Dead code elimination for web applications written in dynamic languages

Master's Thesis

Hidde Boomsma

Dead code elimination for web applications written in dynamic languages

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Hidde Boomsma
born in Naarden 1984, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Department of Software Engineering
Hostnet B.V.
Amsterdam, the Netherlands
www.hostnet.nl

Cover picture: Hoster the Hostnet mascot

Dead code elimination for web applications written in dynamic languages

Author: Hidde Boomsma
Student id: 1174371
Email: H.B.Boomsma@student.tudelft.nl

Abstract

Dead code is source code that is not necessary for the correct execution of an application. Dead code is a result of software ageing. It is a threat for maintainability and should therefore be removed.

Many organizations in the web domain have the problem that their software grows and demands increasingly more effort to maintain, test, check out and deploy. Old features often remain in the software, because their dependencies are not obvious from the software documentation.

Dead code can be found by collecting the set of code that is used and subtract this set from the set of all code. Collecting the set can be done statically or dynamically. Web applications are often written in dynamic languages. For dynamic languages dynamic analysis suits best. From the maintainability perspective a dynamic analysis is preferred over static analysis because it is able to detect reachable but unused code.

In this thesis, we develop and evaluate techniques and tools to support software engineering with dead code identification and elimination for dynamic languages. The language used for evaluation is PHP, one of the most commonly used languages in web development. We demonstrate how and to which extent the techniques and tools proposed can support software developers at Hostnet, a Dutch web hosting organization.

Thesis Committee:

Chair:	prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	dr. phil. H.-G. Gross, Faculty EEMCS, TU Delft
Company supervisor:	ir. S.R. Lenselink, Hostnet B.V.
Committee Member:	dr. S.O. Dulman, Faculty EEMCS, TU Delft

Preface

This master research is done for Hostnet B.V.[1] from Amsterdam, the Netherlands. Hostnet is a hosting provider active since 1999 and offers domain names in almost all available top level domain names, shared hosting, virtual private and dedicated server hosting. The company has about 165,000 customers and sold approximately 600,000 domain names. Currently Hostnet is the no. 1 seller of the Dutch .nl domains.

Hostnet has it's own software engineering department which takes care of developing all internal systems for the provision of the sold domains and hosting as well as the customer relation management, the web store and web site. All the graphic design is done by the design department. PHP[2] is used to build these complex web applications. Within the company the need rose to delete old program code from the application to make it easier to maintain and upgrade the software, reducing the total cost of ownership.

I would like to thank the members of the Hostnet software engineering team for their cooperation and open attitude towards the project and H.-G. Gross for his counselling and pleasant collaboration. Also a word of gratitude for my lovely girlfriend who had to practice a lot of patience when I was working on my thesis and had little time. Last but not least I thank my parents for their support and caring.

Hidde Boomsma
Amsterdam, the Netherlands
April 16, 2012

[1] `http://www.hostnet.nl`
[2] `http://www.php.net`

Contents

List of Figures

Chapter 1

Introduction

Dead code is source code that is not necessary for the correct execution of an application. It is a result of software ageing [34, 21]. Dead code is a threat for maintainability and therefore should be removed. Dead code leads to big applications and applications with a lot of source are known to suffer from their size when it comes to maintainability and adding new features [21, 22, 30]. This can be explained because engineers need to understand an application before functionality can be added in the right place. When the programmer is not aware of the dead code, a piece of code that was written years ago and containing unknown bugs could be resurrected. When porting to new versions of used frameworks or libraries effort put in dead code is obviously wasted, therefore it would be beneficial to know which code is dead and remove it.

Besides maintainability dead code has also implications on the performance and resource hunger of an application. If dead code is included in an executable it will cost more disk space and will occupy more memory. Dead instructions can lead to cache misses, lowering performance. Executing useless code also adds up to the total execution time.

Removing dead code requires huge effort [1, 28] therefore a good balance between cost of removal and a perfectly clean source base should be found [38]. Making the identification and elimination process more efficient enables us to deliver cleaner code in the same time and give a higher return on investment.

Many organizations in the web domain have the problem that their software grows and demands increasingly more effort to maintain, test, fetch from the Version Control System (VCS) and deploy. Old features often remain in the software, because their dependencies are not obvious from the software documentation.

More and more applications are written for the web using domain specific scripting languages such as PHP. As these applications grow more complex the need for quality assurance and software maintenance tools increases. Web applications are in constant motion and are deployed rather than distributed, as is the case with regular applications. This makes them very volatile and dynamic. Many web applications make use of dynamic languages such as PHP. These languages are easy to learn and well suited for rapid development of initial ideas.

A web application differs from a conventional application in the way it is deployed and how the user interacts with the application. A conventional program is installed on

the users computer and ran on that hardware where a web application is deployed to the company servers and the users connect to it via the web browser. Everything in a web application is done via requests to the server. Each request is handled by a separate thread or process which is terminated or reused after an answer to the request has been send to the browser. It is important to keep the execution time of every request very short because the user will be waiting for the server. When implementing dynamic measuring this should be taken into account. Because a web application request is handled fairly isolated, code that was wrongly removed will not crash the whole system but only show an error page to the user. The user can push the back button, report the problem and continue working. This is different from conventional applications where all work could be lost when the application crashes. This implies that there is not a very big penalty when optimistically removing code that is thought to be dead.

Dead code elimination got a lot of attention regarding performance optimization, but not that much with respect to software maintenance. Techniques proposed in literature, usually perform static analysis to detect code that could not possibly be executed. It often concerns optimizations at a compiler level. These optimizations are not applied to the source code and thus do not increase maintainability. There also exist methods to detect dead methods and functions [2, 39] which could improve maintainability. Removing dead functions from a dynamic language with static analysis however is very difficult because of dynamic features like run-time source inclusion, dynamic and weak typing, "duck-typed" objects, implicit object and array creation, run-time aliasing, reflection and closures [6, 7, 5, 15, 42].

While static analysis is quite difficult, dynamic analysis is not very hard, because the dynamic nature of the language can be used to conduct measurements without altering the code in many cases. For a static language, static analysis would provide certainty, but in a dynamic environment this is no longer the case. Dynamic analysis, however, will never result in 100% certainty, but it allows the detection of functionality that exists but is not used any more. From a maintainability perspective this is useful information because maintaining unused code is a waste of effort. Therefore we will use dynamic analysis of the applications at hand to detect dead code. There are two types of dynamic analyses possible, coverage and frequency analysis [3]. The first only captures if a statement, function, method or class is used at all and the second would also measure how often it is used. We have to decide which information will be useful to determine if a piece of code is dead.

The examples used in this thesis are based on the work performed in the case studies at Hostnet. Hostnet is a Dutch web hosting company with its own software engineering department responsible for the internal infrastructure which enables provisioning, offers customer relations management, the web shop, the customer portal and website. Because Hostnet uses PHP for most of its applications, PHP is used for code examples in this thesis.

Problem statement and research questions

The problem of increasing maintenance cost because of the growth and ageing of the software leads to the problem statement. How to identify and eliminate dead code in dynamic languages in general, and PHP as such language, in particular.

We identified the following research questions in this context:

- **Which data from the software is necessary in order to perform dead code identification and removal?**

  Which granularity is needed to provide useful information to the user? Do we need frequency or just coverage information? In short Which information is needed. This fundamental question is discussed in chapter 2 on Dead code identification.

- **How can we extract the necessary data from the software?** How is it possible to obtain the required information from the web application?

  The actual implementation of the system, answering this question on how to fetch the information is presented in chapter 3 (Implementation of dead code identification).

- **How should the data be presented to the software developers?** In which way should the date be shown? Should we use graphs, tables or other representations? Could the representation be integrated with the Integrated Development Environment (IDE) the software developers use?

- **What is the overhead incurred of dynamically extracting the data from the software?** The overhead of the measurement should be small enough to be unnoticed by the end-user of the systems. What is the impact of the overhead and how does it influence the system and server load. The overhead measured in the system will be evaluated in chapter 6 (Evaluation).

- **What is a good strategy for deciding which code to remove?** How long should we wait before we assume that a piece of code will not be ran in the future and can be safely removed? This will be discussed in chapter 5 on Dead code elimination.

Contributions

This thesis contributes not only to the body of knowledge about dead code identification and elimination from web systems programmed in dynamic languages but it shows that the proposed method is feasible for use in a business environment. This has been shown via multiple use cases for applications that are heavily in use at Hostnet. A method to detect which classes are in use, without altering the code is devised. The method can handle a constantly changing application as is often the case for web applications.

To test the method a set of tools has been developed. A plugin for the Eclipse IDE, a web application that can be used to visualize the data and a Command Line Interface (CLI) toolbox that uses the same library as the web application and is capable of collecting and transforming usage data.

These software packages are made open source at github[1] under the Hostnet profile.

[1] `https://github.com/hostnet`

Chapter 2

Dead code identification

This chapter will explain the different types of dead code and discuss where dead code comes from. In the introduction it was already established that dead code leads to a bigger code base which presents an issue for maintenance of the software and therefore should be avoided or removed. After the terminology is clarified the chapter will focus on the granularity of the identified dead code needed to achieve the goal of lower software maintenance cost. Next the different analysis methods and their implications will be discussed.

2.1 Dead code

The following terminology for dead code is used in literature. Knoop [31] refers to code that is unnecessarily executed when speaking of dead code. Chen [12] calls all code that is not executed on any program path dead code. Janota [26] calls this code unreachable

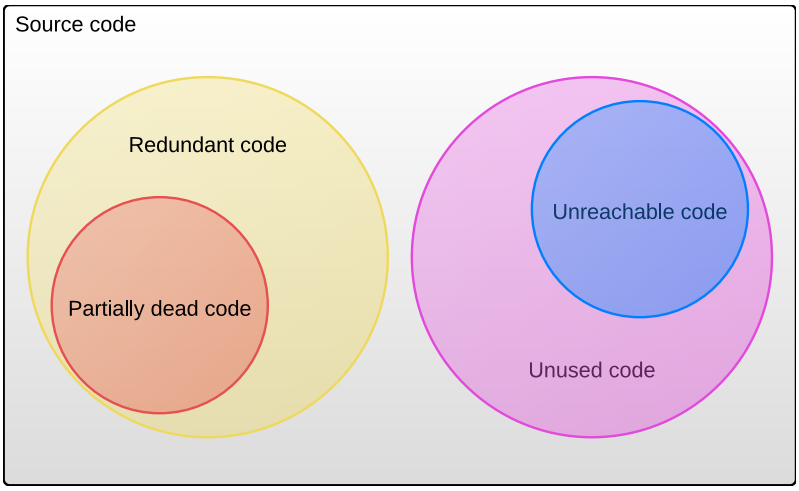

Figure 2.1: Dead code classification, in this thesis dead code is refers to unreachable and unused code

code and code that is unnecessarily executed redundant code, both subclasses of dead code. Knoop [31] writes about partial dead code, which is code that is only dead on some program path.

If looking at dead code we can classify the following sets of dead code: The set of redundant code, code that is executed but does not change the outcome of the program. Partial dead code is a subset of redundant code because it is executed on some paths. Next we have code that is never executed. In this thesis we call this unused code. Some unused code could be executed but never was because the user never used a function to trigger execution of the unused part of the code. Unreachable code would be a subset of unused code. For unreachable code we use the definition of Janota [26], code that is never executed on any program path. This classification can be seen in Figure 2.1.

When the term dead code is used in this thesis it refers to unused code, including unreachable code because this is the code we are interested in from a maintenance point of view. From a performance perspective this would not be a problem as long as the source files are not included in the final distribution as is often the case for static languages because the compiler can optimize them out. Redundant code is not included because it can not be determined with dynamic analysis. When we need to be specific the terms partial dead code, redundant code, unused code and unreachable code will be used.

2.2 Causes of dead code

Dead code is not written on purpose, but still almost all applications possess dead code. The following items cause the existence of dead code:

Unclear or incomplete specifications Unclear specifications lead to dead code when parts of code are only ran when certain conditions are met. If those conditions are never met in practice because such a case will never arise, those pieces of code are written for nothing due to unclear or incomplete specification.

Software ageing Programming and design mistakes are always made, also in the initial writing of a program. But most arise when changing the software to add new functionality or repair bugs. Changing a program and adding new features will often lead to decay of the structure of an application [34]. Which in turn will lead to more dead code because programmers are always weary about removing code when they do not have a clear view of the whole program. Dead code because of unused features is a common pattern when software ages. The engineers do not know that certain features are abandoned or simply do not dare remove them [34, 38]. When it is known that features are not used any more removing them is not a trivial matter because other parts of the application may have common code with the unused feature. A deep knowledge about the application is required to do so.

Code reuse Reuse or reuseability of the software also introduces dead code into an application because not all interfaces and functions will be used [33, 41].

Modelling objects With the introduction of the Object-Oriented programming paradigm it is more likely to model an object and introduce unused methods than with pure procedural code which is only written on demand [39].

2.3 Necessary data for dead code identification

The first thing to know is what should be measured and to which extent, should statements, methods, classes, components, files or something different be used as smallest unit of dead code, in other words, what will the granularity of the dead code that will be recorded be.

Source code can be analysed with different levels of granularity. For example, it is possible to discuss code on a statement, method, class and file level. Before we decide on which data we need to gather for identifying dead code, we should choose what granularity best suits the cause of enhancing software maintainability. The lower the granularity the more information we have available to determine which code is dead. A lower granularity leads to finding more dead code, but also implies more overhead and more data to progress. The overhead should be kept to a minimum and preferable no code should be changed to do the measuring. It is questionable to which extend maintainability can be improved when removing very small pieces of code. Because eliminating dead code needs a lot of effort [1, 28] it is beneficial to start with a large granularity and make it smaller only when there is enough room for improvement left because otherwise too much effort is spend at too little gain. The granularity of dead code identification is a trade-off between overhead and more specific information. We chose the file as smallest unit of dead code to identify. This is part of a top-down approach, first remove dead code at file level and if there is still room for improvement without too much overhead or effort use a smaller unit of dead code, for example a function to get more detailed information.

Because it is impossible to measure dynamically which classes are not used, we measure which classes are used and then take the difference with the set of all classes available in the application. This implies that we can determine which files are dead only after all used files are known. This means we have to wait until all files that are still in use are expected to be executed. Following conventions used in most languages classes relate to files. Thus if it would be possible to measure included files this would also provide data on a class level. With dynamic analysis it is possible to collect coverage and frequency [3] data. As bare minimum, covarage data is needed. However to measure how the number of classes in use develops we need more data. Therefore the first moment in time a class is used should be recorded. It might also be possible that some classes will die within the measurement period. To be able to detect this we also record the last moment in time a class was used. If this date is too far in the past the class can be reckoned dead. Classes that are added to the distribution but still are under development are not likely to be in use and could be mistaken for dead files, it is possible to save the last date a file was changed in the VCS to solve this problem. To have insight in how many things could be broken when wrongly removing classes from a module, the total number of uses of a class can be saved. This implies a basic form of frequency analysis. This leads to the following list of data to be recorded: number of times used, first time used, last time used, last time edited and changed repository.

2.4 Data extraction methods

The data could be acquired by both static and dynamic analysis. But because we look at applications written in dynamic languages and are interested in unused features dynamic analysis is used.

The data collected should be aggregated directly because storing the raw data would give far too much data to store. Storing raw data could use 1 gibibyte per day for just logging all used files. For example storing 4KiB data $\times$ 4 requests per second $\times$ 3600 seconds per minute $\times$ 24 hours per day = 1.318 GiB per day (using binary prefixes conforming to [25]). When measuring applications for multiple months this is simply too expensive. The 4 requests per second is the average that Aurora, the application that will be used for the main use case, has during the day. The 4KiB is based on the query size used to update the table in a test set-up for Aurora.

When an application runs on multiple servers or has shared files with other applications the storage should be done on a shared location. All applications send their data to the central repository whereupon the visualizations can be build.

There are multiple options to record which files are used by the software. The most generic one is to write a library function which is called from every file upon inclusion of the file. This method can easily be extended to methods, functions and blocks of code. The disadvantage is that adding the function calls can take quite some time if it has to be done by hand. It is possible to automatically add the function calls to the files, but it is easy to break things this way. Some scripts may require specific lines to be the first or last in a file which troubles the automatic addition of function calls. When the code base is altered it is very hard to turn everything off and have no overhead any more because all files have to be changed again. Another possibility is to place logging code in the class loading mechanism if the language supports this. This method is preferred over placing function calls in the code because you only have to change the code at one place. The downside is that it is less generic because it can only be used for classes that are dynamically loaded. It is possible to log both classes as well as files that are in use. If the languages provides a garbage collector that can be controlled it is possible to record the elements that were garbage collected. This method has the disadvantage that the objects that are not garbage collected are not added to the set of alive code. If the garbage collector collects all objects for the application this could be overcome by calling the garbage collector one last time before exiting the application. The PHP garbage collector can not handle cyclic dependencies [35], which makes this method not suitable for our use cases.

It is possible that the language at hand offers native support to get a list of all included files, modules or classes. In that case this is possible we do not have to worry about the previously described problems. If a function like this is available, this is the preferred method to use. PHP offers this functionality.

As described in the section about necessary data for dead code identification, not only data from the dynamic analysis should be added to the central storage used for the dead code identification but also data from the VCS should be added. Most VCSs can give you the date a file is last edited. This data can be collected and put in the repository as soon as the set of files that will be deployed is known. If possible this should be integrated into the

deployment process so that every time that new files are deployed to the production server also new VCS data is added to the central dead code storage.

To be able to get from files that are in use by the application to the files that could be dead we need also the set of all files available. This can be achieved by recursively listing the directory contents of the deployed project. In the case of PHP we would only take the `*.php` files into account because we are not interested in non executable files. It is possible to get all available files just before calculating which files are dead. If the file names are put in the database upfront it is possible to always select which files are used and which are not. This also adds the possibility to create an index of all the files so that the usage information can be processed quickly instead of having to add new files and re-index when more files are used. The performance will stay the same over time when all files are added to the index upfront. Performance will only decrease when more files are added to the application and the index.

Besides collecting all used files, this data also has to be send to the database at some time. Because we are working with web applications the process length is limited to one request from the web browser. This leads to the simple decision to send the file names of the used files to the database at the end of each request. This way a minimum number of requests to the database has to be made.

Only recording which code is dead is not enough, the dead code also has to be visualized in a way that is easy for the user to understand and easy to extend when more features or data sets are needed. Therefore the main visualization is build in PHP and javascript as web application. To prevent developers using (possible) dead files by mistake those files should preferably be marked in the IDE. More on the visualization can be found in the chapter on visualization (chapter 4)

2.5 Approach used in this thesis

For this thesis we use the term dead code for code that is not executed [12], this includes unused and unreachable [26] code. Redundant code [26] and partial dead code [31] is not included. Dead code is caused by software ageing [34, 38], code reuse [33, 41], object modelling [39] and unclear specifications. For performance and cost of maintenance [1, 28] the dead files will be identified. If a more fine grained approach is needed this can be done on the files not yet removed. This is left as future work. The data needed to draw conclusions consists of the path of the file, how often it was used and when it was used for the first and last time. VCS data concerning when the file was last modified is also added to prevent removing new features that are already in the production system but not activated. The dead files can be detected by recording which files are in use and subtract those from all files. To detect the files in use multiple methods consist. The best one is to rely on a language feature that gives all included files, if this is not possible the class loader or garbage collector could be modified and as a last resort calls to a logging function can be added to all files. Because of the amount of data that will be collected everything has to be aggregated right from the start. If the applications are deployed to multiple production servers a centralized storage for the usage data of the files is necessary.

The tools to do this should be very modular so multiple ways of input are possible and the visualizations can easily be reused. The toolbox should allow easy porting to other languages and still be able to use the parts that, for example, acquire VCS data.

Chapter 3

Implementation of dead code identification

In this chapter the toolbox that is created to be able to perform the case studies is discussed. Before it is possible to look at the implementation directly the differences and properties of the web applications under inspection are looked upon. The visualization of the data can be found in the next chapter (chapter 4).

3.1 Web applications in PHP

The main differences between a web application written in PHP and a conventional application is that the web application is in memory for a very short period and a conventional application resides in memory for the time the users is using it. This means that the application is only active when the user clicks a link or submits data. This could go via a normal HTTP request or through AJAX [19]. In both cases the browser sends a request to the web server which will spawn a PHP process to handle the request and return the result to the browser. When a PHP process crashes, the web page shown to the user will be a 500 error page but the application as a whole will still function. Only just submitted data will be lost, and in most modern browsers pushing the back button will preserve even that data.

Due to the short lived nature of web applications it is possible to just store the aggregated data at the end of every execution. If a page crashes no valuable statistical data will be lost; code that is used by a page that crashes could be interesting to have for debugging purposes but it is of no value for the code in use because the feature clearly does not work. It is worth noting that Zend Server[1] has functionality to save a full code trace of a crashed page. This could also be done for every page but this will give far too much data to process because the API does not offer usable aggregation or coverage options. This has to be done afterwards. For a small application this could be possible but for a busy site this will use up the memory.

In a web application the overhead can be measured in the amount of additional time used per page load. The overhead relative to the total length of the page load is of not much importance. Only the time that is additional used for a page is important because if the

[1] `http://www.zend.com/en/products/server`

overhead is too big it will be noticed by the user. For example if a page is very quick and does not have any database interaction the overhead will be really big relative to the total length of execution, but the end user would still not notice this because the load time is still short enough to go unnoticed. This is only true for short lived scripts showing direct output to the user, for a long(er) lived batch job which, for example, can be found in the provision system and which are used to create the invoices. For these scripts the overhead relative to the total execution time is important because the periodic tasks could take too long with the dynamic analysis enabled. The goal is to have the overhead go unnoticed by the end users.

3.2 Dynamic alive file identification in PHP

PHP offers functionality to retrieve all included files from a running process. To know all files used, this function should be called after the normal execution has finished. This function loads all files that are included and does not know if those files are actually used, so this method only provides used files if they are dynamically loaded and are not statically included. The function call `array get_included_files(void)` gives an array with all file names.

Since version 5, PHP has the ability to automatically load classes. This feature is also used by Symfony[2] the framework used by Hostnet [36]; it is also used by almost any other framework for PHP. With PHP it is possible to create a function that will be called when a unknown class is instantiated. That function may then take care of loading a file containing the class definition so it will become available before the execution is continued. If the auto load function did not load the needed class definition for the unknown class PHP will fail with an error [35].

Now we have the method that will give a list of all used files, but those file names still have to be saved in a central database. This has to be done after all normal execution. To accomplish this we could add the call to the `*.php` files that are accessed by the users and web server but PHP offers the possibility to automatically prepend and append files to each and every PHP execution no matter if it is executed through the CLI, Common Gateway Interface (CGI) or as Apache module. This property can be set in the `php.ini`, the global configuration file of PHP, with the `auto_append_file` directive[3]. This option can also be set via a `.htaccess` file which controls access properties for underlying folders in the Apache web server. This could prove useful to only turn on the analysis for the web server or for some specific directories. At Hostnet we set this property system wide via the `php.ini`. This has the effect that the setting is applied machine wide. So even batch jobs that are started by hand will be taken into account. All logic that sends the used files that have been measured for a request to the central database can be put into a single file. In this implementation this file is called `append.php` (see Appendix A for an example implementation). This file is then set to be automatically be appended to every execution of PHP. This files also takes care of the aggregation of the data. The dates in the database

[2] `http://www.symfony-project.com`

[3] `http://php.net/ini.core.php#ini.auto-append-file`

about when the file was last used and first used are updated when needed and the usage counter is increased.

MySQL is used as storage solution because it was already available as service at Hostnet and is a very common companion of PHP and Apache. A central MySQL server eases the analysis of applications deployed on multiple servers and allows to run all visualizations without file transfers. The current implemented solution makes uses of PHP Data Objects (PDO) so the toolbox can easily be integrated with other storage solutions like SQLite[4] which does not requires any extra software or drivers to be installed or set-up. This saves the user the trouble of setting up a MySQL server for the sole purpose of analysing one application.

To add files to the database, a simple listing of the directory structure is used. Only `.php` files are added to the database. Whenever an application is deployed to the production server a listing is made to be able to add new files to the database. Files that are not longer in the application will not be removed from the databased but tagged with the date when the removal was detected because with this historic date it is possible to visualize the state of the system in any moment in time since the start of measuring. Every application is primed when a new version of the application is deployed. The deployment of a new application is as simple as an automated Subversion (SVN) checkout on the production server and a new symbolic link to the right folder. After the checkout some (post) configuration is done by the Makefile[5] in the project. This way other configuration files are loaded for the development, testing, acceptance, staging and production environment are loaded. Via the deployment Make file the `prime` subcommand of the toolbox (see appendix C.3) is called.

For every automated deployment system it will suffice to add a call to the `prime` subcommand of the toolbox to the process. If there is no automated process this command should be ran by hand every time new files are added or removed from an application when it is deployed.

As described in the previous chapter also VCS data should be added to the database. This is implemented in the `prime` subcommand that is also used to add the file names to the database as option. Using the `--vcs svn` flag the command will also parse the SVN data of the files. Other VCSs can easily be added, most of the work for Git has already been done.

Overview of the implemented tools

An overview of the process can be viewed in Figure 3.1. All available files are loaded from disk together with the VCS data if needed and available. This information is stored in a database and refreshed when a new version of the application is deployed. This database is updated by the `append.php` file adding information about when the file is last used and how many times it is used. The aggregation by `append.php` is done on a file level. From this database containing all raw data it is possible to create graphs showing how the number of used files changes over time, but this data can also be aggregated over the directory structure to provide the percentage of dead code per directory and file. This information

[4] `http://www.sqlite.org`

[5] `http://www.gnu.org/software/make/manual/make.html`

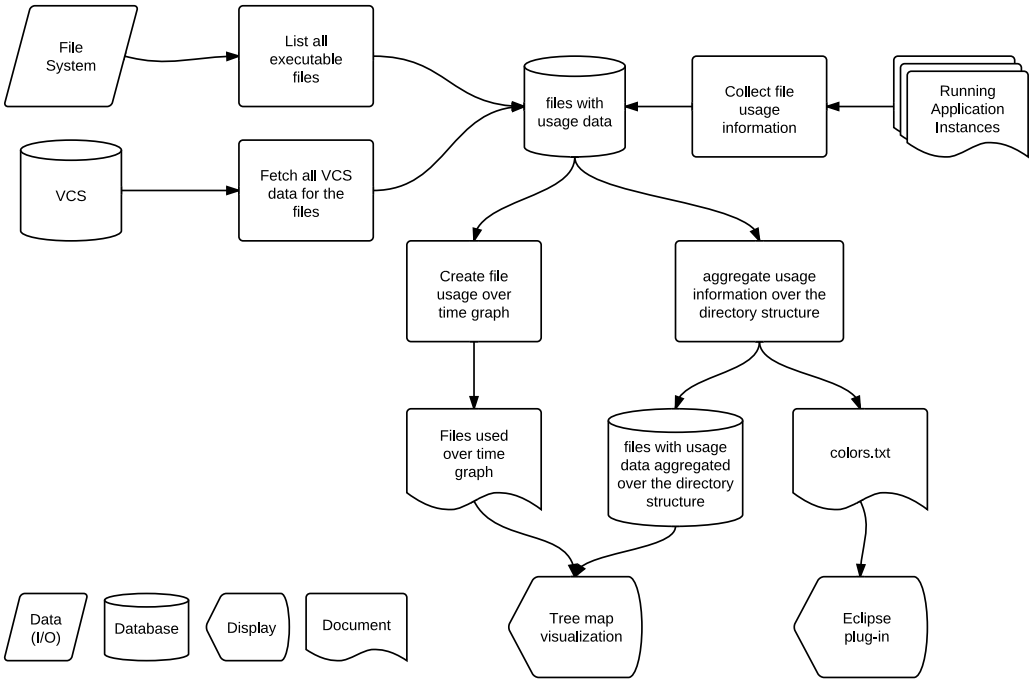

Figure 3.1: overview of the dead code identification process

can be used for the plug-in. This aggregated data is also used for the tree map visualization, for this visualization the data is cached in a separate database. The tree map visualization also includes the graph. More on these visualizations can be found in chapter 4.

Chapter 4

Visualization

The data that is collected should also be visualized so the user is able to see and use it. The data is collected by the process described in chapter 3. The data collected is stored in a relational database and can also be collected by other methods than the one described in this thesis. This allows reuse of the visualizations. Figure 3.1 at the end of the previous chapter shows how the visualization is decoupled from the information gathering part that stores all data in the central databases showed at the top of the figure.

Which data should be available to the user to be able to make well informed decisions is discussed in the chapter on dead code identification (chapter 2). All this data is visualized in a web application because a web application is multi client and cross platform by nature. Besides the web application there also is an plug-in for the Eclipse IDE. This plug-in displays only the percentage of potential dead files and not all information the web application offers. The Eclipse plug-in is meant to make the developers aware of the possible dead code as soon as possible. Users that do not use Eclipse can always use the web application visualization.

4.1 Tree map

The web application uses a tree map [27] (the red and green boxes shown in Figure 4.1) as main visualization method besides a simple sortable table listing the percentage of unused files, the number of unused files, the total number of files and the number of inclusions for the application. The tree map shows the directory structure of the application. The visualization is build up using the directory structure already present on disk to group the files together. This way files that are closely coupled are likely to be in the same (sub) directory, for example modules or plug-ins can be viewed as a whole this way. Because it is easy to spot we have decided to display all files that have been executed green and other files red. Folders get a gradient between green and red depending on the percentage of executed files in the directory. To make it easy to spot which directories only contain executed files or do not contain any executed files these extremes have a distinct green and red color from the gradient start and stop color. The colors and gradient used is visible in Figure 4.1 between the tree map and the table.

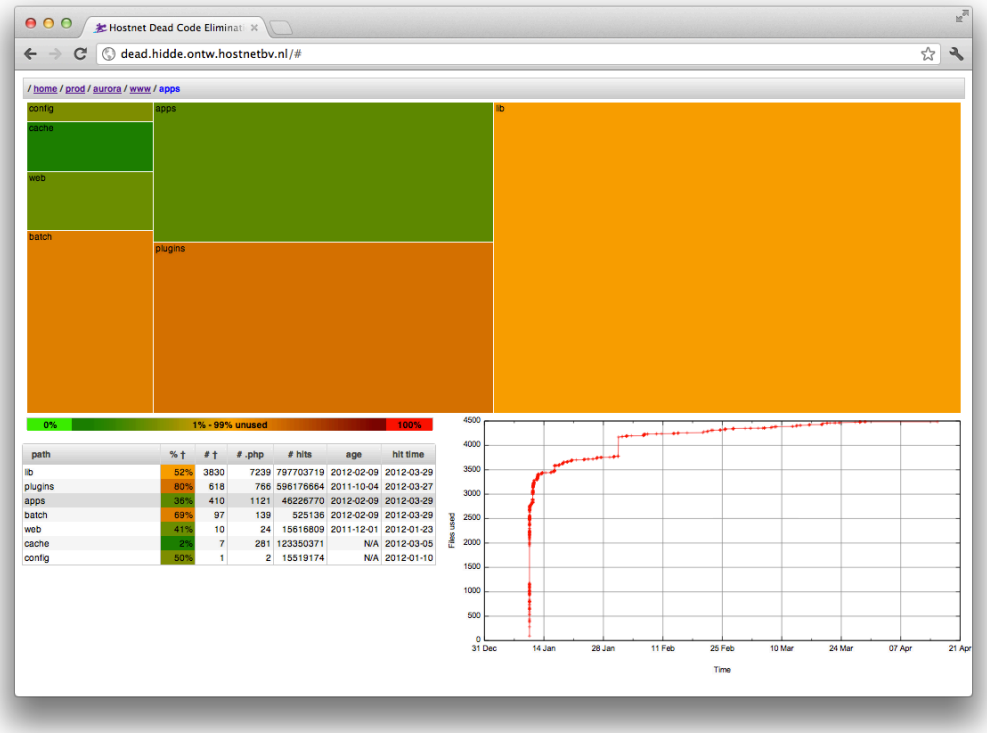

Figure 4.1: An example of the tree map visualization

By clicking the boxes in the tree map it is possible to open the clicked directory and view the contents in the same way it was viewed for the parent. This way it is possible to navigate trough the directory structure and view the distribution of dead files over the subdirectories for each directory. As navigational aid a breadcrumb path [24, 32, 37] is shown just above of the tree map. This is the path to the directory that is currently displayed. The directories in the breadcrumb path can be clicked to open them. At the end of the breadcrumb path the directory or file name of the current selected item is displayed. This is convenient to see which directory will be opened when the name is too long to be displayed in the box in the tree map. By right clicking the tree map it is possible to navigate back to the parent folder.

The size of the boxes in the tree map corresponds with the number of dead files contained within the folder to point the user towards the area where the most unused files can be found, the more dead files, the bigger the box will be. A square scale is used so that big folders with many unused files will not take up all space but the difference in size is still distinct enough to be noticed.

When more detailed information is needed it can be fount in the table on the left bottom It possible to sort the list according to each column by clicking on the header. The colors used in the table are the same as those in the tree map. The rows in the table are linked to

the boxes of the tree map when hovering the box in the tree map will be highlighted and vice versa. The following columns are available:

% † Percentage of unused files in this folder. For a file this is either 0% or 100% depending on the state of the file. This number is useful to be able to quickly determine if all files in a folder are not executed until now and probably can be removed.

† The number of unused files in this folder. For a file this is either 0 or 1 depending on the state of the file. The more unused files are in a directory the more sense it makes to invest time in this part of the system.

 # .php The total number of PHP files contained in this folder. For a single file this is always 1. This column can be used to sort the table according to where to project has the most files. This is usable to get a quick overview of the dead code distribution throughout the project and finding caused of dead code like machine generated code that is never used.

 # hits The number of times this file is included since the beginning of measurement. For folders the hits of all contained files are aggregated. The hit count for a directory can be used to determine the certainty of the data displayed. If the files in a directory are only used rarely and over time new files are being included (visible in the first hit column and graph). Hence it is probable that the files that are not used yet will be used in the future. So extra care has to be taken when removing files in such an area of the project.

age Date at which the file is last changed. For a folder the most recent date of all contained files is displayed. It is possible that files are just added to the application for a new feature that is not enabled yet. When this is the case the file will be recently changed in the VCS. If the last changed date in this column is very near the file will probably not be dead even if it is not used yet.

hit time Date at which this file is included for the first time. For a folder the most recent date of all contained files is displayed. When all files in a folder have very close first hit dates it probably is less probable that the remaining files will be accessed than when all dates are very far apart.

The graph on the bottom right shows the number of unique files used in total over time. The crosses denote the moment when a file was used for the first time. The last data point in the graph is not an actual new inclusion of a file, it is added to plot the graph until the current date (or end of measuring).

These visualizations aid the software engineer when making decisions about when to remove a certain part of the software. It also helps determining where the most benefit could be reached when the goal is to remove all or some portion of the dead code.

This visualization is build using using the d3.js[1] javascript library. This library offers many visualization methods along with the tree map. By using this library it is possible

[1] `http://mbostock.github.com/d3`

to add other types of visualization of the same data without having to rewrite the whole visualization. The data that is needed for the tree map is provided by a small script that is using the toolbox as library. To view the intermediate format the subcommand `json` of the toolbox can also be run on the command line.

The data displayed in the tree map is fetched from an hourly updated table in the central database which is build by aggregating the data in the measurement table. This is done via the `tree` subcommand (see appendix C.4). This command of the toolbox aggregates the number of dead files over the directory structure and stores the number of dead files contained in every directory together with the other fields described above in the description of the table columns.

4.2 Integration into Eclipse

To aid the developer working with the IDE, there is a plugin to use a (configurable) color scheme like the one in the tree map as colors for the files in the Eclipse[2] environment. By default the plugin looks for the `colors.txt` file in the project root directory. If it is found all the projects files matching those in the file will be coloured according to the percentage in the file. An example of such a file can be seen in appendix D. The color files for the Hostnet projects are created hourly from the database and saved on a network share accessible by all developers. This way all developers have a fresh view on which files are used in production. The update of the color file happens with the `color` subcommand from the toolbox (see appendix C.2).

[2] `http://www.eclipse.org`

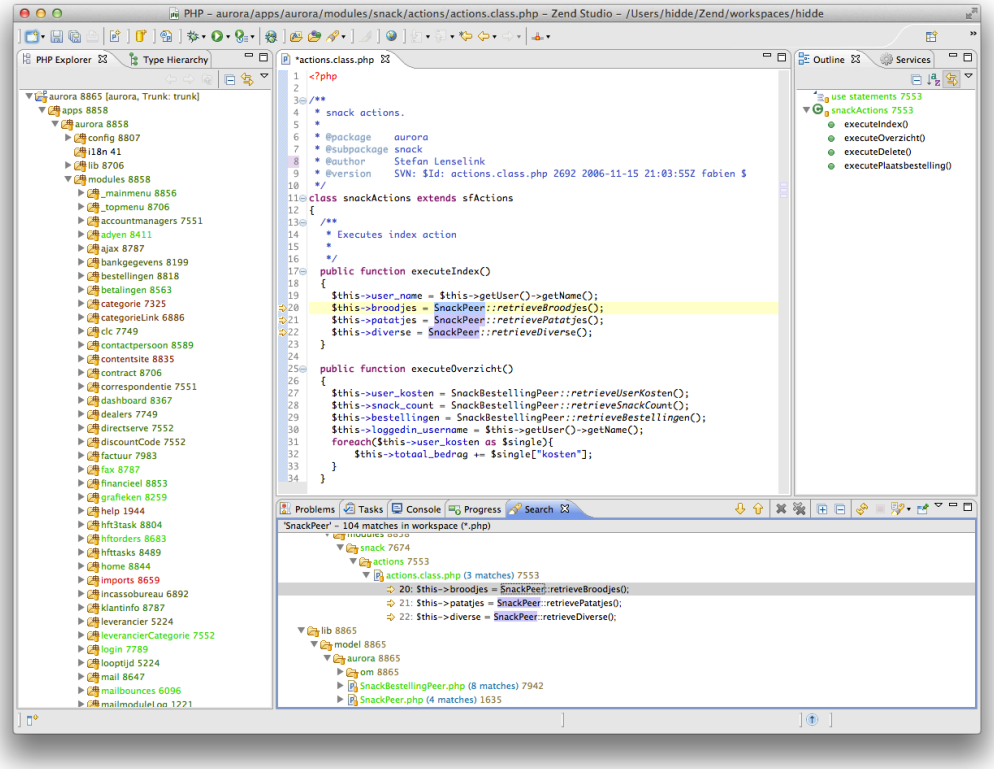

Figure 4.2: Eclipse with the dead files color decorator plugin loaded

Chapter 5

Dead code elimination

At this point we have identified code that possibly is dead. We know that used code is not dead, but we do not know whether unused code is not dead. Before we can declare code dead, we should have an idea about the usage frequency of the code. For example the login page of a system is expected to be used every day, the invoicing system and related files are expected to be used once a month and VAT reports are only used once a year. Knowledge about how much a feature is expected to be used is very important in making the decision if files should be declared dead.

5.1 Decide which code to remove

When cleaning an application, the first place to start is at the tree map (see figure 4.1). The tree map shows per directory the number and the percentage of dead files. The size of each box corresponds to the number of dead files in that directory on a square root scale. The square root scale makes sure that directories with little or no dead files are still visible but at the same time spot which directories contain the most potentially dead files. The color of the boxes corresponds to the percentage of dead files contained in the directory. See chapter 4 for a more elaborate description of the tree map.

The tree map can be used to find files clustered together in a folder that are all dead. Files grouped together in a directory are usually related to each other. If a whole folder with related files is dead it offers a starting point for removal. This method can also be used to detect unused plug-ins if they are located in the directory tree. An example can be seen in figure 5.1. Modules with only dead files can probably be removed if they are not expected to run with a frequency bigger than the period the usage data was collected in. Modules that are only partially dead require more inspection. Here the age of the files can help. If there are two files for that likely perform the same task and one of them is dead and the other is not, this could indicate that a feature was updated and the old file was not removed. The file indicated as potentially dead should not be recently edited in this case. For this case the graph that shows how many files are executed over time for the directory can be consulted. If the graph is stable for a long period and almost all files are accessed at the same time, it probably is less likely that the other files will be accessed in the future. The number of

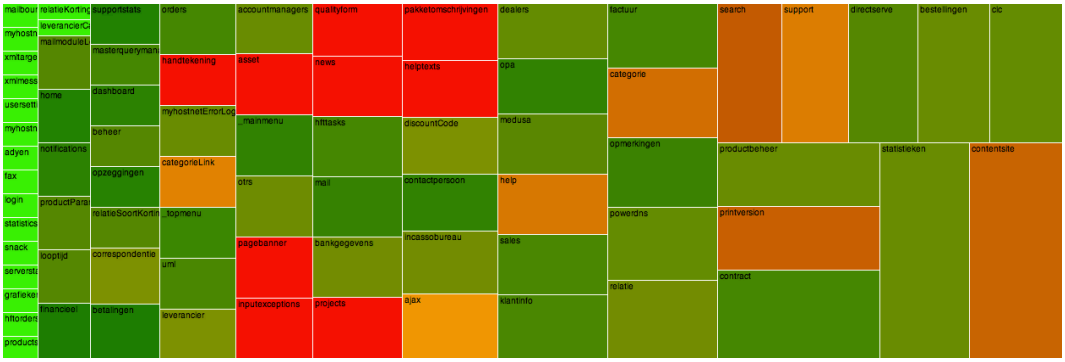

Figure 5.1: Aurora modules in treemap

executions of the files in the folder also indicates the certainty because if it is very low it could very well be that not yet all possibilities and thus files are used.

The process of selecting code for elimination consists of first searching for the biggest related parts of an application that are marked as possible dead by the dead code identification process. When a coherent possibly dead part is found with no files that were executed we should check wherefore the code is used and if it would be expected to have run. If this is not the case it is possible to remove it. If the feature corresponding to the code at hand is not expected to be executed yet we could either wait for the feature to be used or access it by hand if this has no negative side effects for the operation of the organization. If we have removed all coherent parts of the application only containing dead files we can go on removing files from within these related parts. Here we look at the number of executions for the directory containing the files we want to remove. If this is very low or the graph with the number of executed files over time shows that new files are still executed regularly it is not a good idea to remove files from this directory unless we know that the file should not be used any more and this gets confirmed by the identification process. This process can be viewed in Figure 5.2, the elimination part discussed in the next section is also visible in the figure.

5.2 Eliminating dead files

After we have selected code for elimination we cannot just remove the code and deploy the application. Before the code will be removed it will go trough a series of steps described in this section.

When eliminating code the use of a VCS is of great importance because this allows to track back which files and methods has been removed and restore them if necessary. A decent process would let the engineer decide which files to remove and put that code change up for review. In this step the files are only removed in the personal system or branch in the VCS. When the code review is approved, the change set will be committed to the current development branch. Now all engineers in the team have the code with the

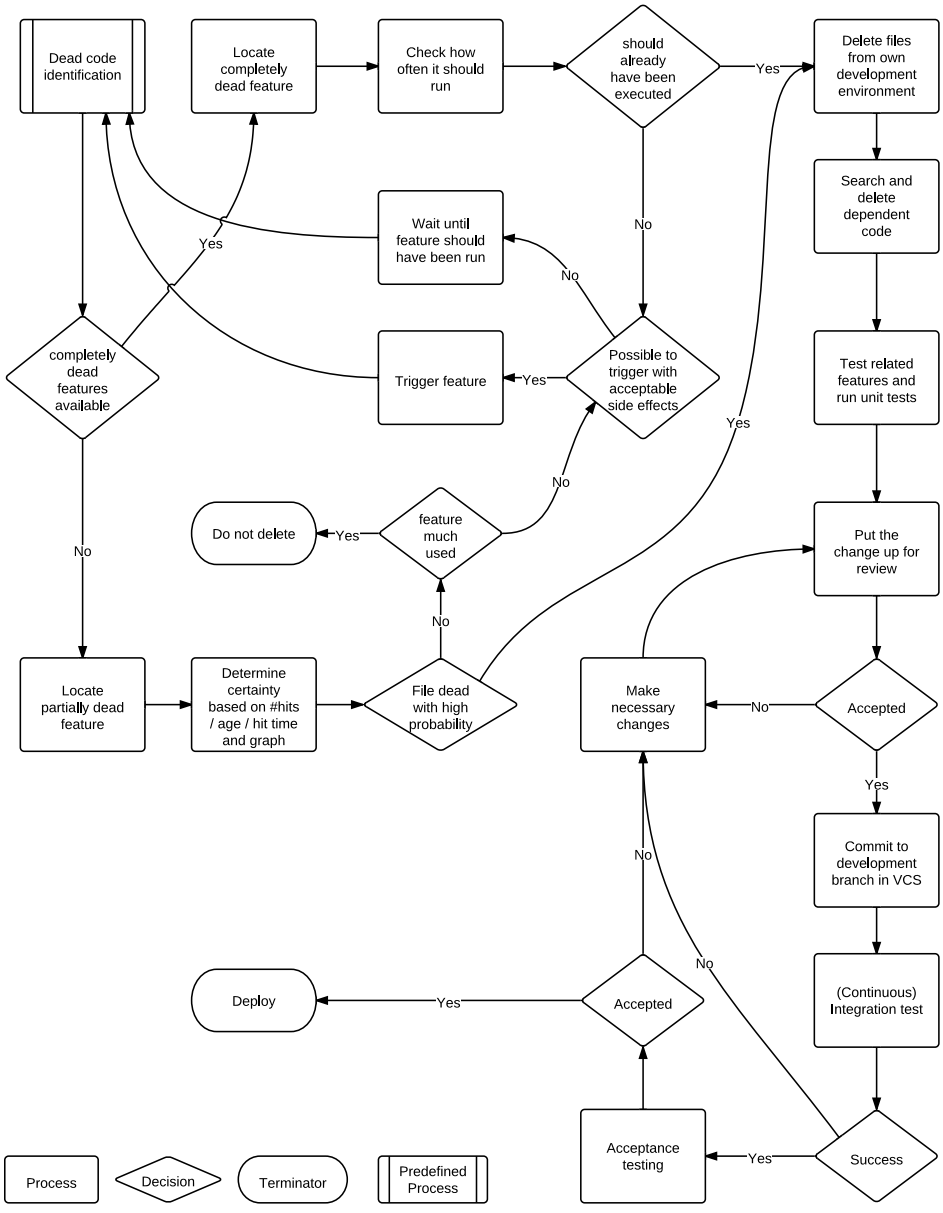

Figure 5.2: The process of selecting code for elimination and removing it

dead files removed. When everything is approved in integration and acceptance testing the change can be used in the next production version (see Figure 5.2).

If code from a dead file is used by other code that code is dead too, otherwise the file would not have been marked dead. Functions that call a class defined in the dead file can also be (partially) removed and so on. The Eclipse IDE with PHP Development Tools (PDT) or Zend Studio can help to find these dependencies, although the Dynamic Languages Toolkit (DLTK) does not offer advanced type inference so it is hard to find all references but it is a start. For now we use full text search of the class and function names. This obviously only works for unique function names. A possible solution for this problem is the identification of dead methods which is described as future work in chapter 8.

When the code is unit tested it is a lot easier to be sure that there are no side effects of removing unreachable code. The code can be removed and if there are no failing tests everything should be fine. The problem lies in removing unused features. Unused features are accompanied by a unit test that also should be removed.

If continuous integration [18] is used, it is best to use small commits to the VCS. This way it is more easy to find which files broke the unit tests. When continuous integration is used defects due to the elimination of files can be detected and handled early. Hostnet uses continuous integration for all projects.

5.3 Effect of wrongly removed code

Despite the effort to only remove code that will never be called upon again it is possible that some used code gets removed. When PHP tries to load a file that is not there it will stop execution with a fatal error and emit a HTTP 500 - Internal Server Error, as shown in Figure 5.3. These error pages are logged by PHP and it is possible to customize an error page so the call stack will be e-mailed to the developers.

If an engineer receives an email he can remove the part of code calling upon the deleted file or place it back from the source repository. It is worth noting that because it concerns a web application, not the whole application crashes and the user will not lose valuable work because the file was removed. At Hostnet we sent an e-mail report for every internal server error to the developers.

5.4 Ceteris paribus not possible

It is common when taking measurements to only vary one input at a time and keep all other things the same so the effect of that one input can be measured. This is called ceteris paribus, keeping all other things the same. Because the measurements take place in a live application it is not possible to freeze it from updates for at least several months to acquire the right measurements in a undisturbed situation. Especially in the fast moving web world this is impossible. Therefore we have to cope with the fact that there are updates coming in all the time. This implies that unused code is not always dead. It could be that it is just not used yet or that it is just added to the application but not enabled for the end-user yet. To be able to detect this last type it is possible to take VCS data into account. At Hostnet almost

Figure 5.3: HTTP 500 error page in the webshop

all projects are stored in SVN an the most recent ones are stored in git. Using this data it is possible to detect that a file is recently added to the system and therefore should not be declared dead. The date a file was last change in the VCS is displayed in the age column shown in Figure 4.1 as described in chapter 4. To get a better view on the certainty that new features will be accessed for a directory it is possible to look at the graph (also in Figure 4.1) which shows new files accessed over time. If all new files added to an application would also be displayed in this graph it would not give a good view of the certainty because files for newly added feature would be included making it look that it takes longer to stabilize then is the case in reality. By not adding files to the graph that has been added later on this problem is prevented. The files will still be displayed in the tree map for a complete overview of the application.

Chapter 6

Evaluation

In this chapter we will discuss and evaluate the proposed method for dead code elimination. The evaluation is done by implementing dead code elimination at Hostnet B.V.[1]. Hostnet is a Dutch web hosting company with its own software engineering department of about 10 engineers. The main language used at Hostnet is PHP because many people in the web hosting business are familiar with PHP and use it because it is easy to learn and adopt and facilitates rapid development.

We will evaluate both the data gathering part: dead code identification and the data processing part: dead code elimination. For the identification part we are mainly interested in the overhead of doing runtime analysis and in determining how long the analysis should be performed before we can say with sufficient confidence that code is indeed dead. The overhead is measured in the code identification part that is implemented in Aurora, the biggest application (in number of files) in use within Hostnet. Because the overhead is dependent on the number of files in use in the application should be the worst case within Hostnet. We compare 6 applications developed within Hostnet to see how long it will take until we can tell if the files that are not executed can be regarded as dead files. These application are all different in the way they are used, the number of features and how often they are used.

Both the visualizations and the method for dead code elimination are evaluated. The Eclipse plug-in is installed for all team members and used by 8 of them currently working on PHP projects. This plug-in is used in the development of all applications done at the moment. The Eclipse plug-in should not hinder the engineers and we would like to know if it helps the developers by letting them focus on files that are actually used. This is difficult to measure in numbers so we used a questionnaire to evaluate the Eclipse plug-in.

The web application visualization with the tree map is only used by 3 people that really removed code from Aurora. For the evaluation of the proposed process of dead code elimination (see chapter 5) we tried it on Aurora and managed to remove more then 28% (2740 files) of the files within just a day work. The evaluation of the visualisation will focus on how the web application is used by the engineers and how they base their decisions on the displayed information (see chapter 4 for a description of the user interface).

[1] `http://www.hostnet.nl`

For the evaluation of both visualizations a questionnaire is used. For the evaluation of the dead code elimination process as described in chapter 5, the evaluation is based on the observations made while the engineers removed the files and monitoring the application to see if there would be any bugs that were caused by the deletion of files.

The chapter will end with a discussion on the applicability of the method for dead code elimination described in this thesis and the threats to validity of the findings reported in the evaluation.

6.1 Overhead

The overhead caused by reporting all used files to the central database should be low enough to not be noticed by the end user of the application. At the end of each request from the browser for a new page, a list of used files is sent to the central database. Prior to building up the connection and sending the list, the response from the web application is flushed to the browser. This way the browser will already render the web page and only indicates that the page is still loading. This is convenient because even when the overhead is noticeable the end user can still access the application without delay.

We look at the overhead in time per page request. Profiling code has been added to the analysis in Aurora to measure the overhead. Aurora is the most used application within Hostnet and has the biggest number of files. The overhead in Aurora should be the worst case within Hostnet, as the time to send the list to the database and process it, is dependent on the size of the list and the load on the database server. Concurrent write operations to a single MySQL database on the same row will lock and create bigger load times when the load on the server increases. It is very likely that the same row will be updated in every request to the database because every request to Aurora will pass through the `index.php` file which is the starting point for the application.

When measuring the overhead, the time spent on the analysis is measured as well as the time required to build a connection to the database to be able to see if network congestion could be a limiting factor. A plot of this overhead can be viewed in Figure 6.1. On the x-axis the time spent in milliseconds is shown. On the left y-axis the frequency of the corresponding amount of overhead is displayed. In red the overhead for the creation of the connection to the MySQL database on the database server is plotted. The time needed to build the connection has an average of 1.6 ms with and standard deviation of 0.6 ms. The connection overhead seems normally distributed with a standard deviation of under 1 ms and without additional peaks which indicates there is no congestion on the network. In green the frequencies of total overhead have been plotted. The average overhead is 7.8 ms with an standard deviation of 153.5 ms. The blue curve denotes the probability that the overhead will be less than the corresponding value on the x-axis. Here we can see that in over 95% of the measured cases the overhead stays below 6 ms. The average overhead is right from the center of the overhead curve because sometimes the database is locked by an large query and the request has to wait for a long time occasionally.

When we use a slower method of storage, for example on disk storage in the database server we notice that two peaks will appear in the total overhead curve (see Figure 6.2).

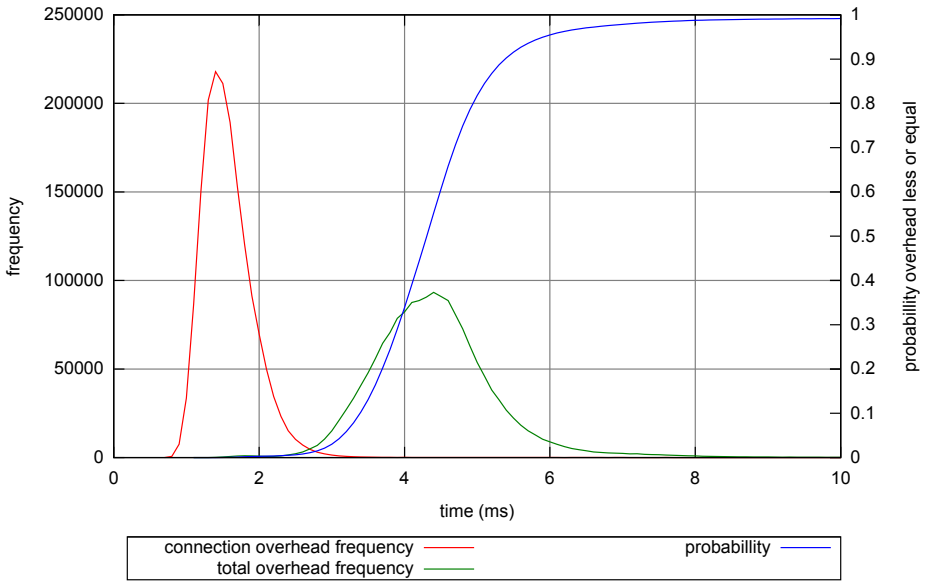

Figure 6.1: Overhead caused by measuring included files (in memory)

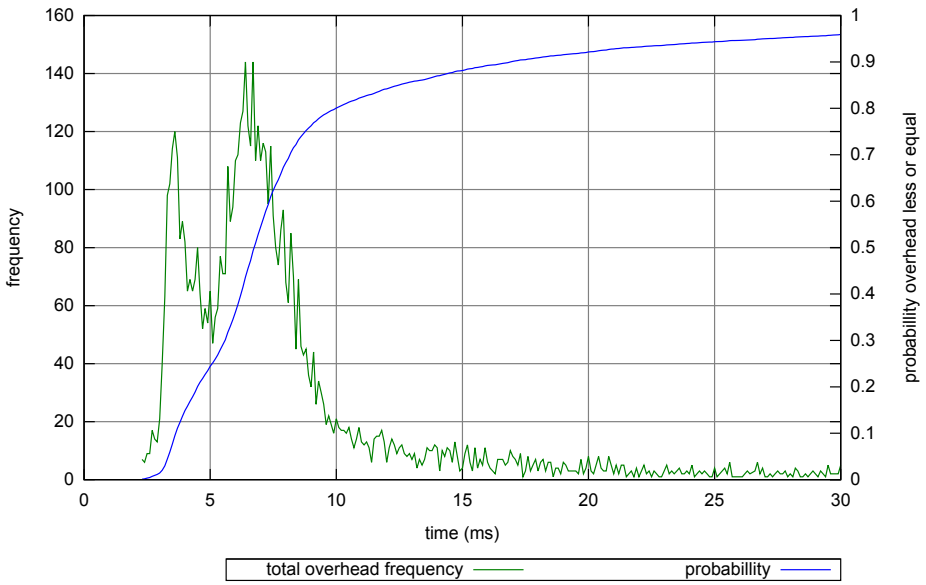

Figure 6.2: Overhead caused by measuring included files (on disk)

These appear because if the table is locked the query will have to wait a few milliseconds. Using disk storage, the average overhead is 27.5 ms with a standard deviation of 231 ms. Still in 80% of the cases the overhead is below 10 ms. In 95% of the cases the overhead is below 23 ms. The problem of locking rows is clearly visible here. The use of a memory table will not solve this problem but helps us determine where the peaks came from. For the analysis at Hostnet the slower on disk storage is used so that there is no need to build a backup mechanism to make sure that when the database server would be restarted the data would not be lost.

When the load on the database becomes too high other solutions must be found to keep the overhead within bounds so it will not be noticed. One solution could be to use multiple MySQL servers and aggregate the data between them less frequently then the normal updates occur. Another solution could be to aggregate the data on the web server side in shared memory and send it to the server at a frequency less then that of the page requests.

From the experience at Hostnet we can say that the overhead was not noticed by the helpdesk and account managers using Aurora for every day work even when using disk storage. We found that the limiting factor is not the project size but the frequency of requests because locks in the database will occur that create a bigger overhead than the time needed to send the list of files to the database.

6.2 Dead Code Identification

In this section we look at how long we have to wait until we can use the results of the dynamic analysis. Therefore a case study has been done. The overhead measurements in the previous section are gathered from this use case. For about three months 6 Hostnet applications are analysed. The applications differ in age, the number of users that uses them and the complexity in number of features they possess. In Table 6.1 the applications used for the case study are shown. The number of files and page requests are used as indication for the size and how much an application is used. We present the idea that small applications with a lot of users will offer certain results faster then a big application with little features. Further we check if the older applications within Hostnet have more dead code than the recently developed ones.

For every application a short introduction about the use of the application in the organization will be given together with a graph that visualizes the number of files used by the application over time. When this graph stabilizes it means that the probability that new files will be included decreases. The graph shows the number of files in the application accessed over time. The last data point in the graph denotes the end of the measurement and not a new file that is included.

All Hostnet applications are build as web applications, written in PHP. All tested applications make use of the Symfony[2] framework and are connected to a MySQL[3] Relational Database Cluster. Using Symfony also implies using the auto load functionality the framework offers. This makes all applications suitable for this analysis. The use of MySQL or

[2] `http://www.symfony-project.com`
[3] `http://www.mysql.com`

Symfony is not needed for the implemented identification method to work, only the dynamic loading of files is.

Applications in the use case

In Table 6.1 the 6 applications that will be used in the use case are shown. The applications are developed in the past 6 years. There are 4 applications, Aurora, My Hostnet, the Web shop and Mailbase that are accessed by humans, the other two, HFT2 and HFT3 are provisioning system that run (mostly) without user interaction. From the 4 web applications Aurora and Mailbase are only for internal use within Hostnet. The Web shop and My Hostnet are accessed by customers. The value of page requests for Aurora in the table has been adjusted to take the number of automated request from some monitoring systems into account. These systems are responsible for almost half of the requests but always request the same page. An example is the TV which shows real-time data about which customers are currently in the web shop.

Aurora

Aurora is the biggest (in files) and most feature rich application developed at Hostnet. It is used to store all customer information. Everything from contact information, bought products, contracts and invoices. Aurora takes care of a lot of the periodic tasks like creating invoices [8]. Aurora also offers all kind of statistics and development tools for the Software Engineering department. Aurora is used for daily business by the majority of employees, most work consists of looking up customer data, like contracts, contact data and products. The advanced function are only used once in a while.

The server has to take care of about 4 visits per second on average. This is because all helpdesk employees use Aurora for almost all their tasks and some TVs which show real-time statistics via Aurora (the automated requests from the TVs are excluded in Table 6.1).

Aurora is build up in clearly distinctive modules, batch jobs, database models, plug-ins and templates. All modules that are indicated as potentially dead and are not changed recently are probably dead. Before removing a module we need to know what its purpose is and if it was already expected to be used or not. If it was not expected to be used it can be removed. This shows that selecting files for removal requires human reasoning.

When looking at the graph for the application in Figure 6.3 we can see it rises quickly and then keeps growing steadily, the slope of the graph never reaches 0. At the end of

Name	Description	# files	Age	# page requests / day
HFT3	New provision system	750	1 year	n.a.
Web shop	web shop	923	3 years	39,643
Aurora	CRM application	9755	5 years	60,383
Mailbase	Lagacy mail filter frontend	490	5 years	2,803
My Hostnet	Customer portal	2422	5 years	53,495
HFT2	Provisioning system	3518	6 years	n.a.

Table 6.1: Applications ordered according to age

January an increase in the number of files is visible, here a new version is deployed and a cache rebuild took place. The cache rebuild used a lot of new files in the database model of Aurora. That the graph is increasing until date of writing means that new features are still being accessed every day even after 3 month time. For Aurora as a whole it is very likely that new files will be used in the future, however sub modules may be stable in less time. Because in Aurora as a whole new files are being accessed every day, it is not possible to draw a general conclusion for the whole application. For the application in general dead code removal is not yet possible but for clearly separated parts, like modules, that only contain dead files or frequently used functionality it is.

The first moment when the data becomes useful is at the point the slope decreases substantially and the graph bends. In the graph of Aurora this point is visible at the 14th of January. From this point all base functionality has been accessed. Aurora has a lot of monthly jobs, so we should at least wait a full month until we start using the data. After this time we can start with the dead code removal. The graph should not show sudden increases in the number of used files any more at this time, if it does there might be something wrong with the estimation of the frequency wherein features are used.

If we would look at the graph and make an educated guess, and take 5000 files as asymptote, we could calculate the probability of deleting a file that will be used in the future (when randomly choosing one). 9755 total files - 5000 alive files = 4755 dead files. 4983 files are not used yet. 4984 unused files - 4755 dead files = 229 alive files not accessed yet. 229 dead files not accessed yet / 4983 files not used yet = 0.046. This gives a chance of 4.6% that a file would be removed that was not dead when randomly selecting a file. These numbers are not included in the visualization because they are based on an educated guess

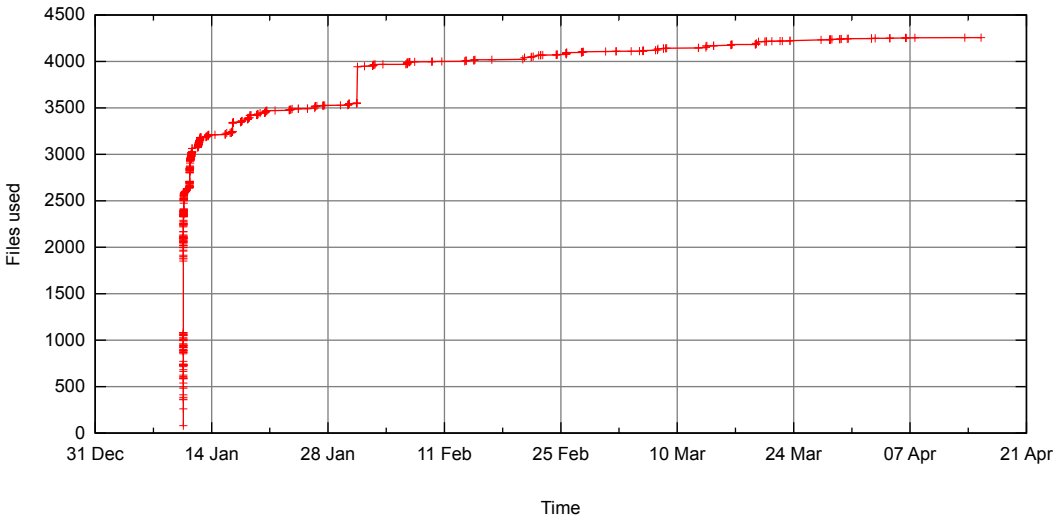

Figure 6.3: Number of used files over time in Aurora

of where the asymptote would be. Building a model to predict the probability of removing alive files at a given point in time is left as future work.

My Hostnet

My Hostnet is the customer portal of Hostnet. When somebody owns a domain name, virtual private server or hosting they can login to My Hostnet to view their address data, invoices, products and so on. It is also possible to change settings for products, upgrade products, pay invoicess or cancel contracts.

My Hostnet has a lot more different users than Aurora and far less files. The graph in Figure 6.4 shows a quick rise over just several days and then the slope is 0 for more than a month. Only couple of new files is included afterwards because a very rarely used feature has been accessed (in this case an error page that was resurrected for use in a new feature). Although new files may always be used in the future, the dead file identification in My Hostnet stabilizes after about 10 days. Then it is reasonable safe to remove the dead files according to the described dead code elimination procedure. We can conclude that we can start measuring after 10 days. All features of My Hostnet are expected to be used at least once a week. With the exception for error pages. The 10 days needed to stabilize is taken from the graph based on the fact that no new files were accessed afterwards for several months. When using the same method as for Aurora to calculate the probability as we did for Aurora the probability of removing an alive file when randomly selecting one approaches 0.

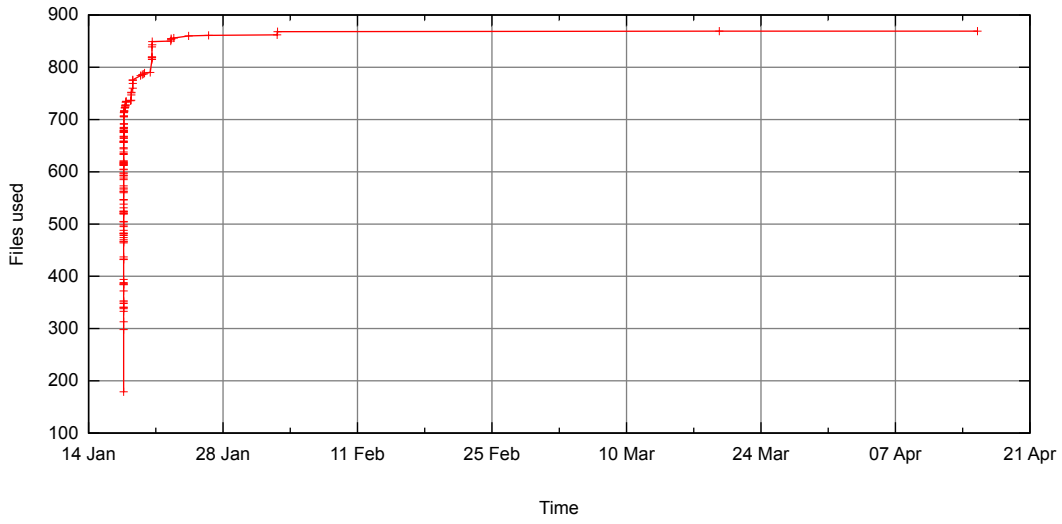

Figure 6.4: Number of used files over time in My Hostnet

Web shop

The web shop has a high number of page requests and a relatively low number of files (see Table 6.1), although the web shop is highly dynamic and completely different paths trough the web shop depending on where the customer came from. Because not all features of the web shop will be activated at all times and there are some very specific feature for rarely sold products, it is to be expected that the web shop will stabilize faster than Aurora but slower than My Hostnet.

When looking at the graph in Figure 6.5 we see that after 7 days still some new files were used. But we can also see there is increasingly more time between new files being included. Around 20 January the slope of the graph already was 0 but this was only for a few days. The distance between the point should be bigger than the expected frequency of the features of the application. For the web shop we know that some products are sold only once per month. It is possible, however, to start dead code elimination at this point as long as we keep in mind that all files related to products, especially the ones sold less often, could still be needed.

When looking into the measurements concerning the web shop we can see that a lot of code that is identified as dead code actually belongs to deactivated features. When new features will be activated they will only show up in the graph if the files were already deployed on the production server at the time the files were first indexed. Once again we see that dead code identification on the scale of files needs human reasoning. Here we can also see that when an increase is seen in the graph it may be worth investigating where it came from, because if it was caused by enabling a feature we can still consider the graph stable. In the future automatic notification of such events could be implemented.

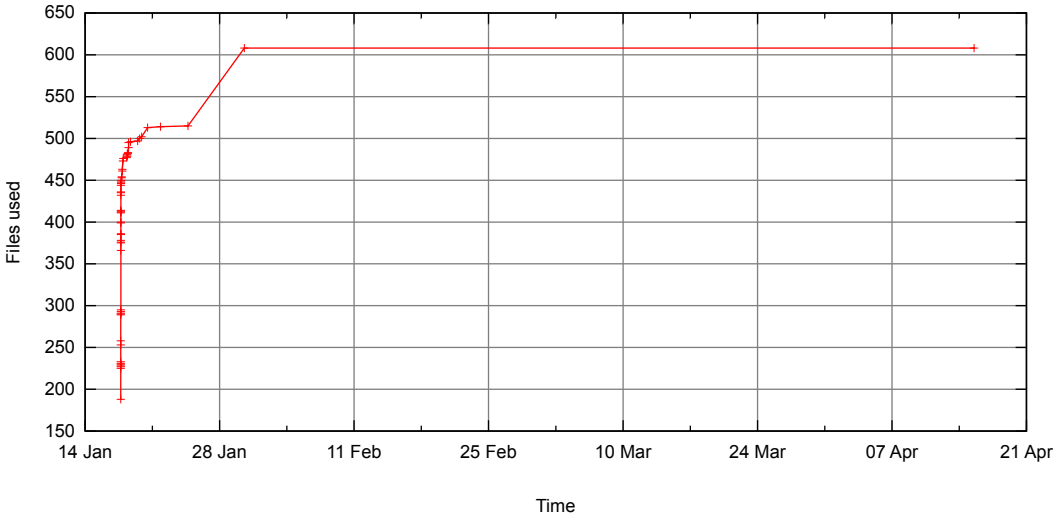

Figure 6.5: Number of used files over time in the web shop

HFT2

HFT stands for Hostnet Full Throttle and is the provisioning system used to order all domains and deliver them to the customers, the HFT2 also sets up hosting accounts and mails all needed information to the owner. There are a lot of products that can be delivered through the HFT2.

The HFT2 is the oldest application under inspection and is not yet (and never will be) fully converted into a Symfony application. This explains the low percentage of dead files; the unconverted part of the HFT2 uses static inclusion of the needed resources. This marks all included files as alive, not telling if they are actually in use.

As a consequence, the applied method only works well on the Symfony part (116 dead files). This, by no means, renders the method useless for the other part. Because the application is fairly aged a lot of old files that are never included can be found in the legacy part (935 dead files). The plots also show that not all inclusions are static in the old part of the application because new files are included over time.

When we look at the graph in Figure 6.6 we can see that the slope approaches zero around 28th of January but that still some new files are accessed later on. These files are visible as the crosses in the line. The last cross only denotes the end of the analysis. However if we would look at the Symfony part of the application we can see that no new files were included within a month. The files that were included in the old part of the application later on, are some specific products and the logout feature in the interface. This shows that we have to be careful when removing code that is related to products that are not regularly sold. The logout function that is used after some time shows that some features are seldom used. For seldom used features it is questionable if we should keep them. It is arguable that

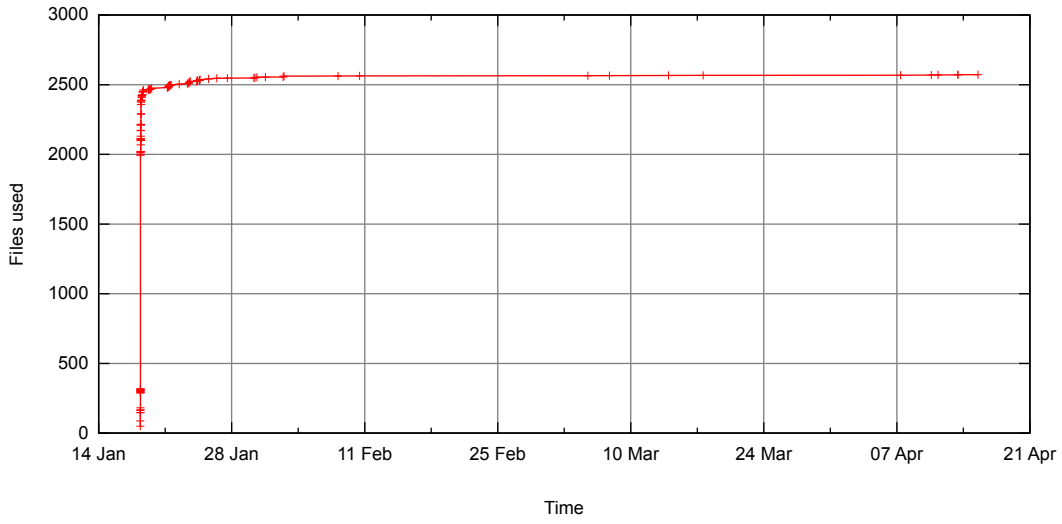

Figure 6.6: Number of used files over time in HFT2

a feature that is only accessed once in the three months should be removed because it costs more time than performing the task by hand.

From the employees of Hostnet we know that a part of the HFT2 is transformed into Symfony and an old legacy part is still there. In the visualization the legacy part of the application might be recognizable by the fact that most of the files have not changes for 6 years. The fact that there is a folder called Symfony in which all files are more recently added would confirm this.

HFT3

HFT3 is the successor of HFT2 and uses a new database model which is put in a shared repository. At this time HFT3 is the major application using the new database model but in the future more applications will be using it. Action should be taken log and accumulate data for shared code in a separate dataset which is used by all applications which make use of that code to see which code is in use. Libraries could also be analysed in the same way. At this moment we have to hard code different locations to store usage data in the dynamic analysis tools. It could be possible to automatically generate the location bases on VCS data in the future.

HFT3 is a lot smaller (in files) than HFT2, but it is used just as much. The work rounds in which HFT3 fetches data and execute its tasks are a lot shorter than in the previous version.

Allmost all dead files can be tracked down to the database. 807 dead files are located in an Object-relational mapping (ORM) plug-in. This plug-in is not included in the other projects because it is bundled with symfony, but the HFT3 uses a newer version so it is

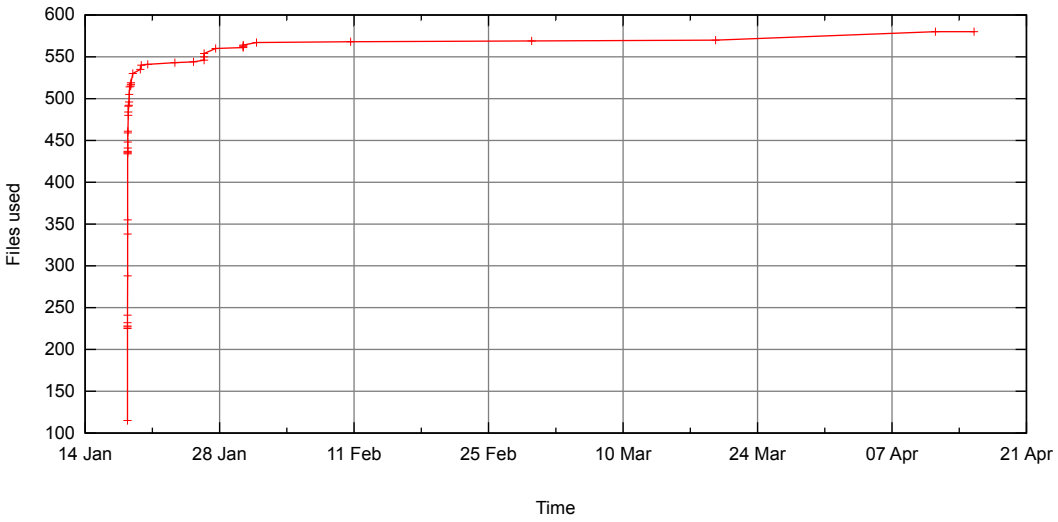

Figure 6.7: Number of used files over time in HFT3

included. The percentage of dead files when not taking this plug-in into account drops from 65% to 40%. This percentage is more in line with the measured values in the other applications and shows that newer applications probably have less dead code.

When we look at the graph for the HFT3 in Figure 6.7 we can see that over time new files are still being accessed. When we look further into the HFT we can see that some of the new files belong to an ORM plug-in (which should be outside of the project). The only new files accessed that does not belong to the ORM plug-in contains an Exception class which was used after one month and some files from a generated API. This leads to the conclusion that the HFT3 has to be measured for about one month to get sufficient certainty.

Mailbase

Mailbase is used to view and search all email that is send to and received from customers as well as mail to domain registrars. Only the user interface is under inspection. Mailbase is almost never used. The code is written using the Symfony framework and is the smallest (in number of files) of the tested applications.

Because Mailbase is used very little it is hard to know when new features will stop showing up. In the graph in Figure 6.8 we can see that the distance between new files being included does not lengthen yet, so there is no indication to assume the graph is stabilizing. This means we can not use the data for maintenance purposes as we can for the other applications.

At Hostnet only one person is still using Mailbase. Everything that he does not use will be removed from Mailbase and the functionality that remains will be ported to Aurora. In this case the only way to get accurate data is to ask the one person sill using the application.

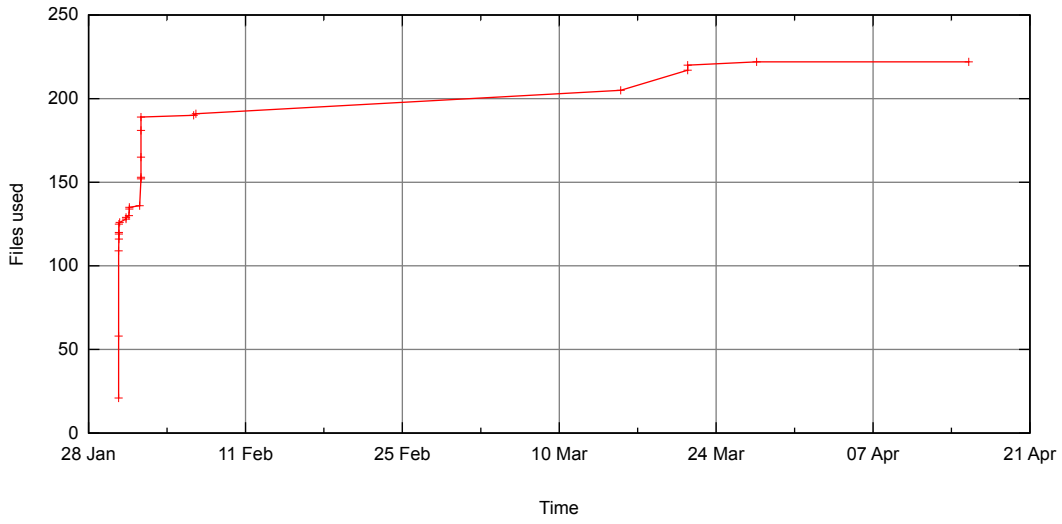

Figure 6.8: Number of used files over time in Mailbase

An option is to just port only the functionality used in the past 3 month and add all other functionality only on request with the old Mailbase code in the VCS as reference. This shows that depending on the goal it is possible to use the dead code identification method even when an application is used by only one user.

Results of the case study

Now we got data for all applications we can compare them and see if the applications behave as we thought within Hostnet. First we look at the age of the applications. We expected older applications to have a lower percentage of used code than the more recent ones.

In Figure 6.9 the graphs that were used to determine when to start with dead code elimination for every application in the use case are combined into one figure. In this figure the x-axis still denotes the time but it is given in days instead of dates, because not all use cases were started at the same time. This is also the reason for the fact that not all lines in the graph end at the same position. The y-axis uses a percentage instead of the number of files so it is possible to compare the applications in this graph according to percentage of used code. This figure shows that, in general, no conclusion can be drawn on how long you have to wait before starting dead code elimination.

When looking at Table 6.2 we see that the HFT2 does contain less unused code than the other applications even though it is the oldest one. This is because a big part of the HFT2 is statically included and denoted alive even if it would not be used at all. The HFT2 also has a newer part of the application which uses dynamic inclusion. This part shown as the HFT2 (Symfony) entry in the table. We also notice that the HFT3 uses less of its code then the web shop does despite of it being younger. This can be explained by the fact that HFT3 contains relatively much shared and generated code. The shared code can be found in the model of the HFT3 which also contains classes for a new mass mailer system.

In general we can say that older applications tend to have more dead code within Hostnet. We also see that code reuse is an important source of dead code in the Hostnet applications. Dead code is found in plug-ins for all applications.

The second prediction was, that small applications and much used application will reach a stable analysis faster than little used complex ones. We look at Table 6.3 to see if this is the case for the applications at Hostnet. The division of the number of files by the number of page views per day is used as combined measure of the complexity and how much the application is used. We compare this to the time needed to stabilize we found in the case study. We see that the prediction is only partly confirmed. The web shop took longer than was to be expected based solely on the number of features and users. We see that it is important to know what type of features an application posses. For example in the web shop some rarely sold products are available that require extra information from the client. The files needed for those products are expected to be only used once a month. This demonstrates that a rough estimation can be made on the number of files and users but that for a better estimation it is needed to know when the application features are expected to be used. We can also see that for Hostnet the time we had to wait to be certain ranges from 1 week to (at least) over 3 month. It is possible to already eliminate dead code from stable

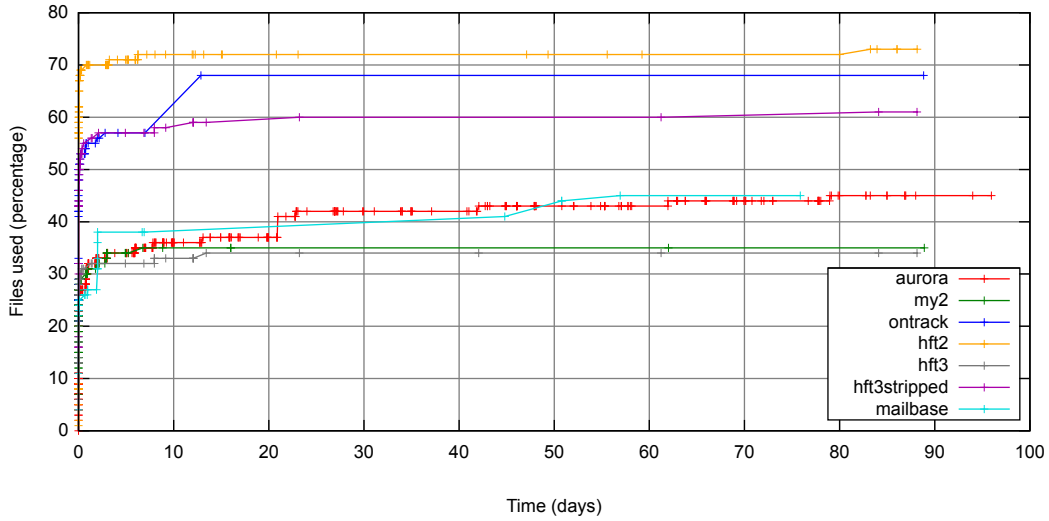

Figure 6.9: Ratio of used files per application

Name	Description	Age	% used
HFT3	New provision system	1 year	60.13%
Web shop	web shop	3 years	68.26%
HFT2 (Symfony)	Provisioning system	3 years	50.64%
Aurora	CRM application	5 years	48.91%
Mailbase	Lagacy mail filter frontend	5 years	45.31%
My Hostnet	Customer portal	5 years	35.92%
HFT2[4]	Provisioning system	6 years	73%

Table 6.2: Applications ordered according to age with percentage of used files

Name	page views / day	# files	# files/# page views	no new files accessed after
My Hostnet	53,495	2422	0.045	1 week
Web shop	39,643	923	0.023	2 weeks
HFT3	n.a.	750	n.a.	1 month
HFT2	n.a.	3518	n.a.	2 months
Aurora	60,383	9755	0.161	Not yet
Mailbase	2,803	490	0.171	uncertain

Table 6.3: Time to wait before now new files are accessed any more per application

[4]HFT2 uses static includes in the legacy part of the application which is not supported. Static included files always report as alive, increasing the percentage of used files significantly. Not all files are included statically so the method is still usable to identify dead code and eliminate it but comparison to the other applications does not make much sense.

Question:	fully agree very useful	agree useful	disagree useless	completely disagree completely useless	do not know
The tree map helps me to find dead code.	3				
I know where I am in the project structure when browsing the directory structure.	1	1	1		
I use the graph in my decisions about code removal.	1	2			
I use the table when looking for dead code.	2	1			
Rate the usefulness of the following entries in the table.					
path	3				
percentage of dead files		3			
number of dead files	1	1	1		
number of .php files	1	1	1		
number of hits	1	1	1		
age	3				
hit time	1	2			

Table 6.4: Questions about the tree map visualization

parts of the system. In the use case on dead code elimination (next section) we successfully removed dead files from Aurora despite the fact that new files are accessed every day.

6.3 Dead code elimination

In this section the dead code elimination process using the created visualization tools is evaluated. The process and tree map visualization is used by two people for a full day to test the described process (chapter 5) and remove dead parts of Aurora. The Eclipse plug-in is installed for all team members and has been used for a full month by the time of writing. Both visualizations and the process are evaluated using a questionnaire, because it is extremely difficult to measure objectively how the visualizations aids the developers eliminating dead code. The Eclipse plug-in could already reduce some of the maintenance cost of dead code without actually removing it by preventing the developers to mistakenly search for a bug or search for a functionality in a file that is not used.

For the visualization containing the tree map, table and graph of used files we look at how the various components of the interface are used and if the user finds them useful. This is done in the questionnaire with the questions that can be seen in Table 6.4.

As is visible in Table 6.4, the questionnaire was only taken by 3 people because it was

Question:	fully agree	agree	disagree	completely disagree	do not know
I have the dead files plug-in enabled all the time.	5	3			
I turn the plug-in only on for the purpose of removing files.			4	4	
The plug-in aids me when debugging applications.	4	2	2		
The plug-in aids me when developing new features.		5	3		
A file that is red can be removed.		1	4	3	
A fully green folder only contains live .php data .	5	3			

Table 6.5: Questions about the Eclipse plug-in

not possible to spare more people for a day of dead code elimination. Two of them worked one full day on removing dead code from Aurora using the tree map visualization. When we look at the answers we can see that the tree map points users in the right direction to find the unused files that can be removed and that it is found helpful. The same is true for the table and the graph. When looking at the columns in the table we can see that data that is already available in the tree map is rated less useful then data that is available only in the table. Also the total number of PHP files seems of less interest than the other data. Nobody rated a part of the interface completely useless. Possibly the interface could be improved to make it more clear which project and directory is currently viewed because this was not clear to all participants. Although it is not possible to draw general conclusions from such a small data set, it is possible to say that the visualization did help the engineers to locate, select and eliminate dead code.

We managed to reduce the size of Aurora by 28% in less then a day's work with two people. So far, no defects were found related to the removal of that dead code.

The Eclipse plug-in was also evaluated using the same questionnaire. For the Eclipse plug-in 8 people gave their opinion on how they use and like the plug-in. The participants consist of three senior developers, five junior developers. The questions and aggregated answers are in Table 6.5.

When we look at the answers (see Table 6.5), we observe that people have the plug-in turned on all the time and do not turn it off despite the fact that they were shown how that would be done. This leads to the conclusion that the plug-in does not get in the way of the developers. Next we want to know if the plug-in also aids the developers with their everyday work and if they understand what the colors mean. Most people find it especially useful when debugging, some others also when developing new features. When looking at the raw data it is possible to see that everybody found the plug-in useful for at least debugging or developing new features except for one person. The last two questions were added to test whether the participants knew what the colors precisely mean. A file that is

red is not used in production, but the fact that it is not used (yet) does not justify removal. A fully green folder does indeed only contain used .php files, although it is possible that it will contain other unused resources. We can conclude that in general people do know what the colors mean and find them useful in their everyday work.

6.4 Discussion

In this section the findings and short coming of the presented method for dead code identification and elimination will be discussed based on the case study results. Also applicability and threads to validity will be looked upon.

The implemented method should be enhanced to be able to measure a shared code base, this will not be much trouble. To also take statically included classes into account more work research should be performed if this is possible with an PHP plugin or that modifying the source is needed.

Outside PHP the method is applicable to all languages which use an auto loading mechanism to dynamically load classes and allow modification of this mechanism or an other method to list all loaded classes. This means the method is applicable for Java, but not for C and C++ programs.

The overhead observed is about 8 ms on average for the fast in memory storage and about 27 ms on average when using disk storage. Overhead only appears at the end of every page, all content is then already send to the client. Only the connection is not closed yet because this would end the execution of the PHP script. This results in a slightly longer visible load indicator in the browser, but all information will be available to the user without delay.

The use cases provided useful data on how to measure an application. They also showed that how long you have to wait before you get useful data is dependent on how many actions are performed on the system and how these actions are distributed along the features and the number of features. For the web shop, customer portal and provisioning system it is not needed to wait longer than 2 months.

When looking at the tested applications measured we observe that an older application usually has a higher percentage of dead code within Hostnet. This also is to be expected because more code remains as a program ages. This can be seen in figure 6.9 and in table: 6.2. This property can not just be generalized, because it depends a lot on how much effort is put in keeping the application small by the development team [38], but within Hostnet everything is build by the same team which makes it possible to compare the applications.

When removing code, searching for code referring to dead code is a tedious job, but the color decorator plug-in in Eclipse aids with this, because not only the files, but also the search results are decorated. This means that when the programmer searches for method invocations of method in a dead file all files containing those invocations will also be coloured in the search result. Automatically indicating the methods that call methods in dead files should be the next step to ease the programmers job.

Applicability

The discussed method for identifying dead files can be used in environments where files are dynamically loaded at run time. As long as files are only loaded when they are actually used it is possible to add tracing code to the application or use language features to get a list of all used files. The method will not work in environments where files are statically loaded at run time or are compiled into an executable.

In practice this method will not work for compiled languages like C and C++ because there will be no files any more when the the application is compiled into an executable. For languages that dynamically loads files like Java does for classes it is possible to apply this method.

The tree map visualization and the Eclipse plug-in can be connected to any data source about dead files. For example determining which images or other resources are used in a web application can be done by parsing the web server access log files and fed into the central database. The visualization tools created will now be able to show information for images without customization.

The procedure for dead code elimination is based on the fact that the analysis is dynamic. It should be equally valid for removing other resources or code with a different granularity as long as the data is dynamically gathered.

The tool box created for the case studies can be reused without modification in every PHP application that uses the auto load functionality instead of statically including all files.

Threads to validity

There are some important threads to validity of the results. The method will only perform well for applications which are used very much. It is not possible to get accurate data from an application only used a few times a week by a couple of users within a reasonable amount of time. A second thread is that the application should not make use of static file inclusion, this would not make the data less reliable but certainly less usable because all static included files will be marked as alive. The use of already available code coverage tools could be used to circumvent this problem in situations where an overhead of several times the normal execution time is no problem.

The toolbox and method are especially developed for use within Hostnet. The method has not been tested on applications outside of Hostnet.

When a comparison will be made between different applications it is important to only measure those files that are maintained by the team the measurement is done for because otherwise unused features of plug-ins or a framework will pollute the statistics.

Elimination of dead files sill is a humans job, this implies that somebody with better knowledge of the code will likely be more efficient in removing the dead code than somebody that is not familiar with the application at hand. This could compromise a clear view on how well this method performs and how the tools aid the developer. Someone who knows the application well can be more accurate than someone only using the data from the visualizations.

The current overhead data is only valid for a MySQL database. It could be worse when using an alternative way of storage for the central database. This said, MySQL is freely available to everybody and could be used when needed.

The usefulness of the visualization and the elimination of dead code is only evaluated within Hosnet with 8 developers from the software engineering department. It is not possible to draw conclusions from this data set although the results look promising.

Chapter 7

Related work

A lot of research has been conducted in the field of software optimization, making programs quicker, more memory efficient and smaller. Removing code can serve the purpose of a quick and small application with a memory footprint as small as possible but also the purpose of making the program easier to maintain and less prone to bugs which is the same purpose as in this thesis.

Srivastava [39] describes how the use of Object-Oriented programming can introduce unreachable procedures and describes a way to detect and remove them in C and C++ by building a static call graph. This method can not handle virtual functions. This problem is later addressed by Bacon [2] with research on fast static analysis of virtual functions calls in C++. Removing functions in such a context is even more important in newer languages which often declare functions virtual by default. Strivastava sees dead code elimination just as a way to enhance the performance of the application whereas Bacon sees it as a way to reduce the size and complexity of the application to improve human and automated analysis. Improving human analysis and comprehension is the goal in this thesis too, together with maintainability. Srivastava and Bacon focus on static analysis for compiled languages and detecting unreachable code whereas in this thesis is focused on the identification of unused code to improve maintainability which is only possible with dynamic analysis.

Debray [16] describes techniques for compilers to reduce executable size including redundant code elimination and unreachable code elimination. Over time many refinements and techniques for specific situations have been proposed. Liu [33] describes eliminating dead code in the presence of recursive data. Sunitha [40] uses value numbering to improve the dead code identification.

Biggar and de Vries [7, 6, 5, 15] created a method for static analysis for PHP with the purpose of creating a compiler with an dead code elimination step. This dead code elimination only removes dead code from the compiled program and does not offer dead function removal. In this research is mainly focused on gathering enough information to be able to compile PHP into C. There still is a dependency on PHP to solve some of the dynamic behaviour. When deciding to use dynamic or static analysis this work was used to see if it were possible to get a call graph from the code that could be used to find dead methods and classes. Because static analysis is used opposed to the dynamic analysis used in this thesis with this method only unreachable code can be eliminated. Because of the

dynamic behaviour of PHP, even eliminating unreachable code found with static analysis, can not be automated safely.

The tool phpCallGraph[1] could be used for unreachable function detection, but the program does not offer type inference and only supports situations where the type hinting of PHP applies, it does not take annotations into account. The program showed not to be very robust when building a call graph for any of the applications from the use cases. WebSSARI [23], Web vulnerabilities [44] and Pixy [29] also lack support for type inference or do not model PHP close enough to give to give data that could be used for dead code elimination in a reliable way [6]. These tools can be used to get methods invocations and type information to build a call graph which enables dead code identification but because they all do not very well model PHP the type information will often not be available making it impossible to build a proper call graph. By using dynamic analysis in this thesis we do not have the problem of missing information.

Two other projects that could be used for coverage analysis of a running program over a longer time could be Xdebug[2] and Zend Server[3]. Because here dynamic analysis is used these projects could used to gather data about alive files or methods and feed it into the visualizations created for this thesis. XDebug and Zend Server will be able to detect unused code as is the case in this thesis. Both record traces of the PHP application at runtime at the expense of longer running times and huge amounts of data. XDebug can not be used because the big overhead and Zend Server not because the amount of data is too big (see chapter 2).

When using virtual machines, optimization is possible in the Just-in-time (JIT) compiler. JIT compilers could also benefit from the runtime type information in detecting dead code. Efficient JIT execution of dynamically typed languages is discussed by Chang [11], this work could be used to identify dead code using runtime trace and runtime type information from the virtual machine. This is would be a dynamic extension of a static analysis to determine unreachable code, instead of a real dynamic analysis that is capable of detecting unused code too.

Lucca [17] proposes the use of both static and dynamic analysis to get a better understanding of an application. The combining of data from multiple ways of analysis is something also done for this thesis. Not only the dynamic data but also the VCS data is used. In the future also basic static analysis data could be added.

Knoop [31] and Briggs [9] describe how partial dead code can affect the performance of an application and how it can be removed. Their method relies on static analysis. Removing partial dead code is done to make the executable small and faster. Whereas this is also dead code elimination it doest not have the intention of improving human comprehension of the application or to ease maintenance.

Beyer [4] describes a way to uncover dead code via formal model checking and implemented their method as Eclipse[4] plug-in. This is on a far more abstract level than the work described in this thesis, but formal methods are equally valid for applications written

[1] `http://phpcallgraph.sourceforge.net`
[2] `http://www.xdebu.org`
[3] `http://www.zend.com/en/products/server`
[4] `http://www.eclipse.org`

in dynamic or static languages, however they do not uncover unused features. A lot of effort has to be put in to formally proving the correctness of an application, something not seen often in the fast moving web application business.

Janota [26] does not only consider the code in the reachability analysis but also takes annotations into account which pose extra limitations on the values of parameters. This way redundant and unreachable code can be detected that otherwise would have been considered reachable but in practice will be never executed because the variables will never hold the required values to enter a specific branch.

Besides dead code elimination methods for low level code and source in procedural/imperative languages, also dead code identification and elimination in functional languages has been researched. Genèves [20] looked into eliminating dead code from XQuery Programs. Damiani describes dead code elimination in functional languages using type inference [13] and rank 2 intersection [14]. Xi [43] also uses type information to detect and eliminate dead code.

Dynamically removing dead code is possible in hardware too. Butts [10] describes a way using flow detection to predict if an instruction will be dead, this information is used for the scheduling of the instructions within the processor.

Most of the dead code elimination techniques are used to optimize the application, where this thesis eliminates code to make the source code more maintainable and easier to test [22]. Maintainability often conflicts with optimizations as described by [1, 30].

Chapter 8

Summary, conclusions and future work

Software undergoes evolution which inevitably leads to software ageing and disuse of software pieces which leads to higher maintenance cost and less performance [12, 21, 22, 30, 38]. Elimination of dead code takes a big effort [1, 28, 38]. We look into dead code elimination for web applications written in dynamic languages. We discuss where, when and how to measure which code is dead, what the overhead is and what granularity is needed for the dead code identification (e.g. files, classes or functions).

In this thesis, we developed and evaluated methods and tools for supporting engineers in the detection and removal of dead code. We used dynamic analysis because it allows the detection of unused but reachable features and also because we deal with PHP, which is a dynamic language for which static analysis is very cumbersome [6, 7, 5, 15, 42]. We measure which files are used and translate this in a set of potentially dead files. The measurement is done by using native language features of PHP. PHP offers the possibility to get a list of all files used and execute code after every execution of PHP. This is used to send the usage data to a central database after every request to PHP. Two visualizations are built, one integrated in Eclipse, the other is a website containing a tree map to easily indicate where the most potentially dead code can be found. As result of a case study we know that how long we have to wait after the dead code identification phase to start with the dead code elimination is dependent on the size and type of application and how much it is used. The case study also shows that the visualizations are found helpful by the developers, and that the Eclipse plug-in prevents time being wasted on dead files while debugging or developing new features.

The identification, visualizations and dead code elimination procedure are thus far only used and tested within Hostnet. Future research should be conducted to find out how well the work done in this thesis is usable in other companies and projects. In the current tools shared code between applications has to be taken into account manually, in the future an automated way of detection of shared code between multiple applications could be build based on VCS data. For the dead code identification a statistical model should be build to indicate with which amount of certainty a file can be considered dead. During the identification phase sometimes new files are accessed after a while. A monitoring system could be

made to warn the developers when new files are accessed. We did not investigate the effects of lowering the granularity on the results. With a lower granularity it is possible to remove more dead code. The main concern when performing dynamic analysis with a granularity of a method is the overhead caused.

Bibliography

[1] B. Andreopoulos. Satisficing the conflicting software qualities of maintainability and performance at the source code level. In *WER-Workshop em Engenharia de Requisitos*, pages 176–188. Citeseer, 2004.

[2] D.F. Bacon and P.F. Sweeney. Fast static analysis of c++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–341, 1996.

[3] T. Ball. The concept of dynamic analysis. In *Software EngineeringESEC/FSE99*, pages 216–234. Springer, 1999.

[4] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. An eclipse plug-in for model checking. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 251–255. IEEE, 2004.

[5] P. Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, 2010.

[6] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916–1923. ACM, 2009.

[7] P. Biggar and D. Gregg. Static analysis of dynamic scripting languages. Draft Augustus 2009, 8 2009.

[8] H. Boomsma. Factureringsproces. Bachelor's thesis, TU Delft, 2008.

[9] P. Briggs and K.D. Cooper. Effective partial redundancy elimination. In *ACM SIGPLAN Notices*, number 6 in 29, pages 159–170. ACM, 1994.

[10] J.A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *ACM SIGOPS Operating Systems Review*, number 5 in 36, pages 199–210. ACM, 2002.

[11] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference, 2007.

[12] Y.F. Chen, E.R. Gansner, and E. Koutsofios. A c++ data model supporting reachability analysis and dead code detection. *Software Engineering, IEEE Transactions on*, 24(9):682–694, 1998.

[13] F. Damiani and P. Giannini. Automatic useless-code elimination for hot functional programs. *Journal of Functional programming*, 10(6):509–559, 2000.

[14] Ferruccio Damiani and Frdric Prost. Detecting and removing dead-code using rank 2 intersection. In Eduardo Gimnez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs*, volume 1512 of *Lecture Notes in Computer Science*, pages 66–87. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0097787.

[15] E. de Vries and J. Gilbert. Design and implementation of a php compiler front-end. Technical report, Trinity College Dublin, Ireland, 2007.

[16] S.K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):378–415, 2000.

[17] G.A. Di Lucca and M. Di Penta. Integrating static and dynamic analysis to improve the comprehension of existing web applications. In *Web Site Evolution, 2005.(WSE 2005). Seventh IEEE International Symposium on*, pages 87–94. IEEE, 2005.

[18] M. Fowler and M. Foemmel. Continuous integration, 2006.

[19] J.J. Garrett. Ajax: A new approach to web applications. Online, February 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php.

[20] P. Genevès and N. Layaïda. Eliminating dead-code from xquery programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 305–306. ACM, 2010.

[21] M.W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142. IEEE, 2000.

[22] Zhenyu Huan and Lisa Carter. Automated solutions: Improving the efficiency of software testing. In *IACIS*, pages 171–177. IACIS, 2003.

[23] Y.W. Huang, F. Yu, C. Hang, C.H. Tsai, D.T. Lee, and S.Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[24] W. Hudson. Breadcrumb navigation: there's more to hansel and gretel than meets the eye. *interactions*, 11(5):79–80, 2004.

[25] IEEE. IEEE Standard for Prefixes for Binary Multiples, 18 2009.

[26] M. Janota, R. Grigore, and M. Moskal. Reachability analysis for annotated code. In *Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 23–30. ACM, 2007.

[27] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*, pages 284–291. IEEE, 1991.

[28] C. Jones. The economics of software maintenance in the twenty first century (2006), 2006.

[29] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[30] M. Kiewkanya and P. Muenchaisri. Measuring maintainability in early phase using aesthetic metrics. In *Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems*, pages 1–6. World Scientific and Engineering Academy and Society (WSEAS), 2005.

[31] J. Knoop, O. Rüthing, and B. Steffen. *Partial dead code elimination*, volume 29. ACM, 1994.

[32] B. Lida, S. Hull, and K. Pilcher. Breadcrumb navigation: An exploratory study of usage. *Usability News*, 5(1):1–7, 2003.

[33] Yanhong Liu and Scott Stoller. Eliminating dead code on recursive data. In Agostino Cortesi and Gilberto Fil, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 83–83. Springer Berlin / Heidelberg, 1999.

[34] D.L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[35] The PHP Group. *PHP Online Manual*, 2011. http://nl.php.net/manual/en/.

[36] F. Potencier and F. Zaninotto. *The Definitive Guide to symfony*. Apress LP, 1.1 edition, 2010.

[37] B.L. Rogers and B. Chaparro. Breadcrumb navigation: Further investigation of usage. *Usability News*, 5(2):1–7, 2003.

[38] G. Scanniello. Source code survival with the kaplan meier. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 524 –527, sept. 2011.

[39] A. Srivastava. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):355–364, 1992.

[40] KVN Sunitha and VV Kumar. A new technique for copy propagation and dead code elimination using hash based value numbering. In *Advanced Computing and Communications, 2006. ADCOM 2006. International Conference on*, pages 601–604. IEEE, 2006.

[41] E. Tempero. An empirical study of unused design decisions in open source java software. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 33–40. IEEE, 2008.

[42] L. Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[43] Hongwei Xi. Dead code elimination through dependent types. In Gopal Gupta, editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 228–242. Springer Berlin / Heidelberg, 1998.

[44] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium*, pages 179–192, 2006.

Glossary

CGI Common Gateway Interface. 12

CLI Command Line Interface. 3, 12

DLTK Dynamic Languages Toolkit. 24

IDE Integrated Development Environment. 3, 9, 15, 18, 24

JIT Just-in-time. 46

ORM Object-relational mapping. 36, 37

PDO PHP Data Objects. 13

PDT PHP Development Tools. 24

SVN Subversion. 13, 24

VCS Version Control System. 1, 7–10, 13, 17, 22, 24, 25, 36, 38, 46, 49

List of URLs

Auto_append_file directive in php.ini
http://php.net/ini.core.php#ini.auto-append-file.

Bash
http://www.gnu.org/software/bash.

Cron
http://linux.die.net/man/5/crontab.

d3.js
http://mbostock.github.com/d3.

Eclipse
http://www.eclipse.org.

Hostnet B.V.
http://www.hostnet.nl.

Hostnet Open Source
https://github.com/hostnet.

Makefile
http://www.gnu.org/software/make/manual/make.html.

MySQL
http://www.mysql.com.

PHP
http://www.php.net.

phpCallGraph
http://phpcallgraph.sourceforge.net.

SQLite
http://www.sqlite.org.

Symfony
http://www.symfony-project.com.

Xdebug
http://www.xdebu.org.

Zend Server
http://www.zend.com/en/products/server.

Appendix A

PHP dynamic analysis

This is an example file for the recording all the used files from the PHP application. This file could be enhanced to put shared code from multiple applications in the same location.

```
// push all data to the browser
ob_flush();
flush();

// Connect to the database
$db = mysql_connect('db.hostnet.nl','username','password');
mysql_select_db('deadfiles',$db);

// Collect the included files and put them into a string for ...
   SQL
$files = implode('\',\'',get_included_files());

// Create the SQL and execute
$query = "UPDATE aurora SET count = count + 1, first_hit = if(...
    first_hit IS NULL, NOW(), first_hit)  WHERE file IN ('...
   $files')";
mysql_query($query,$db);
mysql_close($db);
```

Appendix B

Raw questionnaire data

On the next page all answers to the questionnaire are grouped in one table. Eight people answered the questionnaire. Three people made a remark. The remarks are visible below and belong to survey number 3, 6 and 8 in the table on the next page. The fist remark was:

> Nice plugin, awsome tool to check if it's worthy to add new features to a file or not, because now you know if the file is actually being used or not.

The second, less serious one, was:

> Cheers!
> Ajax!!

The last one was:

> 15: Iff a file is dead, the file may be removed, but all code that refers to this file is also dead.

Question:	1	2	3	4	5	6	7	8
How many years have you worked in the Software Engineering field?	> 4 y	1 – 2 y	> 4 y	0 – 1 y	0 – 1 y	> 4 y	> 4 y	> 4 y
Which type of work did you do in the past two months	pfb	pf	fbo	fb	fbo	box	x	pfbo
What is your position within the Hostnet Software Engineering team?	s	m	j	j	t	o	s	
The tree map helps me to find dead code				fa		fa	fa	
I know where I am in the project structure when browsing the directory structure				fa		da	a	
I use the graph in my decisions about code removal				fa		a	a	
I use the table when looking for dead code				a		fa	fa	
Rate the usefulness of the following entries in the table								
path				vu		vu	vu	
percentage of dead files				u		u	u	
number of dead files				u		ul	vu	
number of .php files				vu		ul	u	
number of hits				vu		ul	u	
age				vu		vu	vu	
hit time				vu		u	u	
I have the dead files plug-in enabled all the time	a	fa	a	fa	fa	fa	fa	a
I turn the plug-in only on for the purpose of removing files	d	d	cd	cd	d	cd	cd	d
The plug-in aids me when debugging applications	fa	d	fa	fa	a	a	fa	d
The plug-in aids me when developing new features	a	a	a	a	d	d	fa	d
A file that is red can be removed	d	a	d	d	cd	cd	cd	d
A fully green folder only contains live .php data	fa	a	a	fa	fa	fa	fa	a

y = year(s)				
p = projects	f = features	b = bug fixes	o = operational	x = other
s = senior	m = medior	j = junior	t = team leader	x = other
fa = fully agree	a = agree	d = disagree	c = completely disagree	x = do not know
vu = very useful	u = useful	ul = useless	vul = very useless	x = do not know

Appendix C

Help for Toolbox

C.1 main command

```
Dynamic Dead Code Detection Tools For PHP5
Developed by Hostnet B.V.
http://www.hostnet.nl
Hidde Boomsma
hidde@hostnet.nl

Usage:
  /usr/bin/dead [options]
  /usr/bin/dead [options] <command> [options] [args]

Options:
  -q, --quiet                       don't print status messages to stdout
  -d dsn, --dsn=dsn                 The Data Source Name
                                    <driver>://<username>:<password>@<host>:<...
                                        port>/<database>
                                    ex.
                                    "mysql://user:pass@my.server.nl/database"
  -u username, --user=username      Username for the datasource
  -p password, --password=password  Password for the datasource
  -t table, --table=table           the table in the database where you
                                    want to store information or fetch it
                                    from
  -i, --information                 display memory and run time information
                                    after the command has finished (on
                                    stderr)
  -o output, --output=output        the output file (use - for stdout)
  -h, --help                        show this help message and exit
  -v, --version                     show the program version and exit

Commands:
  color  generate a colors.txt file for use in eclipse
  prime  read a directory path and store it in the database
  tree   puts aggregated data in the database
  stats  display some basic statistic information from the table
  json   get json for the treemap for a specific path from a cashe tree
         database
  graph  create various visualizations
```

C.2 color subcommand

```
generate a colors.txt file for use in eclipse

Usage:
  /usr/bin/dead [options] color [options]

Options:
  -p workspace, --workspace_path=workspace  Cut off this string from the
                                            path generated for colors.txt
                                            to match your workspace
                                            checkout
  -h, --help                                show this help message and exit
```

C.3 prime subcommand

```
read a directory path and store it in the database

Usage:
  /usr/bin/dead [options] prime [options] <path>

Options:
  -h, --help  show this help message and exit

Arguments:
  path  The path to read the files from for priming
```

C.4 tree subcommand

```
puts aggregated data in the database

Usage:
  /usr/bin/dead [options] tree [options]

Options:
  -t table, --table=table  Table where the aggregated data should be saved
  -h, --help               show this help message and exit
```

C.5 stats subcommand

```
display some basic statistic information from the table

Usage:
  /usr/bin/dead [options] stats [options]

Options:
  -d date_format, --date_format=date_format     The format to use for the
                                                date fields
  -l latex_prefix, --latex_prefix=latex_prefix  A prefix for the latex
                                                command generated
  -u, --utc                                     display dates and times in
                                                UTC
  -f format, --format=format                    format to use for output
  --list-format                                 lists valid choices for
                                                option format
  -h, --help                                    show this help message and
                                                exit
```

C.6 json subcommand

```
get json for the treemap for a specific path from a cashe tree database

Usage:
  /usr/bin/dead [options] json [options] <path>

Options:
  -h, --help  show this help message and exit

Arguments:
  path  path to get the json for
```

C.7 graph saturation subcommand

```
gives a graphical representation of how many new files are used over time

Usage:
  /usr/bin/dead [options] graph [options] saturation [options] <tables...>

Options:
  -p path, --path=path  only inspect files under this path
  -s, --scale           Scale the graph to days and % all with all
                        applications starting at 0 days
  -h, --help            show this help message and exit

Arguments:
  tables  tables that should be included
```

Appendix D

Eclipse plugin input file: colors.txt

Example of a `color.txt` input file for Eclipse[1]. The format is simple, every line starts with the percentage (0 – 100) followed by the path of the file relative to the project. Every file is on a separate line. This file is created with the color subcommand (described in appendix C.2).

```
25 /
50 app
50 app/blog
0 app/blog/category_app.php
100 app/blog/post_app.php
0 index.php
0 url.php
```

[1] `http://www.eclipse.org`

Appendix E

Update scripts

To update the color files and aggregated tree tables in the database some update scripts are used. These are very simple Bash[1] scripts which are executed by Cron[2]. These scripts are included here because they show nicely how the toolbox is used. The `update_colors` and `upate_tree` script are used to update the information for the developers. The other two, `update_help` and `update_thesis` are solely used to update contents of this thesis. These two scripts are included only to illustrate how the toolbox can be used.

E.1 update_colors cron file

```
#!/bin/bash
/usr/bin/dead -t aurora color   -p /home/ontw/aurora/www   > /pool/groups/se/...
    AURORA/colors.txt
/usr/bin/dead -t my2 color      -p /home/ontw/my2/www      > /pool/groups/se/...
    MYHOSTNET/colors.txt
/usr/bin/dead -t ontrack color  -p /home/ontw/ontrack/www  > /pool/groups/se/...
    Bestelmodule/colors.txt
/usr/bin/dead -t hft2 color     -p /home/ontw/hft2/www     > /pool/groups/se/...
    HFT/COLORS_HFT2/colors.txt
/usr/bin/dead -t hft3 color     -p /home/ontw/hft3/www     > /pool/groups/se/...
    HFT/COLORS_HFT3/colors.txt
/usr/bin/dead -t mailbase color -p /home/ontw/mailbase/www > /pool/groups/se/...
    MAILBASE/colors.txt
```

E.2 update_tree cron file

```
#!/bin/bash
/usr/bin/dead -t aurora tree
/usr/bin/dead -t my2 tree
/usr/bin/dead -t ontrack tree
/usr/bin/dead -t hft2 tree
/usr/bin/dead -t hft3 tree
/usr/bin/dead -t hft3stripped tree
/usr/bin/dead -t mailbase tree
/usr/bin/dead -t symhostnet2 tree
/usr/bin/dead -t test tree
```

[1] http://www.gnu.org/software/bash

[2] http://linux.die.net/man/5/crontab

E.3 update_help cron file

```
#!/bin/bash
DEAD='/usr/bin/dead'
COMMANDS="color prime ast tree stats json"

for CMD in $COMMANDS;
do
  $DEAD $CMD --help > "/home/ontw/dead/thesis/tex/toolbox/$CMD.txt"
done

`$DEAD graph saturation --help > /home/ontw/dead/thesis/tex/toolbox/graph.txt`
`$DEAD --help > /home/ontw/dead/thesis/tex/toolbox/dead.txt`
```

E.4 update_thesis cron file

```
#!/bin/bash

PATH="/home/hboomsma/usr/:$PATH"
#PHP="/opt/eclipse/plugins/com.zend.php.debug.debugger.linux.x86_64_5.3.18....
    v20111214/resources/php53/php -c /home/hboomsma/php.ini"
PHP="/opt/eclipse/plugins/com.zend.php.debug.debugger.linux.x86_64_5.3.18....
    v20120224/resources/php53/php -c /home/hboomsma/php.ini"
INKSCAPE="inkscape"
DIR="/home/hboomsma/projects"
DEAD="$DIR/dead.phar"
IMGDIR="$DIR/dead/thesis/img/usecase"
TEXFILE="$DIR/dead/thesis/tex/stats.tex"
TABLES="aurora my2 ontrack hft2 hft3 hft3stripped mailbase"

rm $TEXFILE

for TABLE in $TABLES; do
  $PHP $DEAD -o /tmp/$TABLE.svg -t $TABLE graph -w 800 -h 400 saturation
  $INKSCAPE -f /tmp/$TABLE.svg -A $IMGDIR/${TABLE}_saturation.pdf
  $INKSCAPE -f /tmp/$TABLE.svg -E $IMGDIR/${TABLE}_saturation.eps
  rm /tmp/$TABLE.svg
  $PHP $DEAD -t $TABLE stats -f latex >> $TEXFILE
done

$PHP $DEAD -o /tmp/all.svg graph saturation -s $TABLES
$INKSCAPE -f /tmp/all.svg -A $IMGDIR/all.pdf
$INKSCAPE -f /tmp/all.svg -E $IMGDIR/all.eps
rm /tmp/all.svg
```