Furthermore, to provide information on such a low level, the measurements the tools perform will slow down the system or even break it. Summarized:

- “Use case A takes too long” is too little or too high-level information.
• “Method B takes x time” or “Method B is called y times” for every method in the system is too much or too low-level information.

From this the following research questions and sub questions can be derived:

Q2 How can we get insight in the execution and timing of a use case?

Q2.1 What level of detail does the information need to have to be useful?
An intermediate view is required. It must be more detailed than an overview of which use case takes too long, but less detailed than what a profiler generates.

Q2.2 How can this information be conveyed to a developer?
Text gives the user exact information, but an image can tell much more in less time. There are many different ways to visualize the same data. It is important that the attention is drawn to important parts such as outliers.

Q2.3 How can the relevant information be acquired?
Before anything can be shown to the user, the data must first somehow be acquired.

Q2.3.1 What data should be obtained?
It might not be possible to measure the data that is of the right level of detail. That means other data will have to be obtained.

Q2.3.2 How can the data be obtained?
To acquire accurate data, the system should not be influenced too much by the measurements.

Q2.3.3 How can the required information be derived from the obtained data?
After measurements are finished, the resulting data must be converted to the level of detail determined in Q2.1.

1.6 Structure of the thesis
In chapter 2, the research related to question Q1 is treated. It will talk about the approach used and what the results were. Chapter 3 will introduce the reader to the field of dynamic analysis. It will continue by talking about a problem Philips is facing where dynamic analysis can help. Finally it will finish with the first sub question of question Q2: “What level of detail does the information need to have to be useful?”. After this, Chapter 4 moves on to the next sub question and talk about the visualization of the information discussed in Chapter 3. Some related work will be discussed and the method of visualization that has been chosen for this project. Chapter 5 will treat sub question Q2.3 and go deeper in to the measuring of data. Some related work is discussed and the sub questions Q2.3.1-2.3.3 are answered. The resulting method of the answer to the research question Q2 will be put to the test in Chapter 6. The tool and trace guidelines that have been created for Philips will first be discussed. Then there will be a section talking about how this has been applied in BeX for validation and what the results of this were. Finally there will be a section talking about an extension to the method which solves some of the problems that were discovered during validation. Chapter 7 will revisit all the important conclusions that have been drawn throughout the thesis. Next to this chapter 7 will contain some suggestions for future work.
2 Guarding the architecture
The first research question as stated in Section 1.5 aims to guard the quality of the software architecture of Philips Healthcare. The software of Philips Healthcare has been written in the programming language C#. The system is divided into several components (called units) which are allowed to communicate with each other through interfaces. In the system design, it is specified which component communicates with which other component. Philips wants to be able to verify this, preferably on a regular basis. This chapter will start with a Section 2.1 that introduces C# and some important construct used in this language. Section 2.2 will go a bit deeper into the design of Philips Healthcare and the concepts used inside this design. Next, Section 2.3 will talk about dependencies, after which Section 2.4 introduces a visualization technique called Dependency Structure Matrix (DSM). A DSM is a common means of visualizing the dependencies of a system. Section 2.5 presents the findings of the survey of three tools that use a DSM. The evaluation of the three different tools and their suitability for use at Philips Healthcare for the purpose of verifying architectural rules is presented in Section 2.6. Section 2.7 will describe the findings after the best suited tool was applied to BeX. Finally, Section 2.8 shortly reviews the findings presented in this chapter and draws a conclusion.

2.1 C#
Because the software system that has been analyzed is written in C#, it is important to understand some basic concepts used in programming language C#. The programming language C# is one of the latest mainstream object oriented programming languages. It has been developed within Microsoft and its first widely distributed implementation was released in July 2000 [5]. It is part of the .NET framework which is also developed within Microsoft. One of the goals of C# as stated in [5] is for C# to be a simple, modern, general-purpose, object-oriented programming language. It is considered a multi-paradigm programming language as it supports multiple programming paradigms, such as imperative, functional, generic, object-oriented and component-oriented programming disciplines. Originally, PII was using Java as a basis for its implementations. They switched from Java to C# when there were problems between Microsoft and Sun concerning Microsoft’s implementation of the Java Virtual Machine.

Even though in the C# specification it is not stated as a requirement, C# code is usually compiled to a special intermediate language called Common Intermediate Language (CIL). After compilation, a file called an assembly is created. This file can then be loaded into the virtual machine provided by the .NET framework which has to be installed on the target system. This virtual machine is responsible for executing the program. The idea behind this is that the only thing that has to be dependent on the target system (hardware, operating system, etc.) is the .NET framework, allowing the C# programmer to not worry about which operating system is used.

C# contains a construct called namespaces which is used to group classes together that are related to each other. Every class must belong to exactly one namespace and namespaces can be nested within one another. For a class to be able to reference another class from another namespace there must be an explicit statement at the top of the C# file stating that this class uses the other namespace. Namespaces play a big role in the design of software written in C#.
2.2 Design BeX

As mentioned earlier, BeX is built upon PII. There are three concepts that play a big role in the system:

- Components
- Units
- Layering

A component is a logical entity within the system and is used as a building block. It provides some functionality and can be used through the public interfaces it provides. A system that is based on PII will choose the components from PII it needs, tie these together (see Figure 2.1) and most likely add some custom components that provide application specific functionality.

![Figure 2.1: Components](image1.png)

This is exactly the way that BeX is build upon PII. In BeX another level of abstraction is introduced by Philips: units. A unit is a building block which is used in the architecture and consists of a group of components that work together to implement the unit’s functionality (see Figure 2.2).

![Figure 2.2: A unit containing components](image2.png)

All units and components are categorized in different layers. Software layering is a common practice in software design. The basic idea is that all components are grouped in layers. The layers have a hierarchical ordering from top to bottom. Classes in a layer can only use other classes from the same layer or classes from in a lower layer. No assumptions can be made about any higher layers. In BeX, a component can only be part of one layer, in contrast to units, which can span over the boundary of (i.e. contain components of) two layers (for an example, see Figure 2.3).
Figure 2.3: Units and components in a layered system. Components not in a unit are most likely pure PII components.

For the communication between components, two types of interfaces are defined, public and private. The following rules are set up within BeX concerning the usage of these interfaces:

- Communication through private interfaces can only happen between components within the same unit.
- All communication between components from different units must go through public interfaces.

This separation between private and public interfaces is reflected in the convention used for namespaces. All namespaces of BeX are prefixed with:

- “Philips.PmsComp.BeX.Public.Interfaces.<UnitNamespaceName>”
- “Philips.PmsComp.BeX.Private.<UnitNamespaceName>”

If a component from unit A wants to talk to a component from unit B, it will have to use an interface in the Philips.PmsComp.BeX.Public.Interfaces.UnitB namespace. In this case, unit A is considered to be dependent on unit B. When the public interface of unit B is changed, unit A will also have to be changed to use this new interface correctly. Considering this, it is clear that such dependencies are not something that should be taken lightly. If all units are dependent on all other units, changing one will force developers to change the entire system.

This is why in the BeX system design it is clearly stated which unit is allowed to depend on which other unit. Making sure that these architectural rules are not violated by checking this on a regular basis will keep the dependencies under control and increase the maintainability of the entire system.
2.3 Dependencies
As mentioned before, to perform static analysis one only looks at the source code of a software system. Many ways already exist to perform such an analysis and many different aspects of a software system can be analyzed in this way. A few examples are “number of lines of code”, “number of methods”, “fan in” and “fan out”. The research performed for this report, with respect to static analysis, was focused on finding out which tools can assist Philips in keeping track of the dependencies within their system.

A dependency can be defined as follows: item A depends on item B, if and only if a change in B can change the behavior of A. Where an item can be anything from a method to a class to a unit and behavior indicates the behavior as is specified by the static definition of the item. Often, a weight is assigned to a dependency to give an indication to how difficult it will be to remove the dependency. When trying to analyze the dependencies in a system, the definition stated above is too general. Different definitions are used by different dependency checking tools for analyzing dependencies. In Section 2.5 three different dependency analysis tools for C# are discussed. The different definitions of a dependency used in the tools will also be discussed.

The problem with dependencies is that a change in one part of a system can force another change in a dependent part. It is preferred to keep all the parts independent to keep the impact of changes small. A specific type of dependency that is almost always unwanted is a cyclic dependency. This can be defined as follows: consider all the dependencies within one system as a directed graph where the nodes are the items among which dependencies exist and the dependencies are edges between these nodes. A cycle in this graph is then considered to be a cyclic dependency. The problem with such a dependency is that if any of the items involved in the cycle are changed; all other items also have to change.

2.4 Dependency Structure Matrix
The basic idea behind a DSM is very simple. Create a matrix and for every item, for which you want to check dependencies, create a row and a column like in Figure 2.4. When an item x depends on an item y, mark the cell \((y,x)\). In Figure 2.4, item 6 has a dependency on item 2, represented by the cross in cell \((2,6)\). The benefit of this visualization is that you can easily see for a given item, which other items depend on it. Next to this, the layering of the system can also be checked using a DSM. In Figure 2.5 dependencies between components are shown. Next to the components, the layers to which the components belong are shown. In a layered system, dependencies can never go to a higher layer. If such a dependency exists, then the DSM easily shows this, because the cross representing this dependency will be above the diagonal line. This allows a developer to immediately detect these illegal dependencies.

---

1 Depending on the implementation of this technique, \((x,y)\) is also possible.
It is possible to extend the DSM by assigning a weight to a dependency and displaying this weight instead of the cross. There are several possible ways to define this weight. An example of such a definition is, when checking dependencies of components, to specify the weight as the number of times the interface of the other component is referenced. The benefit of showing this weight in the DSM is to determine how strong a dependency is. This will give an indication of how easily a dependency can be removed (e.g. if it is illegal). In [14] a more detailed description of a DSM is given and also more possibilities of what can be done with these diagrams are described.

2.5 Available tools
Wanting to test dependencies of a system is not something new and unique to Philips\(^2\). Many tools are available to perform all kinds of diagnostics on a system, both for static and dynamic analysis. When it comes to dependency checking, three tools have been found that can generate a DSM from .NET assemblies. The reason assemblies are used and not plain source code is due to the fact that all .NET DSMs have been used in the previous version of Philips’ system which is now already in production. The tool (within Philips called: “Mondriaan Views”) used to generate the DSM only works on C++, hence it could not be applied to the new system which is written in C#.
languages compile to the same type of assembly. By supporting analysis on assemblies, a tool can support programs written in any .NET language.

The first tool, named NDepend, is a commercial tool created by SMACCHIA.COM S.A.R.L. For the evaluation, only a trial version with limited functionality has been used. The second evaluated tool is .NET Reflector, which is an open source tool for which several plug-ins are available, including a DSM plug-in. Lastly, a trial version of Lattix is reviewed, which is a commercial tool specifically made for generating DSMs and reviewing your design using a DSM. An evaluation has been done to decide whether it is beneficial to acquire one of these tools or to develop one in house. This evaluation will be based on the following criteria:

- **Analyze on design level.**
  The information provided by the tool must be on design level. Where “units” (as used in the BeX design) are an important entity.

- **Reveal illegal dependencies.**
  Constraints exist with respect to dependencies to make sure the design is decoupled enough. This is the most important aspect that has to be checked with the tool.

- **Does not require much time to use.**
  It must be possible to do these checks frequently; hence the time to get the required information must not be too high. Initial set-up of the tool can take a bit more time, as long as this set-up is not required every time the tool is used.

- **Cost.**
  Is it beneficial to buy the tool, or is it better to let someone in house make it?

### 2.5.1 NDepend

There are a number of features that NDepend offers the user, all of which are connected.

- DSM
- Dependency graphs
- Tree map representation of code
- Code Query Language

The trial version introduced some restrictions, but none prevented from exploring all features. The DSM of NDepend shows dependencies twice. If item 1 depends on item 2 then in the matrix, cell (1, 2) will be colored green and cell (2, 1) is colored blue. This might seem a bit redundant at first, but this allows NDepend to show two types of weight in the DSM. In cell (1, 2), the number of sub items of item that depend on item 2 are shown. Conversely, in cell (2, 1) the number of sub items of item 2 are shown on which item 1 depends (see Figure 2.6).

The DSM provided by NDepend uses the assembly files as the main objects between which dependencies can be examined. It allows the assembly files to be expanded so the namespaces and types inside the assembly can be used for checking dependencies on that level. The problem with this approach is that the highest level on which one can check dependencies is on assembly file level.
and PII are built in a very modular way and the system is spread out over many assemblies. As can be seen in Figure 2.7, the assemblies are split up on a relatively low level (i.e. there are a lot of assemblies). Units can consist of multiple assemblies and hence this level is lower than the desired level for dependency checking.

![Figure 2.6: Dependencies between sub items of an item 1 and an item 2 and the resulting DSM in NDepend on the right (The cell with the 5 is blue and the cell with the 3 is green).](image1)

![Figure 2.7: DSM in NDepend. Blue means the entity on top uses the entity on the left and green means the reverse. In the picture, all the colored cells above the diagonal are green and the ones below are blue.](image2)

The dependency graph feature allows the inspection of dependencies between a subset of the assemblies and shows dependencies on method level. This is useful for visualizing for example a cyclic dependency or inspecting a certain dependency between two modules more closely (see Figure 2.8).
Figure 2.8: A small dependency graph showing which function from the green module (left) depends on which function of the blue module (right). Bigger block indicate a stronger dependency.

The tree map representation of the code gives an overview based on the work presented in [3], from which you can easily identify where certain elements are in your system. In combination with this view, NDepend also provides a query language (CQL, resembling SQL) that allows querying the source code for different types of metrics. The result of such a query will be displayed as a list of results, but also the corresponding parts in the tree map will be highlighted in blue as seen in Figure 2.9. This allows the user to easily specify certain design constraints and query the source code to verify whether the system complies.

Figure 2.9: A CQL query showing all methods using Private PII code. Left: a list of the results. Top-right: A treemap view showing the resulting functions in blue and a specifically selected (by hovering over it with the mouse) function in purple.

2.5.2 .NET Reflector

This tool basically only opens .NET assemblies and reverse engineers the source code from them. The power of this tool lies in its plug-ins. The plug-in that is relevant for this research is the DSM plug-in. This plug-in displays a DSM, much like NDepend. The difference with NDepend is that it allows grouping per namespace instead of per assembly. Every namespace can be expanded and collapsed separately giving the user quite some freedom on what to compare as can be seen in Figure 2.10.
Unlike NDepend, Reflector displays dependencies in a more traditional sense. The lowest level on which Reflector visualizes dependencies is classes. The weight is calculated by traversing through the class and every single reference to another class adds one to the weight of the dependency on that other class. A reference can be anything from a variable declaration in a function signature to a class inheriting from another class.

![Figure 2.10: BeX system where all Private and Public namespaces are inspected. A number in cell \((a, b)\) indicates a dependency from \(b\) to \(a\) At the bottom the use of PII private and public interfaces is visible.]

### 2.5.3 Lattix

This tool is specifically made for DSMs. It generates a DSM in the same way as .NET Reflector’s plug-in basing the hierarchy on the namespaces. Figure 2.11 shows how the tool allows the user to specify design constraints for which violations are visualized in the DSM. Another useful feature of this tool is the possibility to change the hierarchy as desired. The user can group some nodes of the hierarchy tree together and create a new common parent for them. Another possibility is to drag a node from one branch to another. Finally, Lattix can generate a high level system design diagram (see Figure 2.12), showing which subsystems reside in which layer.

A dependency in Lattix is defined as follows: “If there is a symbol defined in \(B\) that is referenced in \(A\), then \(A\) is said to depend on \(B\).” This results in the same dependencies as Reflector. The difference between Lattix and Reflector is that in Lattix it is possible to adjust how the weight of a dependency is calculated. The default setting will set a weight of 1 or 0 for every item that does not have any sub items.
as its children, no matter how often that item references another. The weight of dependencies involving higher level items is again the sum of their children.

---

**Figure 2.11:** Cells representing design violations are marked in the corner.\(^3\)

**Figure 2.12:** A generated Conceptual Architecture Diagram.\(^3\)

---

\(^3\) This image was taken from Lattix’s online tour of the tool.
2.6 Evaluation
The evaluation of the tools is based on the four criteria listed in Section 2.5.

- Analyze on design level.
- Reveal illegal dependencies.
- Does not require much time to use.
- Cost.

A summary of the tools and how they are evaluated for every criterion can be found in Table 2.1.

2.6.1 Design level analysis
To get a good look at design level of the BeX system, it is necessary to see the dependencies between units. Because .NET Reflector only shows a diagram grouped by either assembly or namespace, it is hard to get a good picture of the system at unit level. NDepend has the same problem; as it groups only at assembly level in the DSM, rendering it relatively useless for BeX due to the large number of assemblies. The CQL queries together with the tree map view do group by namespace and it is possible to make a query that will check all the namespaces of one unit at the same time. Lattix however, allows the user to specify his own subsystems. It starts with assemblies or namespaces, but everything can be moved around from there. This makes it easy to get a good view of the system at unit level.

2.6.2 Visibility of illegal dependencies
.NET Reflector does not contain a mechanism to automatically check and display illegal dependencies. Of course it is possible to see them in the diagram and find them manually. NDepend allows for automatic checking using its CQL features, the results will be visible in the tree map view, which will show the exact method where the violation occurs. Lattix also allows specification of dependency constraints allowing for automatic checking. The result is shown in the DSM.

2.6.3 Time needed for use
The first two criteria greatly influence the third. Using .NET reflector on BeX will take quite some time. The reason is that to be able to find illegal dependencies, the DSM has to be checked manually. In order to do the manual check, it is necessary to have the diagram on design level. To organize the namespaces in the diagram in such a way that the original design can be recognized takes some time. When the system has to be analyzed again after some changes have been made, all the work of arranging the namespaces is lost and has to be repeated. NDepend performs better due to the automatic checking of dependencies. After creating the CQL queries that correspond to your constraints, they can be run on any version of the software. Using lattix is also quite fast due to the automatic checking of the design constraints that can also be defined in advance. They only have to be specified once and can then be reused. Also the custom grouping that Lattix provides is preserved over different versions of the checked software. On a side note, when processing the selected .NET assemblies; Lattix is very fast and does the job within seconds while .NET Reflector and NDepend require some more time to load. This is less relevant because the user does not need to perform any actions during this time, but is still worth noting.
2.6.4 Cost

When it comes to cost, .NET Reflector is obviously the favorite because it is free. NDepend offers quite a good price starting at 300 Euros for one license to 180 per license for more than 20 licenses. The costs for Lattix are dependent on the company size. Bigger companies will need to pay a higher price than smaller ones. This is a disadvantage for Philips, due to it being a rather large company. The price for one license for Lattix is 4000 Euros, a second license 2000 euro.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Criterion</th>
<th>Design level analysis</th>
<th>Visibility of illegal dependencies</th>
<th>Time of use</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDepend</td>
<td>Namespace level</td>
<td>Automatic</td>
<td>Automatic</td>
<td>Medium</td>
<td>€180-€300/license</td>
</tr>
<tr>
<td>Lattix</td>
<td>Custom</td>
<td>Automatic</td>
<td>Automatic</td>
<td>Short</td>
<td>€3000-€4000/license</td>
</tr>
</tbody>
</table>

Table 2.1: Quick reference for Tools X Criteria

2.7 Validation of tools

Validating that using static analysis to check for illegal dependencies can benefit the BeX system is done by performing a simple analysis of the system using .NET Reflector and Lattix. The reason .NET reflector was chosen next to Lattix was that for Lattix only a trial version was available, preventing a good analysis of the whole system. The analysis using Lattix is done on a subset of the system. This subset is chosen to best represent the whole system. This subset was determined by first excluding the dependencies to PII and by removing assemblies which seem to have no dependencies to other assemblies. For an analysis of BeX’s dependencies on PII, .NET Reflector is used.

The following issues were found:

- Non documented dependencies:
  * The design document specifies which unit is allowed to depend on which other unit*
- Dependencies on private interfaces between units:
  * Units are only allowed to depend on public interfaces of other units.*
- Dependencies not going through interfaces at all:
  * Units are only allowed to depend on public interfaces of other units.*
- Dependencies on private PII interfaces:
  * PII offers public interfaces that have to be used when using PII. It is however possible to access private interfaces if a developer chooses.
2.7.1 Findings

Two dependencies were found that were not specified in the design. In Figure 2.13 the violating dependencies are marked with a triangle:

- XRayIPServic illegally depends on FEClient
- BEC illegally depends on UIDeviceService

In this case the dependencies were not the real problem. The problem was that the design document was not updated yet to include these dependencies. After discovering these problems the design document was updated.

In Figure 2.14, the public interfaces have been moved to a separate namespace. This will reveal all the dependencies that do not use public interfaces. In Figure 2.14, the public interfaces are seen in the bottom. All the dependencies in the cells above are not through one of these public interfaces and are therefore illegal.

In fact, there were several causes for these violations. Firstly, BEC was not exposing enough interfaces; hence the dependencies of XRayIPServic and Acquisition on BEC were solved by making the interfaces that were being used public. Secondly, BEC and FEClient contain generated code which caused the dependency. For this reason, an exception was made and the dependency was legalized. Thirdly, a naming error in the namespace of UIDeviceService caused some of its public interfaces to appear as private. This was easily solved by simply renaming the namespace.
Lastly, the cause of the dependencies between FieldService and UIGuidancesService and between BEC and StatusArea can be seen in Figure 2.15. By separating the private interfaces as well as the public interface, Figure 2.15 shows the dependencies that directly point to implementation instead of going through an interface. Three such dependencies exist:

- BEC to FEClient
- BEC to StatusArea
- FieldService to UIGuidanceService

BEC depending on FEClient directly is due to the generated code, which by now is already a known issue. The other two are due to a programming error and have been rerouted to use public interfaces.
Due to PII not exposing enough public interfaces, it is not uncommon for developers to use private interfaces from PII components. This can be observed in the DSM generated by .NET Reflector in Figure 2.16. The last two rows of the diagram show the dependencies on public and private interfaces respectively.

As can be seen in Figure 2.16, the problem is quite extensive as lots of private interfaces are used. The reason that PII is not exposing enough interfaces is because they know certain functionality will undergo a lot of changes in the near future. If a public interface is going to be changed, all stakeholders (i.e. Philips Healthcare business units) must first approve of the change. This process takes up time and therefore PII does not want to make interfaces public that are still going to change. Developers in BeX have two options when they want to use functionality that is still private. To solve this issue, PII is working to make more interfaces public. Until enough interfaces are public, developers are allowed to use private interfaces. They have to keep a close eye on any changes in new releases of PII to ensure that their own code does not break.
Figure 2.16: Usage of private PII code and direct dependencies between BeX units. .NET Reflector does not allow a custom hierarchy, this is why the items on the left are not (all) grouped per unit.

2.8 Conclusion

In Table 2.1 it can be seen that .NET Reflector is not a good candidate. Though being free, the tool cannot be used in an effective way on a regular basis due to a user having to rearrange all the namespaces manually every time dependencies have to be checked. NDepend is a better option, considering the cost are relatively low and it allows a user to set up queries that can be used to automatically do dependency checks. However, setting up the queries is not trivial and only works on the level of namespaces. This results in it being difficult to do the initial set up for the automatic tests. Lattix seems to be the perfect solution. It allows a custom architecture to be defined together with a specification of dependency rules. This allows for an easy initial setup and automatic dependency checking. The only downside to Lattix is that it comes with a high price tag.

Furthermore, the validation showed that the four types of dependencies that are important for Philips can be detected using the DSMs.
3 Insight into execution and timing

The second research question as stated in Section 1.5 is: “How can we get insight in the execution and timing of a use case?” In the previous chapter, the analysis of the system is considered to be static. Only the static definition of the system is analyzed (source code, assemblies, etc). To answer the second research question dynamic analysis is required. This chapter will first describe what dynamic analysis is. After this there is a section revisiting the problem Philips is experiencing with respect to this question. The section thereafter will treat the sub question Q2.1: “What level of detail does the information need to have to be useful?”.

3.1 Dynamic analysis

Static analysis can reveal a lot about a system and is a valuable source of information. Nevertheless, static analysis has limitations like revealing dynamic dependencies that are only created after the system is started. Furthermore, static analysis has no knowledge about the hardware running the software or other processes running on the same hardware, so capture all this in an accurate model is not feasible. This is where dynamic analysis comes into play. For instance, dynamic dependencies can be found when the system is actually running. Next to this, dynamic analysis can also reveal all kinds of aspects concerning resource usage. Examples of this are memory usage, network bandwidth usage as well as the speed of the system.

Nevertheless, dynamic analysis also has its downsides. As opposed to static analysis, which uses the specification (usually source code) of the system resulting in exact measurements, dynamic analysis is based on one or more runs of the system. The results of the dynamic analysis are not guaranteed to be the same the every time the same scenario is executed. Also, it is not possible to know for sure that all possible behavior of the system has been captured by only a finite number of runs.

As a result, people use dynamic analysis in some situation and static analysis in others, depending on their requirements [15][18]. Next to this, there are also efforts being made to combine static and dynamic analysis [2][7].

In the remainder of this chapter, a specific method of dynamic analysis is explained. Dynamic analysis is a wide area of research. The research described in this document falls in the category of performance analysis. To be more exact, the method described in this chapter concerns the localization of performance problems.

3.2 Current method of dealing with performance

For the medical equipment that Philips delivers, quality is an important aspect. It is important that a doctor can rely on the equipment to function as it is supposed to. Furthermore, when a doctor is performing a procedure, he should be able to respond to different situations quickly. Providing the doctor and operator of the equipment with short response times is therefore very important.

In the requirements specification, several use cases have been specified. These use cases can contain one or more steps that are performance critical. The user requirements specify a maximum duration
between the action the user takes in this step and the response of the system. An example of such a step is the time between starting x-ray and the first image appearing on the screen.

Typically, a use case in BeX’s requirements will contain only one such step; mostly at the end. For example:

- Many steps doing different settings
- User presses a button
  *The system performs some action*
- User gets feedback

The time between “Press a button” and “User gets feedback” is the critical step from a performance perspective. In this thesis, when considering a use case, only the performance relevant steps are considered.

During the testing phase the performance requirements are tested by the test team. Violating a performance requirement results in a problem report. The problem Philips is currently facing is that it may take an engineer a lot of time to pinpoint the root cause of such a performance problem.

When a unit is tested separately, the performance can be much better than when it is tested in the context of the entire subsystem. As mentioned in Section 1.5, performance problems usually surface during integration testing or tests on subsystem level. If such a test fails, a tester will issue a problem report which is assigned to an engineer. The only information the engineer has to help him find the problem is which use case has failed. This information by itself is not enough to know what is causing the problem and how to fix it.

### 3.2.1 A flood of data

After a performance problem is detected during an integration test an engineer has three means of finding and solving the problem:

- Knowledge of the system
- Profiling tools
- Debug messages (known as tracing)

The first item in the list is a very important one, but is not always enough. If the problem could be solved by just the knowledge the engineer has of the system, the problem would probably not have been introduced. Performance problems can sometimes be caused by code that no one expected to be performance sensitive. This is where the next item comes in to play: Profilers. A profiler can be described as an external tool that monitors the system during execution and measures various performance related aspects.

There are many commercial profilers available that allow an engineer to analyze a system and find performance bottle-necks, be it memory problems, inefficient algorithms, etc. The problem with these tools is that they try to serve a very general audience. As a result, the information the profiler provides is on a very low and detailed level. Common measurements are: showing memory usage, disc accesses
and CPU usage. Next to this, profilers often can give information on which methods have been called and per method, the total amount of time spent in the method. This information is very useful for smaller systems. But in a large system and applied to the whole system, the amount of information is simply too overwhelming.

Next to this flood of data, the method of measuring used by profilers influences the execution of a system. With large systems, the overhead can be too large to prevent the system from even functioning correctly (i.e. crash, hang, etc.).

The debug messages used by the engineers are the most commonly used tool at this moment for finding the performance bottle neck. These messages are generated by trace statements in the code that are placed there by engineers implementing the system. Although this information is retrieved without influencing the system too much, it may still be a large amount of data. It is difficult for an engineer to determine what is going on and therefore will take the engineer a long time to analyze the data. Figure 3.1 shows a small fragment of some of these messages stored in a trace file.

![Figure 3.1: A fragment of a trace file containing debug messages](image)

The systems Philips Healthcare is developing are of such a size that profiling the entire system is not a good option and looking at the trace files is a time consuming activity. All options provide data on a too low level. The next section will discuss a higher level view on the system that will still provide the engineers with enough information to localize the performance problem.

### 3.3 Activity diagrams

To find a suitable level of information to detect performance problems, meetings were held with several engineers working on BeX. At these meetings, it was pointed out that the design of BeX contains certain sections where performance critical use cases are decomposed. The performance sensitive steps in such a use case are split up and divided into several activities to be performed by different components (or
units). Every activity will receive a time budget based on the total time the step in the use case is allowed to take. These specifications as found in the design document are in the form of a UML activity diagram as seen in Figure 3.2.

![Figure 3.2: A performance specification of an imaginary use case: Data Calculation.](image)

If engineers can somehow be presented with information on this level, they will be able to see which activity is taking too much time. This will allow them to localize the problem to a specific unit. Furthermore, the number of activities in these diagrams usually ranges from five to ten activities per diagram. The low amount of information should make it easy enough for an engineer to understand what is going on in less time.

The method described in this thesis gives engineers measurements on the same level as these diagrams do. This will result in, for every unit, a list of activities it performed, how much time an activity took and when it performed this activity. That last piece of information (when the activity is performed) is not really represented in the diagram in Figure 3.2, but would allow an engineer to see which activities actually run in parallel and which do not.

Using this as an answer for research question Q2.1, the following two chapters will attempt to solve the remaining research questions. Chapter 4 will look into the problem of visualizing the information. The chapter after that will focus on the problem of measuring the data in an accurate feasible manner.
4 Conveying the information to the user

Now that research question Q2.1 has been answered and a suitable level of information has been determined, it is possible to try and answer research question Q2.2: “How can this information be conveyed to an engineer?”. The most important concept concerning this question is that a picture is worth a thousand words. This chapter will start by going through some methods of visualizing executions of a system as done by others in Section 4.1. After this, the visualization using bar diagrams that is used in this project will be discussed.

4.1 Related work

One result of the literature study is that no work has been found on the specific problem of the combination of performance analysis and analyzing on a higher level (i.e. the level of the design). Nevertheless, other research has been found that attempts to solve a similar problem: trying to understand the flow of the program based on an execution of the system. The research found during the study can be divided into two categories.

- Visualizing large numbers of function calls.
- Abstracting large numbers of function calls to higher level features.

Both these categories mostly base their work on the analysis of function traces. This section is split up into two subsections. Before going deeper into some of the research, first the concept of function traces will be explained in the Section 4.1.1. After this, the research related to visualizing large function traces will be explained. In chapter 5, the research concerning the abstraction of large function traces will be treated as it fits better in that context.

4.1.1 Function traces

A function trace is a list of trace items, where a trace item either represents a function entry or a function exit. Figure 4.1 shows an example of a function trace of a simple program. In the function trace, two string objects are instantiated after which the function “OutputStrings” is called that concatenates the strings and prints them to the screen. Extra information could also be present, depending on the application. For example, for detecting performance problems, a time stamp for every trace item will be recorded. Different methods exist for extracting function traces. Two of the most common will be discussed in chapter 5. The example in Figure 4.1 is of an extremely simple program. In relatively simple as for example a Sudoku solver, the function trace can already contain thousands of lines. In a subsystem as BeX this number will be much larger.
4.1.2 Visualizing large numbers of function calls

The goal of the research in this category is to take a huge amount of data and then visualize it in a way that is understandable to a user. None of the results are specifically used for performance analysis, but only for program understanding. This does not mean that it is per definition not possible to augment the results with timing information on a level similar to what was specified in chapter 3.

In [7] and [4] two visualization methods are presented called Massive Sequence View (MSV) and Hierarchical Edge Bundles (HEB) as seen in Figure 4.2. These two views are used to visualize large function traces of thousands of calls. A line in the diagrams indicates a call, where the green end of the line is connected to the caller and the red end to the callee. The massive sequence chart shows the calls ordered by time. It allows an engineer to easily identify parts of the code that call each other frequently. Next to this, it also shows outliers that might indicate strange calls that an engineer might want to inspect closer. The HEB provides a different view of the system. In the HEB one can identify aspects like subsystems that are only called (e.g. library), subsystems that mostly call others without being called (e.g. a UI), as well as calls that go in the wrong direction (red color where only green should be). Neither visualization methods used in [7] and [4] incorporate duration of calls or other timing aspects other than the order. Adding this information to the MSV will only show which function is taking a lot of time, this does not provide any more information than a standard profiler does.
The research in [15] attempts to make the trace more readable by recognizing certain types of repetition and replacing them by a more compact representation. After this compaction, a diagram similar to a message sequence chart is generated. The detection is based on four rules each representing a certain type of repetition.

- **Completely the same:**
  Detects if the same function is called many times in a row and if that function performs the same sub calls every time. It replaces this by one call with a number representing the amount of times the function is called.

- **Different objects:**
  Detects if the same function is called, but on a different object. The objects will be replaced by one object with both the names of the two objects as its name.

- **Lack of method calls:**
  Detects if two parts of the trace are similar, except that one of the parts has a few calls less. The two will be combined and the calls that are only present in one of the parts will be marked as optional.

---

4 Images taken from [7]
• Recursive calls:
  This will detect if a method is called itself and only show the outer calls after marking it as recursive.

Their method works better in some cases than in others. They have achieved results where a compacted trace was 0.85% of its original size, while in other cases the trace still had more than 7% of its original size. Furthermore, [15] claims that their method improves the program understanding by stating that in one of their case studies they were able to more easily understand a program. They do not show however; if the resulting trace gives a correct understanding of the code. Furthermore, the compaction rules used in their algorithm do not allow timing information to easily be preserved.

4.2 Bar diagrams
To create a visualization that does show the right information, some criteria have to be established first. This has been done by discussing this with an engineer at Philips. The three following criteria resulted from this. The visualization must convey the following information to an engineer:

• The time duration of every measured activity.

The total time of the activity must be immediately visible in the visualization to see which activity is the one causing the performance problem. It is important that problematic activities stand out so they can be easily spotted.

• Which unit performed which activity.

Once an activity has been spotted that takes too long, someone has to fix the problem. The subsystem BeX is divided in units. Each unit is related to one or more engineers. If through the activity, the performance problem can be traced to a unit, it will be possible to assign the problem to the right engineers that can solve it.

• Which activities work in parallel and which work in sequence (their relation to each other in time).

If two activities can be executed in parallel, the performance will most likely improve. Sometimes it is not possible for two activities to work in parallel. If two activities can run in parallel, then it is useful to be able to see this in the visualization.

4.2.1 Earlier usage of bar diagrams
Before the start of this project, another project within Philips has tried to visualize the execution and timing of the startup of one of Philips Healthcare’s systems. The visualization used in this project used a bar diagram to visualize the startup of several components. Every component goes through three phases during startup:

• Creation
• Initialization
• Activation
The diagram used in this situation showed for every component a row. In the row there would be a bar representing each of the startup phases. This bar is horizontally positioned relative to the time the related phase took place. The size of the bar is related to the duration of the phase. An example of such a diagram can be found in Figure 4.3.

![StartUp Diagram](image)

**Figure 4.3: An example of the visualization used for the startup**

### 4.2.2 Generalizing the bar diagram

To really answer research question Q2.2 and find a suitable visualization, the diagram described in the previous section can be generalized to fit this purpose. The diagram already fulfills the criteria as mentioned at the start of Section 4.2. It fulfills the criteria for the startup of the system. The startup of the system can be generalized as a use case and the phases of the startup can be considered as activities. Using this visualization, an instance of a use case (called scenario) can be visualized. Figure 4.4 shows an activity diagram and Figure 4.5 an example of a bar diagram representing a scenario of the imaginary use case “Data Calculation”. From the picture it is clearly visible that the activity “store data” performed by “database” is taking most of the time. In the activity diagram, the budget assigned to “store data” is only 400 milliseconds while it takes almost 2000 milliseconds. Considering the mentioned criteria, the bar diagram qualifies as a good means to convey the information to an engineer.
Figure 4.4: Activity diagram decomposing the imaginary use case: “Data Calculation”.

Figure 4.5: Bar diagram of a scenario “Data Calculation”.
5 Obtaining the relevant data

To be able to visualize the activities as described in the previous chapter, the required data must first be acquired. This relates to research question Q2.3: “How can the relevant information be acquired?”.

There are three fundamental aspects that play a role in answering this question. Firstly, data needs to be measured. This data can be many different things. It is important to find out what data is accessible. Then the data has to be obtained in a way such that the measurements produce accurate results. Lastly, the raw data needs to be interpreted to be in the shape of activities in unit/components as presented in chapter 3. After which it can be visualized using the bar diagrams described in chapter 4. From these three aspects, three sub questions are formulated:

Q2.3.1 What data should be obtained?
Q2.3.2 How can the data be obtained?
Q2.3.3 How can the required information be derived from the obtained data?

The purpose of this chapter is to provide answers to these questions. Section 5.1 will start with discussing a few related research projects.

5.1 Related work

In the chapter 4, the visualizations discussed in related work focused on trying to preserve all the information in a function trace and letting an engineer understand all of it. The research described in this section focuses more on removing information that is not relevant and trying to create a higher level view of the execution of the system.

The research in [18] tries to accomplish this using an algorithm that detects high level “phases” in the execution of a system. The detection algorithm is based on a function trace and finding the functions that denote the construction of new objects. To be able to determine when a new phase has started during the execution of the system the following assumption is made: “At the start of a software phase, a relatively high amount of new objects will be created.”. Under this assumption, being able to detect these concentrations of new object creations makes it possible to detect the start of a software phase. The proposed algorithm traverses the function trace in order of timestamp and uses a so called Least Recently Used (LRU) cache to store every object constructor function it finds. The size of the LRU cache must be provided as a parameter to the algorithm. When a new constructor is found and the LRU cache is full the oldest object will be removed from the queue and the new object is added. The time stamp of this removed object is compared to the time stamp of the new object. If the difference between the two is less than a specified window size (a second parameter to the algorithm) then the timestamp of the new object is stored as a start of a new phase.

The case studies described in [18] showed their technique to be effective with a precision of about 80% of the detected features. This was achieved after tweaking the two parameters of the algorithm. Two problems with this approach are stated in [18]. When concurrent user interactions with the system occur, the result might not be valid anymore. Secondly, the resulting list of phases does not contain a
After the phases have been detected, an engineer will manually have to identify which phase is which.

Finally, there is a third problem with the described approach that is relevant to the measuring of activities from the function traces. The phases described in [18] are of a quite high level. The way the algorithm is described in the paper is that it detects phases at the level of scenarios. In the case of activity diagrams, the level of a phase should be somewhat lower to match that of an activity. It might be possible to tweak the parameters to the algorithm to detect a phase on a lower level. In combination with the other two stated problems, it is likely that the accuracy of the technique will decrease.

5.1.1 Obtaining a function trace

The technique described above uses function traces as a basis. This provides an answer to research question Q3.2.1: “What data should be obtained?”. This section will explain two ways of how function traces can be obtained in an attempt to answer the next research question Q2.3.2: “How can the data be obtained?”.

The first method of measuring function traces is by instrumentation. The idea of instrumentation is relatively simple: Insert special trace statements at all the function entry and exit points. Such a statement outputs, for example, a line of text into a file. At the end of the execution of the system, the text file contains the trace and can be used for further analysis. These instrumented trace statements will generate a certain amount of overhead. The exact amount of overhead depends on a few factors:

- The method of instrumenting trace statements.
  - In the source code
  - In a compiled binary
  - At runtime by a monitoring program
- The way to process a trace statement.
  - Write line to a text file
  - Insert record into database
  - Write into memory

In general, most profiling tools generate a complete function trace (tracing every function call), which does affect performance. Some filtering is possible, for example Existing tools try to compensate for this by estimating how much time their trace statements increase the performance and subtracting this from the obtained time.

A second method for generating a trace is by sampling the call stack. The basic idea behind this technique is to sample the call stack on a regular interval. The begin of a method is then determined by when it is first seen on the stack. Conversely the end of a method is determined by the last sample in which it is seen on the stack. See Figure 5.1 for an example of this process.
A benefit of this method over instrumentation is that sampling does not affect the performance as much. The performance gain comes at the cost of precision. If a function were to start and end between one sample, this would go unnoticed. Assuming the example in Figure 5.1 represents the same program as in Figure 4.1, one of the calls to the constructor of String has been missed in Figure 5.1. This call probably happened in between either samples 1 and 2 or in between samples 2 and 3. Decreasing the interval will improve precision, while increasing the interval will improve performance.

5.2 BeX tracing

In the previous section, all the methods were based on function traces. There are two problems with this approach.

The first problem is that acquiring a function trace is a very invasive procedure. Having every function call recorded will influence the system, especially with respect to performance. Previous experience of engineers at BeX has shown that running tools that perform these kinds of measurements on the entire system (like profilers) can cause the system to break down, even when using the sampling method.

The second problem is that the function names are not necessarily related to the activity names. This will make it difficult to accurately find activities. The research in [18] shows a way to extract high level “phases” from function traces. Their method however does not handle multithreaded systems in a correct way. The activities that have to be measured can run in parallel. All these problems combined, make the method described in [18] rather unsuitable and an alternative method would has to be designed.

Another basis for the measurements will be used to solve this problem. BeX provides a generic mechanism to enable all possible tracing information. A mechanism built into the BeX system used for tracing other information is available. This system shall be referred to as BeX tracing.

The BeX system generates a trace file by default. This file does not contain traces for all function calls, but contains traces the engineer added to help him understand the flow of the program when analyzing problem reports. A trace file consists of many trace items each representing one of these debug message. There are different categories for these trace items and depending on the how the system is configured, more or less trace items will appear in the trace file. When a system is deployed at the
customer’s site, only one trace category of trace items will be traced. Basing the measurements on the trace items in this trace category will overcome the difficulty of influencing the systems performance. The influence that tracing these trace items has on the system will also be present when the system is running in the field.

5.2.1 Trace items
This section will go deeper into the format of a trace item in a trace file generated by BeX to give a better idea on this source of data. Trace items in the trace file are generated by trace statements in the source code of the system that have been put there manually by a developer. A trace item contains several fields with different information. A description of the fields in a trace item can be found in Table 5.1. In BeX, guidelines exist that force developers insert some standard trace statements in their code. An example of this is: for every call to an interface of a different unit, trace statements are inserted in the caller and the callee indicating the start and end of the call. Trace statements are also added when a developer just wants to mark a certain piece of code for debugging purposes, or when exceptions occur.

Compared to a function trace, BeX tracing contains less data and the data within is less structured (an engineer can trace whatever he thinks he needs to see in the trace file). Using this as a source of data however, makes it possible to have much more influence on what is traced.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateT</td>
<td>Timestamp denoting the time the trace statement was executed.</td>
</tr>
<tr>
<td>ClassName</td>
<td>The class in which the trace statement is located.</td>
</tr>
<tr>
<td>Component</td>
<td>The component that executed the trace statement.</td>
</tr>
<tr>
<td>Description</td>
<td>A textual description of the trace item made by the developer.</td>
</tr>
<tr>
<td>Severity</td>
<td>Determines the category of tracing.</td>
</tr>
<tr>
<td>ThreadID</td>
<td>The thread id of the thread executing the trace statement.</td>
</tr>
<tr>
<td>ThreadName</td>
<td>The name of the thread executing the trace statement.</td>
</tr>
</tbody>
</table>

Table 5.1: Fields of a trace item.

5.3 Interpreting the data
To be able to get the right information out of the trace files, the data must somehow be interpreted in another way than as described in [18]. There a few options that one can choose from to accomplish this.

- Automatically interpret the trace
- Instrument the code
- A combination of both

This section will shortly treat the three approaches and name some benefits and drawbacks leading to a selection of one approach.

The first approach makes the assumption that the description field of the trace items present in the trace file gives an indication to what the system is currently doing. The problem with this approach is not that this assumption is 100% false, but a developer can put anything he wants in the description field making it very hard to analyze. To analyze such data, a complex algorithm would have to be developed which will most likely not produce an accurate result.
The second option makes it much easier to interpret the trace files. If the trace file contains a trace item indicating the start and end of an activity, then the algorithm will only have to match these to get the final data. The difficulty in this approach is to get the trace statements in the right place in the code. Letting developers do this themselves creates extra work and the accuracy depends on whether or not the trace statement is correctly inserted into the code.

The third option combines instrumentation with automatically interpreting the code. The idea behind this option is to first instrument the code using a simple algorithm. This algorithm would for example add trace statements at the entry and exit point of a specific type of function (e.g. calls between interfaces of components or units, etc.). This will result in a sort of high level function trace. Based on this trace another algorithm will then derive which activities have been performed.

The option that was chosen for the first prototype was the second option: instrumentation. The downside of this approach is the manual labor that a developer will have to perform to instrument the code. In the case that all functions need to receive a trace statement at their entry and exit point, this would indeed be way too much work. The reason that it is possible to live with this drawback is because the amount of inserted trace statements is relatively low. Only inserting two trace statements (begin and end) for every activity, is not at all a lot of work, especially if the code is already somewhat known to the responsible engineer. Well structured code will make the process easier. For example, functions with multiple exit point can be a cause for trace statements to be accidentally skipped because the responsible engineer overlooked one possible code path. The amount of activities will never become too large because the whole purpose of this project is to get a high level view of the system. This means that there should not be too many activities.

5.3.1 Analysis algorithm

After the code is instrumented the resulting trace file needs to be parsed to extract the data. Algorithm 5.1: ParseData is used to accomplish this. Algorithm 5.1: Parse contains several calls to sub methods, which have been omitted to improve readability. Appendix A.1 contains pseudo code for all the sub methods except the methods extracting the name of a scenario, unit or activity and the methods determining the type of a trace item (i.e. is it a begin or end of a scenario or activity).

Algorithm 5.1: ParseData will process all trace items in the order of their timestamp. Based on the instrumented trace items it finds, it will return a set of scenarios. A scenario will be linked to a set of units and every unit will be linked to a set of activities. Each of these three entities will have a name, a begin time and an end time linked to them. Figure 5.2 shows this structure in a diagram.

![Figure 5.2: Structure of data returned by Algorithm 5.1: Parse](image-url)
The algorithm makes the following assumptions:

- **All scenarios occur in sequence (i.e. no two scenarios overlap).**
  Making this assumption simplifies the algorithm considerably. It is plausible to make this assumption because in BeX the defined scenarios are never supposed to run in parallel.

- **If the begin of an activity is traced during one scenario, its end is traced during the same scenario.**
  It is possible that due to an error this will not be the case. But to simplify the description of the algorithm, this border case is ignored.

- **Every begin scenario has a matching end scenario.**
  Again, it is possible that due to an error this will not be the case. But to simplify the description of the algorithm, this border case is ignored.

- **No two activities executed by the same unit have the same name.**
  It is necessary to make this assumption to be able to identify an activity uniquely by the combination of the unit executing it and its name.

---

**ParseData**

*T*: List of trace items sorted on time stamp.

1. \( S \leftarrow \emptyset \): set of scenarios
2. \( s \leftarrow \text{undefined} \): current scenario
3. \( U \leftarrow \emptyset \): set of unit names of active units
4. \( A \leftarrow \emptyset \): set of pairs of unit and activity names of active activities
5. **For Each** \( t \in T \) **Do**
6.   
7.   **If** TraceMatchesScenarioBegin\((t)\) **Then**
8.     **If** \( s \neq \text{undefined} \) **Then**
9.       **CloseScenario**\((S, s, t, U, A)\)
10.     **End**
11.   **End**
12.  \( U \leftarrow \emptyset \)
13.  \( A \leftarrow \emptyset \)
14. **Else If** TraceMatchesScenarioEnd\((t)\) **Then**
15.   **CloseScenario**\((S, s, t, U, A)\)
16. **Else If** \( s \neq \text{undefined} \) **Then**
17.  \( un \leftarrow \text{RegisterUnitTrace}(U, t) \): Registers which unit is active and returns its name.
18.  \( \text{CheckActivityMatch}(A, t, un) \): Registers an activity begin/end if \( t \) marks this.
19. **End**
20. **End**
21. **Return** \( S \)

---

**Algorithm 5.1: ParseData**
5.4 Conclusion

The three sub questions that have been answered in this chapter together provide the answer for question Q2.3 “How can the relevant information be acquired?”.

The three sub questions and their answer will be repeated here shortly:

**Q2.3.1 What data should be obtained?**
Trace files that contain the results of instrumented trace statements denoting the start and end of both scenarios and activities.

**Q2.3.2 How can the data be obtained?**
BeX contains a standard mechanism to trace any information. The instrumented trace statements use this mechanism to output the result into a trace file.

**Q2.3.3 How can the required information be derived from the obtained data?**
By scanning the trace files for the instrumented trace statements, the start and end of both activities and scenarios can be determined.
6 Validation

After having answered the three sub questions of research question Q2 in Sections 3 through 5, the resulting method of performance analysis can be put to the test. The method has been applied in the field to be able to know if the method described in Section 5 is actually effective in solving the problems Philips is facing. This chapter will start with Section 6.1 explaining what kind of tool is required to make it possible apply the proposed technique and the prototype that has been developed. In Section 6.2, the guidelines will be discussed that were issued to engineers to help them instrument their code. Section 6.3 will describe how the method has been applied in BeX and what the results of this were.

6.1 Tooling

When combining the instrumenting of the code and the visualization using bar diagrams from the previous section, it should be possible to get a good understandable view of the system. However, to actually use this technique in a real life case, it is necessary to develop tool support to analyze the trace and visualize the result. Next to this, the necessary trace statements will have to be added to the code. In case of a large team, guidelines have to be created to ensure that everyone who is working on the code knows where to add which trace statement.

A prototype has been developed to analyze the trace files and to visualize the results. The analysis is done on the trace files of the BeX system. It identifies the instrumented trace items and derives for which scenario which activity was performed by which unit. This section will describe this prototype that has been built to validate the proposed method of performance analysis.

6.1.1 Design

The architecture of the prototype consists of four main components. Figure 6.1 shows the architecture of the system and flow of information between the main components. In the following subsections each of the separate components will be described.
The component Measuring is responsible for extracting the data required. The component needs to make sure that the measurements are in the right format. If in the future a different source of data is selected other than the instrumented trace statements, then only the Measuring component should be adapted.

In the implementation used for the prototype, the data is acquired by traversing a trace file (text) generated by the system using the algorithm described in Section 5.3.1. This trace file contains trace items which have been generated by trace statements in the source code. Every trace item has several fields and is represented in the trace file by a line of text where every field is separated by a tab. The most significant fields in a trace item are:

- Date time: a timestamp of the moment the trace statement was executed.
- Component name: the component that executed the trace statement.
- Description: a textual description or message passed to the trace statement as a parameter.

The algorithm scans for instrumented traces and uses these fields to identify the scenarios and activities.

The visualization component obtains a list of scenarios that have been found by Measuring. It is responsible for generating the visualization. In the current the design, the Visualization component uses Excel to visualize the diagrams. Excel is used because it provides the right functionality to draw the diagrams and is easy to use. The Visualization component creates an Excel file containing a bar diagram that is formatted in the right way to represent the visualization described in Chapter 4.

### 6.1.2 Performance

To be able to use the tool effectively for validation, it must not work too slowly. There are two actions the tool performance that can take a lot of time.
- Parsing the trace files searching for scenarios
- Generating the Excel files containing the bar diagrams

For the first item, the analysis algorithm described in 5.3.1 determines the speed. The asymptotic running time of the algorithm is $O(n)$, where $n$ is the amount of trace items to be analyzed. In a test run on an input of about 50000 trace items, the algorithm was able to finish with approximately two seconds. The trace files that will be analyzed for the validation will contain considerably less trace statements.

The output of the analysis contains a relatively small amount of data (scenarios, units/component and activities). The most determining factor for the second item is the exporting of the diagrams to Excel. Formatting the bar diagrams correctly is a relatively slow. Visualizing 15 bar diagrams with a total of about 450 activities takes one minute and twenty seconds. For the purpose of validating the proposed analysis method, this time is still acceptable.

6.2 Guidelines

For the algorithm to be able to analyze a trace file, the correct trace items need to be present in this file. The engineers responsible for implementing an activity are responsible for inserting the right trace statements into their code. To make it possible for developers to consistently add the trace statements to their code, it is necessary to set up clear guidelines. These guidelines must tell an engineer the following things:

- Requirements with respect to the tool
- Requirements with respect to understandability

It is important that the input abides by these assumptions, because the algorithm the tool uses makes certain assumptions.

- All scenarios occur in sequence (i.e. no two scenarios overlap).
- If the begin of an activity is traced during one scenario, its end is traced during the same scenario.
- Every begin scenario has a matching end scenario.
- No two activities executed by the same unit/component have the same name.

Furthermore engineers must understand when to qualify something as an activity. If a developer defines an action that is performed 1000 times in a loop as an activity, then the visualization will become completely cluttered with 1000 activities.

The following guidelines have been created for engineers to follow to accommodate this:

With respect to scenarios:

1. For each scenario a begin/end trace statement should be added to the code. This will allow the algorithm to exactly know when a scenario starts and ends.
2. Scenarios cannot be nested or overlapping. 
   This will allow the algorithm to more easily know to which scenario to attribute discovered 
   activities.
3. Scenarios need to have a unique use case name. 
   This way, two scenarios of the same use case can be linked through this name.
4. The resulting trace statement needs to have its description in the following format:
   a. Prefixed with $> SCENARIO or $< SCENARIO for the begin and end respectively.
   b. Separated from the prefix with a space, should be the unique name for the scenario
      This allows the algorithm to identify the instrumented trace items and extract the 
      scenario name

With respect to activities:

1. For each Activity a begin/end trace statement should be added to the code. 
   This will allow the algorithm to exactly know when an activity is starts/ends.
2. An activity’s name needs to be unique within a component. 
   Multiple instances of the same activity should be seen as one activity in order to keep the level of 
   the measurements high enough. This will also avoid problems with matching begin and end trace 
   items.
3. An activity’s begin and end should be traced in between a scenario begin and end (i.e. activities 
   should be nested in scenarios). 
   The activity will be attributed to the scenario during which it is executed.
4. The resulting trace statement needs to have its description in the following format:
   c. Prefixed with $> SCENARIO_ACTIVITY or $< SCENARIO_ACTIVITY for the begin and end 
      respectively.
   d. Separated from the prefix with a space, should be the unique name for the scenario
      This allows the algorithm to identify the instrumented trace items and extract the 
      activity name

6.3 Application to BeX
The performance analysis method proposed in this thesis has been applied in BeX to test whether it 
works. There are a few aspects that are important when it comes to determining whether the proposed 
method is effective or not.

• Usability
  This refers to the time it takes for an engineer to instrument his code. If it takes hours to 
  instrument the code for one use case, the method is not usable.
• Understandability
  When an engineer sees the diagrams, he should be able to locate the problem faster than using 
  the old method where he would just scroll through the trace files.

Five performance sensitive use cases were selected for the case study. These use cases represent very 
common actions for which performance requirements have been defined.

• System start up
• Start image acquisition (prepare)
To be able to test how effective the performance analysis method is, trace statements have been added to the BeX code. Two engineers that work on the implementation of these use cases were responsible for adding the trace statements into the source code. The prototype described in Section 6.1 has been used to generate bar diagrams of several scenarios of the above mentioned use cases. These bar diagrams were taken back to the engineers for evaluation.

The results of every use case will be treated separately in short in Sections 6.3.1 until 6.3.5. A bar diagram is shown in each section together with a description of the use case and the findings.

6.3.1 Start up
The start up of the system is a use case that is relevant every time the x-ray system is used for a procedure. Not wasting time starting up the system can increase the amount of patients that can be treated in one day.

There are a few main processes that have to start. Every process performs at least the following activities: it is first created, then registered and finally activated. Next to these standard activities, it is
possible for a process to have additional activities, like “StartSession” for the PMW process as seen in Figure 6.2.

When it comes to usability, adding traces for this scenario was not applicable here because tracing marking the begin and end of these activities already existed. With a small adjustment, these traces could be re-used for the experiment. Furthermore, it is immediately clear that two activities take more than 50% of the total time.

### 6.3.2 Prepare

The prepare use case is executed when the doctor starts the x-ray and images have to be shown on the display. This again is a very critical use case, due to the x-rays going through the patient.

This use case contains a total of seven activities. Adding the trace statements for all these activities cost about half an hour. Furthermore, Figure 6.3 shows one activity that is taking up most of the time. Next to this activity there is one other activity worth noting. The rest of the activities are so small compared to “IP Prepare For Acquisition” that they are barely visible in the diagram. Lastly, there is a time gap of almost 800 ms where the diagram is not showing any unit as active.

![Figure 6.3: Measurements of a prepare scenario.](image)

### 6.3.3 Unprepare

This use case is the opposite of prepare. After the doctor switches the x-ray off BeX will switch to a mode where the doctor can examine the collected data.

In this use case there are four activities. Adding the trace statements took less than half an hour. Figure 6.4 is very similar to Figure 6.3 in that one activity is really taking up all the time. Next to this, there is again a time gap in about the same location.
6.3.4 Case selection

When a patient is selected from the patient data base, some default data is loaded and the system is brought into a state where it can easily access the patient’s data.

This use case contains 14 activities. Figure 6.8 shows two activities that immediately attract attention as being large activities. Adding trace statements for this use case took about an hour.
6.3.5 Protocol selection

Before the x-ray is started, it is possible to select different settings. When selecting a set of settings, they have to be loaded into the system.

The use case Protocol Selection contains nine activities. It again took about half an hour to instrument the code for this use case. The activity that takes up most of the space in Figure 6.6 is “IP Prepare For Viewing”.

![Protocol Selection Diagram]

Figure 6.6: Measurements of a protocol selection scenario.

6.3.6 Observations

This section will discuss the results of the validation. The results are in the form of feedback from the two engineers who looked at the generated diagrams after having added tracing to their code.

First of all, adding trace statements to the code was a relatively easy process. It takes about half an hour on average to add all the trace statements in the code for a use case of about 10 activities. This means that when it comes to usability, the method can easily be applied to a large system.

Secondly, with respect to the understandability of the diagrams, problematic activities are easy to spot. The bar diagram gave the engineers an intuitive overview of the execution of the system. Where manually scrolling through trace files could be very time consuming, the bar diagram instantly draws attention to activities that cost a lot of time.
Next to the positive feedback, there were also some problems. The instrumenting of the code is a manual process, meaning that it is possible for a human to make a mistake. On a few occasions an activity would not appear in the diagram because a trace statement was not inserted correctly. The two main reasons for this were:

- A trace statement was placed in a code path that sometimes is not executed.
- The scenario end trace statement was executed before the last activity end trace statement was executed (for example due to multithreading).

An example of this is the activity “IP Prepare For Acquisition” in Figure 6.3. In some measurements, that activity would not be there.

Despite of the problems mentioned above, the performance analysis method has turned out to be a vast improvement. The next section will discuss another problem that was not discovered immediately after the first experiments.

### 6.4 Trace counting

At the time that this research project was executed, BeX was still under construction. New code was being added and existing code refactored. After the initial validation tests, weekly measurements were scheduled to be performed for the same use cases. In these tests a new problem was discovered.

- New activities were introduced in the use case which did not show up in the initial activity diagram in the design. This results in gaps in the bar diagram.
- Functionality is delegated to another unit, for example: Unit A performs activity 1, but waits until unit B is finished with activity 2 before finishing activity 1. This will result in two parallel bars in the bar diagram. Only one of those bars is representing actual work being done.

Figure 6.3 shows such a time gap in the prepare scenario and Figure 6.7 shows an example of a scenario containing both these problems. To try and solve these issues, an extension to the prototype was made. This extension will be described in Section 6.4.1.
6.4.1 Tracing to indicate work

Both problems stated in the start of Section 6.4 can be summarized as:

- Not showing a unit as performing an activity when it is doing work.
- Showing a unit as performing an activity when it is not doing work.

To solve these problems, an extension to the method is proposed that tries to identify whether a unit is really working or not. This extension is based on the other (unrelated to this project) trace items in the trace file from which the activities are extracted. These unrelated trace items represent information that engineers use to analyze the system. They originate from trace statements in the code that have been added by an engineer for various reasons. The number of trace statements in a given piece of code can vary. Some piece of code will contain a lot more trace statements than others. It is however possible to assume the following: “When a unit is generating a lot of trace items it is likely working. When it is not generating many trace statements, it is likely not working.”. If the information of the amount of trace items if added into the diagram, then an engineer will be able to determine where the actual work is being done. He can then fix the problem by insert the right trace statements in the right position.

6.4.2 Extending the prototype

To try out this new extension, the prototype has been modified to count the trace items per unit. It will count the number of trace items detected during a certain activity; it will count the number of trace items in an interval between activities (and before the first and after the last activity); and, for units without activities, it will count the number of trace items between the first and the last trace item detected (for the corresponding unit). See Figure 6.8 to see the difference between a diagram with and without trace counting.
6.4.3 Results

To verify whether the assumption made in 6.3.1 is correct and this method works, measurements were done on a scenario that showed a gap. Figure 6.9 shows a result of a measurement of the prepare scenario. It is clear that there is a gap of about 2000ms in the scenario. At first glance, one might speculate that XRayIPService is performing an activity that is not included in the instrumentation. However, the result in Figure 6.10 shows that another unit called Viewing has traced 94 trace items exactly in the gap. This would indicate that the unit “Viewing” is actually performing an activity. After checking with the engineers, this was indeed confirmed to be correct.
One drawback of relying on the extra trace items in the trace file is that it is not a reliable source of data. The amount of trace items depends heavily on how many trace statements the engineer has added to his code. If an engineer adds a trace statement in a loop, then it is possible that in a short time span, a large amount of trace items can be generated for a low amount of work. Vice versa, when a programmer does not like adding trace statements, then it cannot be determined that work is being done by looking at the trace items. For this reason, the trace counting can only provide an engineer with an indication as to what might be the cause of anomalies in a diagram (like a gap).
7 Conclusions
Two types of analysis have been performed on the BeX subsystem, static and dynamic. For both analyses, there have been some results. The results for each type of analysis will be discussed in this chapter in Sections 7.1 and 7.2 respectively. Section 7.3 provides some ideas for improvements and extensions to the research presented in the thesis.

7.1 Static Analysis
The research question that has been solved by doing static analysis is as follows: “How can the software architecture be guarded using dependency checking?”

First, it has been determined that dependencies within the system were to be the aspect of the architecture to be examined. Three tools for dependency checking have been evaluated and the best candidate has been applied on BeX. A few conclusions can be drawn from this.

First of all, of the three dependency checking tools that have been examined, Lattix meets the needs of Philips the most because it allows grouping of classes on a custom defined level. The price however is quite high, yet the cost of a decaying architecture probably outweighs this.

Secondly, The DSM technique has proven itself to be effective in BeX. It has successfully uncovered a number of problems in the architecture of the system. Suitable solutions have been found for the problems, except for the fact of private interfaces of PII being used. This is however noted by Philips. PII will attempt to make more interfaces public while BeX will attempt to limit the use of private interfaces.

7.2 Dynamic Analysis
In the dynamic part of this project research question Q2 was central: “How can we get insight in the execution and timing of a use case?”. Three sub questions were formulated to split the problem up into smaller pieces:

Q2.1 What level of detail does the information need to have to be useful?
Q2.2 How can this information be conveyed to a developer?
Q2.3 How can the relevant information be acquired?

7.2.1 What level of detail does the information need to have to be useful?
When a test indicates that a use case is taking too much time, engineers had to look at tracing or use a profiler to find out where the problem lies. BeX is a large system and these two methods of analysis provide information on a very low level. Before having to resort to looking at tracing or the use of profilers, an engineer should first be able to localize the problem to a smaller part of the system. The view that the proposed method gives shows an engineer which unit is doing what activity at what time. After knowing the activity that is taking too much time and which unit performs that activity, an engineer can look at the tracing or use a profiler to analyze only that part of the system. This level of detail will be very useful to an engineer.
7.2.2 How can this information be conveyed to an engineer?
The data that is of the level as described in Section 7.2.1 contains the right information for an engineer to find a performance problem. However, the information still needs to be displayed to the user in an efficient manner. Research question Q2.2 covers this problem. The method chosen for this was to plot the obtained data of an activity and a scenario on a timeline in a bar diagram grouping the activities per unit. This choice was based on an earlier project that used a similar visualization for analyzing startup performance. This visualization has shown to be effective in the validation as later explained in Section 7.2.4.

7.2.3 How can the relevant information be obtained?
The data had to be acquired somehow and the last research question covers this. It was important to find out what data can be obtained. Obtaining function traces would influence the system too much and was therefore not chosen. Manually instrumented trace statements however would generate only a few trace items making it possible to obtain this data without influencing the system.

To obtain this data, the already existing tracing framework of Philips was used. This means that all the activities can easily be traced. Lastly, deriving the required information from the obtained data was relatively simple. The information traced contains exact start and end times for all activities. Therefore using the BeX tracing to manually instrument the code is the answer to this last research question.

7.2.4 Validation
The validation performed at Philips Healthcare on the BeX subsystem confirmed that the method was indeed effective. The method for analysis described in this thesis was applied on five performance critical use cases. Engineers added tracing to their code and afterwards examined the resulting bar diagrams.

Firstly, it can be concluded that adding the tracing to the code is about half an hour of work per use case. This means that it is acceptable to let engineers manually instrument the code.

Secondly, engineers could immediately see in the bar diagram which activities were taking the most time. If performance of a certain use case needed to be improved, it was immediately clear which part of the code to look at.

7.2.5 Extension
Even though the positive results presented in 7.2.4, there were cases that a bar diagram would show a certain unclear situation.

To solve the problem that sometimes there would be a time gap in the generated diagram or sometimes it would seem that two units were performing the same activity an extension to the method is proposed. The extension counts the number of trace statements a subsystem executes in a certain period of time. The assumption is made that when a subsystem is active, it will produce more trace items in the trace file.

Validation of the extension showed that in the diagram in which there was a time gap it was now clear which unit was performing an unknown activity. However, the number of trace statements in a piece of
code is also very dependent on how it is implemented. This extension can therefore only give an indication as to where work is being done and is not 100% accurate.

7.3 Future work
This section contains ideas to continue research in this area. Sections 7.3.1 until 7.3.3 suggest some improvements to the performance analysis method. In Section 7.3.4 two recommendations are done with respect to the application of the performance analysis method in Philips Healthcare.

7.3.1 Augmenting the diagrams
The extension that added trace counts to the bar diagram, helped solve problems in the tracing. Adding other information into the diagram might be useful in the same way, or even give an extra hint to the cause of the performance problem. Some examples of data that could be added are:

- Disk usage over time
- CPU usage over time

7.3.2 Statistical analysis
At this moment, the bar diagram represents one scenario. Running a large number of scenarios of the same use case and analyzing the result using statistics might provide a more accurate result. A simple example could be: Run a use case 100 times and then calculate averages for the starts and ends of every activity and visualize the result based on these averages.

7.3.3 Interactive diagrams
At this moment, a bar diagram is just one image. If one activity is taking a lot of time, an engineer might like to zoom in on that activity before turning to a profiler. The activity could then be split up into sub activities. Care needs to be taken that the workload of instrumenting the code will not increase drastically due to the extra trace statements required for all the sub activities.

7.3.4 Within Philips Healthcare
For Philips to use the performance method more effectively the prototype developed for this research project needs to be replace by a full fletched tool. The guidelines need to be issued to all engineers work on BeX.

Furthermore, the performance analysis does not have to be restricted to BeX only. The tracing mechanism used in BeX is provided by PII. Every other software system in Philips Healthcare that is based on PII, can adopt the performance method with the same ease as BeX.
8 Bibliography


Appendix A  Pseudo code

A.1 Trace parsing algorithm

ParseData($T$)
$T$: List of trace items sorted on time stamp.

1. $S \leftarrow \emptyset$: set of scenarios
2. $s \leftarrow$ undefined: current scenario
3. $U \leftarrow \emptyset$: set of unit names of active units
4. $A \leftarrow \emptyset$: set of pairs of unit and activity names of active activities
5. **For Each** $t \in T$ **Do**
6. **If** TraceMatchesScenarioBegin($t$) **Then**
   7. **If** $s \neq$ undefined **Then**
   8. CloseScenario($S, s, t, U, A$)
   9. **End**
10. $s \leftarrow$ StartNewScenario($t$)
11. $U \leftarrow \emptyset$
12. $A \leftarrow \emptyset$
13. **Else If** TraceMatchesScenarioEnd($t$) **Then**
14. CloseScenario($S, s, t, U, A$)
15. **Else If** $s \neq$ undefined **Then**
16. $un \leftarrow$ RegisterUnitTrace($U, t$): Registers which unit is active and return its name.
17. CheckActivityMatch($A, t, un$): Registers an activity begin/end if $t$ marks this.
18. **End**
19. **End**
20. **Return** $S$

StartNewScenario($t$)
$t$: Trace item denoting the start of a scenario

1. $s \leftarrow$ new scenario
2. Name[$s$] $\leftarrow$ ExtractScenarioName($t$)
3. Begin[$s$] $\leftarrow$ TimeStamp[$t$]
4. **Return** $s$

CloseScenario($S, s, t, U, A$)
$S$: Set of all found scenarios
$s$: Currently active scenario
$t$: Trace item denoting the end of $s$
\( U \): Set of all active units found during \( s \)
\( A \): Set of all activities found during \( s \)

1. \( \text{Units}[s] \leftarrow U \)
2. \( \text{Activities}[s] \leftarrow A \)
3. \( \text{End}[s] \leftarrow \text{TimeStamp}[t] \)
4. \( S \leftarrow S \cup \{s\} \)
5. \( s \leftarrow \text{undefined} \)

**RegisterUnitTrace**\( (U, t) \)
\( U \): Set of unit names of all active units found
\( t \): Trace item

1. \( \text{un} \leftarrow \text{ExtractUnitName}(t) \)
2. \( \text{If} \ \text{un} \notin U \text{ Then} \)
3. \( U \leftarrow U \cup \{\text{un}\} \)
4. \( \text{Begin}[\text{un}] \leftarrow \text{TimeStamp}[t] \)
5. \( \text{End} \)
6. \( \text{End}[\text{un}] \leftarrow \text{TimeStamp}[t] \)
7. \( \text{Return} \ \text{un} \)

**CheckActivityMatch**\( (A, t, \text{un}) \)
\( A \): Set of pairs of all active unit and activity names of active activities
\( t \): Trace item
\( \text{un} \): name of the unit that generated \( t \)

1. \( \text{an} \leftarrow \text{ExtractActivityName}(t) \)
2. \( \text{If} \ \text{TraceMatchesActivityBegin}(t) \text{ Then} \)
3. \( A \leftarrow A \cup \{(\text{un}, \text{an})\} \)
4. \( \text{Begin}((\text{un}, \text{an})) \leftarrow \text{TimeStamp}[t] \)
5. \( \text{Else If} \ \text{TraceMatchesActivityEnd}(t) \text{ Then} \)
6. \( \text{End}((\text{un}, \text{an})) \leftarrow \text{TimeStamp}[t] \)
7. \( \text{End} \)